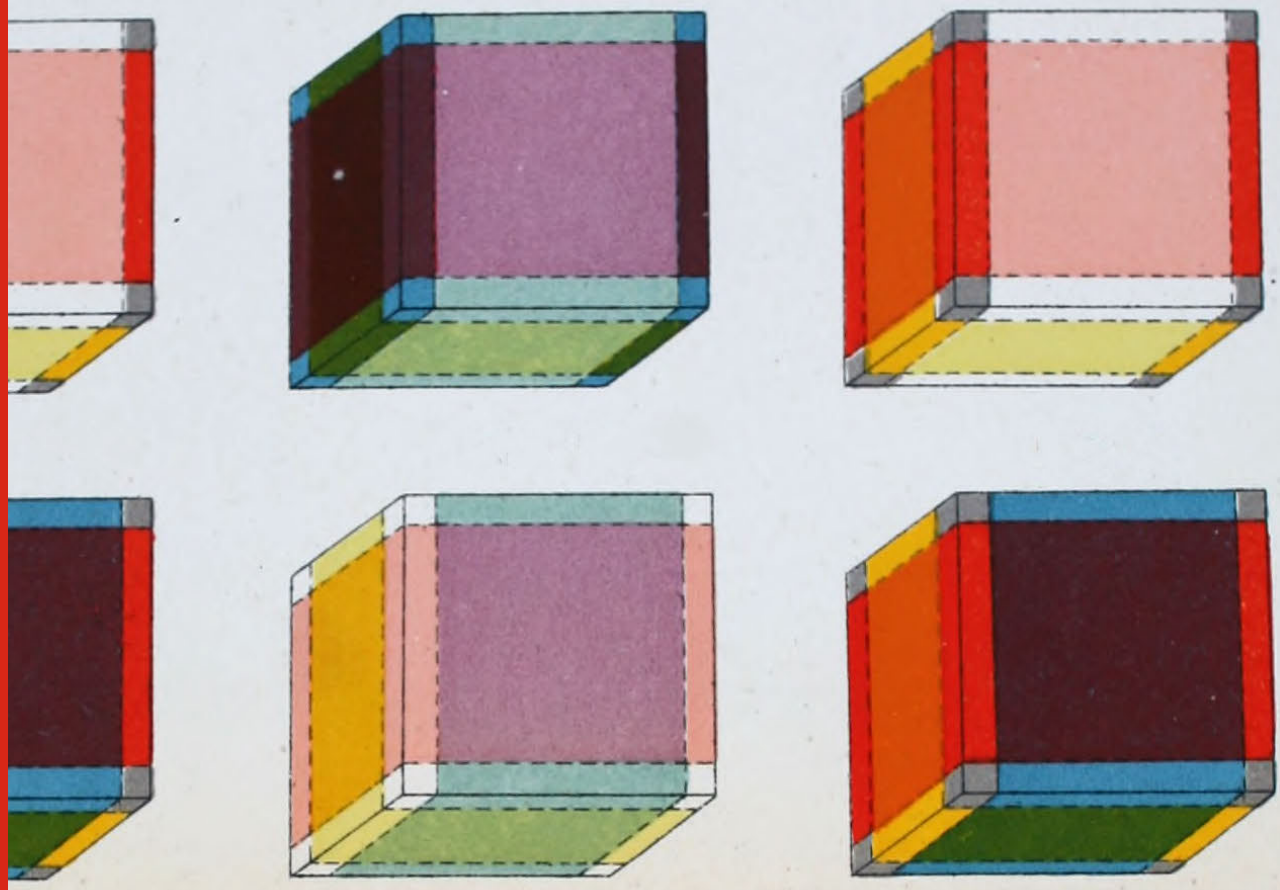
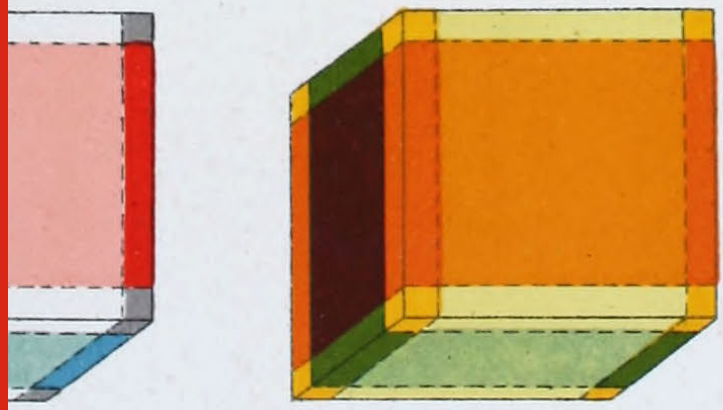
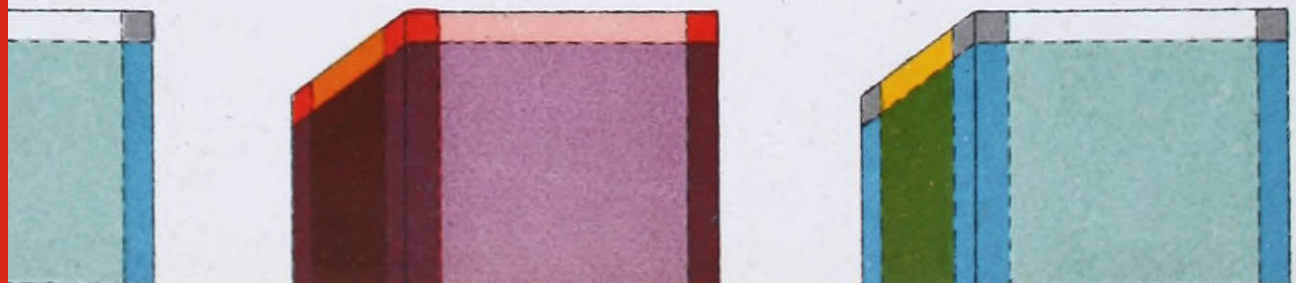


EVENT SOURCING IN LARAVEL

A Beyond CRUD strategy

By Brent Roose





EVENT-SOURCING-LARAVEL.COM SPATIE

EVENT SOURCING IN LARAVEL BY SPATIE

A hands-on course to start using event sourcing in large apps.

Brent Roose

Published 2021 by Spatie.

Reviewers: Freek Van der Herten, Matthias Noback, Stephen Moon, Wouter Brouwers
Design & composition: Sebastian De Deyne, Willem Van Bockstal

This book is a distillation of the lessons learned by the Spatie team the past two years:
Adriaan, Alex, Brent, Jef, Freek, Niels, Rias, Ruben, Sebastian, Willem, and Wouter.

Version 12a70f4-1
To report errors, please send a note to info@spatie.be.

Created in Belgium.

TABLE OF CONTENTS

PREFACE

Do you know the feeling of looking back on your old code and thinking to yourself, “what was I doing?” I doubt there will ever be a project where I wouldn’t want to change anything after a few years or even start over again from scratch. Whether that’s a productive thing to do is another matter because often it isn’t the wisest choice, but that’s OK. Still, I can’t shake the feeling of wishing I knew better “back then”.

I’ve begun to use those feelings whenever I start a new project. I think about previous projects and their difficulties, and aim to avoid them now that I have a clean slate. Looking back on the last three projects I’ve done (each taking about two years), I can see how that strategy has worked. Every new project has been better and more polished, and every new project has been built on the knowledge of previous ones. I doubt this learning process will ever stop: when my knowledge and skill grows, I’ll be able to tackle more complex projects, forcing me to improve further.

To me, this “lifecycle of a developer’s ability”: the constant learning, growing and improving, discovering new things, again and again, is what makes this job worthwhile.

This book describes the next step in my developer’s journey and aims to teach you as well. I consider it the successor to my previous book “Laravel Beyond CRUD” — you don’t need to read it to follow along with this one, and this book also doesn’t make Laravel Beyond CRUD obsolete. In the previous book, my colleague Freek stated: “*There’s no approach that is inherently right or wrong*”.

I can wholeheartedly agree with this statement: there's no right or wrong way. However, some techniques might be better suited for specific projects, of course, and this book will explore one of those techniques: a way of designing projects with very complex processes that make them highly flexible and sustainable. You'll learn new ways of thinking about the "program flow" and simplify what seems to be extremely complex processes.

On reading this book

All examples are written in PHP 8, and most of them will be in the context of a Laravel application. However, you don't need any prior knowledge of them to follow along; you can apply the same patterns and principles in any project or framework.

Whether you already have theoretical or practical experience with event sourcing, or whether you've never even heard of it, you'll be able to learn from this book. We'll start with the very basics of events and an event-driven mindset, then we'll introduce the foundations of event sourcing and the patterns that come with it, and finally, we'll cover in-depth topics.

So if you're already familiar with event sourcing, you could choose to skip the first part of this book or come back to it after you've read the more complex topics.

TESTING

Whenever you encounter a block like this, it'll show you how to test specific components. Specifically, the first part of this book will focus on all the building blocks needed in event-driven design and event sourcing; it's those chapters where you'll find these dedicated testing blocks.

One feature we'll often use when writing tests are test factories. They look like this:

```
CartItemAddedFactory::new()
    ->withProduct(
        ProductFactory::new()->withPrice(10_00)
    )
    ->create(),
```

Unlike Laravel's built-in model factories, we will need a more general approach to build test objects that aren't models, objects like events, for example. That's why we'll use these simple factory classes. If you're interested in how they work, you can check out the source code that came with the demo application.

Keep in mind that testing is *crucial* when building event-driven applications. I know it's sometimes tempting to skip testing and come back to it later, so I've done my best to keep examples as short as possible and directly to the point.

Don't skip tests; you'll save yourself a lot of trouble.

Demo app

Bundled with this course comes a demo app: a shopping cart. This package has grown to fit the needs of this book but could also be a stepping stone to start your new real-life shopping cart project. We'll often refer to the demo app throughout this book, so definitely keep it close by.

I realise that it is impossible to write a shopping cart that fits everyone's needs. Based on my own experience, I know that there are often lots of tiny details and intricacies that come with a client's business, and it wouldn't make any sense trying to code all of those possibilities in one package. That's why, if you want to use this code as the starting point for your new project, you're free to use and change it in whatever way your project requires.

With all of that being said, it's time to dive in. I hope you have fun!

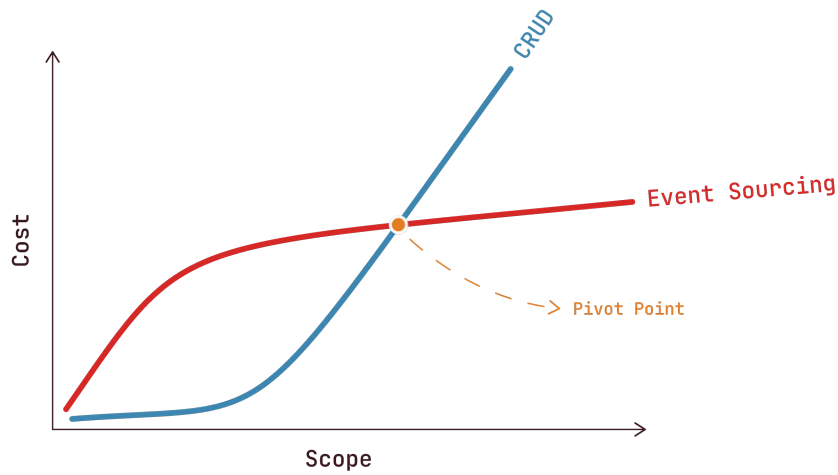
INTRODUCTION

To many developers, event sourcing is a magical beast that's used to build extremely complex, often distributed projects. And there's good reason for that: event sourcing is a pattern that forces code to be built in a way that fits those complex projects exceptionally well.

Words like "modularized", "distributed", "scalable" and "versatile", come to mind to describe it - characteristics you can't do without if you're building applications at scale. Think about a popular webshop or a bank, handling maybe thousands if not millions of transactions per second. Event sourcing is most often associated with those kinds of projects.

And thus, event sourcing is rarely ever used in smaller projects: the ones many developers deal with, the ones many of my regular blog readers work on daily. I think this is because of a fundamental mistake of what we think event sourcing is.

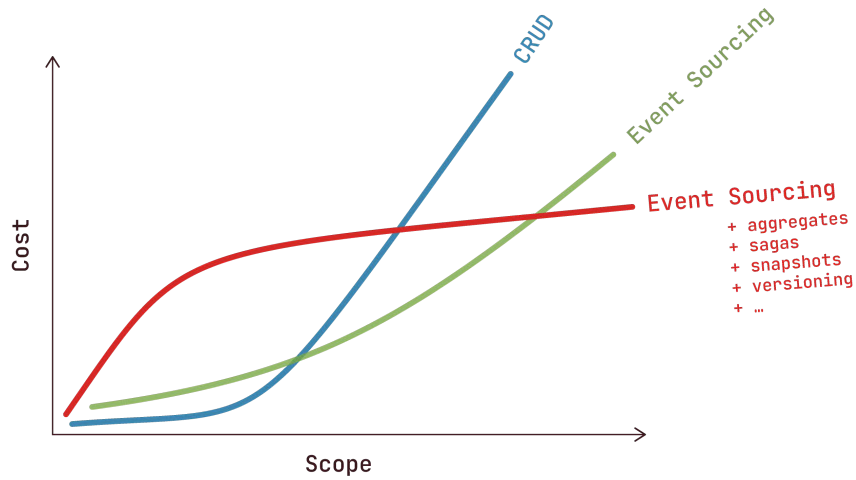
When I discuss event sourcing, people often assume that it comes with significant overhead and that that overhead isn't justified in smaller projects. They say that there's some pivot point, determined by the scope of the project, where event sourcing reduces costs, while in smaller projects, it would introduce a cost overhead.



And, of course, this is an oversimplification, but it visualizes the argument very well; we should only use event sourcing in projects where we're sure it'll be worth it.

The problem with this statement is that it doesn't talk about *just* event sourcing. It talks about event sourcing *with* all its associated patterns: aggregates, sagas, snapshots, serialization, versioning, commands etc.

This is why I'd say our graph should look something more like this:



But does it even make sense to use event sourcing without the patterns that build on top it? There's a good reason why those patterns exist. Those are the questions I want to answer today.

Here's Martin Fowler's vision on what event sourcing is:

> *We can query an application's state to find out the current state of the world, and this answers many questions. However, there are times when we don't just want to see where we are, we also want to know how we got there.*
>
> *Event Sourcing ensures that all changes to application state are stored as a sequence of events. Not just can we query these events, we can also use the event log to reconstruct past states, and as a foundation to automatically adjust the state to cope with retroactive changes.*
>
> *The fundamental idea of Event Sourcing is that of ensuring every change to the state of an application is captured in an event object, and that these event objects are themselves stored in the sequence they were applied for the same lifetime as the application state itself.*

In other words: event sourcing is about storing changes instead of their result. It's those changes that make up the final state of a project.

With event sourcing, the question of "is a cart checked out" should be answered by looking at the events that happened related to that cart and not by looking at a cart's status.

That sounds like overhead indeed, so what are the benefits of such an approach? Fowler lists three:

- **Complete Rebuild**: the application's state can be thrown away and rebuilt only by looking at events. This gives lots of flexibility when you know changes to the program flow or data structure will happen in the future — I'll give an example of this later in this post, so don't worry if it sounds a bit abstract for now.
- **Temporal Query**: you can actually query events themselves to see what happened in the past. There's not just the end result of what happened; there's also the log of events themselves.
- **Event Replay**: if you want to, you can make changes to the event log to correct mistakes and replay events from that point on to rebuild a correct application state.

There's one important thing missing in Fowler's list though: events model time extremely well. They are much closer to how we humans perceive the world than CRUD is.

Think of some process in your daily life. It could be your morning routine, it could be you doing groceries, maybe you attended a class or had a meeting at work; anything goes, as long as there were several steps involved.

Now try to explain that process in as much detail as possible. I'll take the morning routine as an example:

- I get up at 5 AM
- I brush my teeth — dental hygiene is important
- I get dressed
- I go downstairs to make a coffee
- I go to my home office (with my coffee)
- Next, I read up on e-mails
- I start writing this book

Thinking with events comes naturally to us, much more naturally than having a table containing the state of what's happening right now, as with a CRUD approach.

If you can discover a "flow of time" in something as mundane as my morning routine, what about any process for our client projects? Making bookings, sending invoices, managing inventories, you name it; "time" is very often a crucial aspect, and CRUD isn't all that good in managing it since it only shows the current state.

I'm going to give you an example of how extremely simple event sourcing can be, without the need for any framework or infrastructure, and where there's no overhead compared to CRUD. In fact, there's less.

I have a blog, and I use Google Analytics to track visitors, page views, etc. anonymously. Of course, I know Google isn't the most privacy-focused, so I was tinkering with alternatives. One day I wondered if, instead of relying on client-side tracking, I could simply rely on my server logs to determine how many pages were visited.

So, I wrote a little script that monitors my NGINX access log; it filters out traffic like bots, crawlers etc., and stores each visit as a line in a database table. Such a visit has some data associated with it: the URL, the timestamp, the user agent, etc.

And that's it.

Oh, were you expecting more? Well, I did end up writing a little more code to make my life easier. Essentially, what we have here already is event sourcing. I keep a chronological log of everything that happened in my application, and I can use SQL queries to aggregate the data, for example, to show visits per day.

Of course, with millions of visits over time, running raw SQL queries can become tedious, so I added one pattern that builds on event sourcing - projections - also known as the “read model” in CQRS.

Every time I store a visit in the table, I also dispatch it as an event. Several subscribers handle them; for example there’s a subscriber that groups visits per day which keeps track of them in a table with two columns: `day` and `count`. It’s just a few lines of code:

```
class VisitsPerDay
{
    public function __invoke(PageVisited $event): void
    {
        DB::insert(
            'INSERT INTO visits_per_day (`day`, `count`)
            VALUES (?, ?)
            ON DUPLICATE KEY UPDATE `count` = `count` + 1',
            [
                $event->date->format('Y-m-d'),
                1,
            ]
        );
    }
}
```

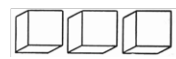
Do you want visits per month? Per URL? I’ll just make new subscribers. Here’s the kicker though: I can add them after my application is deployed and replay all previously-stored events on them. So even when I change my projectors or add new ones, I can always replay them from the point I started storing events, and not just from when I deployed a new feature.

This was especially useful in the beginning: there was lots of data coming in that were bots or traffic that weren't real users. I let the script run for a few days, observed the results, added some extra filtering, threw away all projection data and simply replayed all visits again. This way, I wouldn't need to start all over again every time I made a change to the data.

Can you guess how long it took to set up this event sourced project? Two hours, from start to a working production version. Of course, I used a framework for a DBAL and an event bus, but nothing specifically event sourcing related. Over the next few days, I did some fine-tuning and added some charts based on my projection tables etc., but the event sourcing setup was very straightforward to build.

And so, here's the point I'm trying to make: the event sourcing mindset is extremely powerful in many kinds of projects. Not just the ones that require a team of 20 developers to work on for five years. There are many problems where "time" plays a significant role, and most of those problems can be solved using a very simple form of event sourcing, without any overhead.

In fact, a CRUD approach would have cost me way more time to build this analytics project. Every time I made a change, I would have to wait a few days to ensure this change was effective with real-life visits. Event sourcing allowed me (a single developer) to be much more productive, which is the opposite of what many people believe event sourcing can achieve.

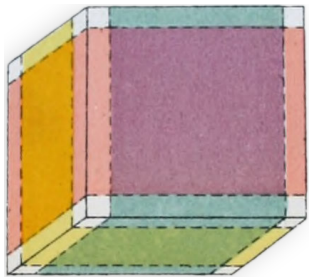


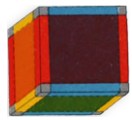
Now, don't get me wrong. I'm not saying event sourcing will simplify complex domain problems. Complex projects will take lots of time and resources, with or without event sourcing. Event sourcing simplifies some problems within those projects, and makes others a little harder.

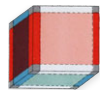
But remember, I'm not trying to make any conclusions about the total cost of using event sourcing or not. I only want people to realise that event sourcing in itself doesn't have to be extremely complex, and might indeed be a better solution for some projects, even when they are relatively small ones.

Greg Young, the one who came up with the term "event sourcing" more than ten years ago, said that if your starting point for event sourcing is a framework, you're doing it wrong. It's a state of mind first, without needing any infrastructure. I hope that more developers can see it this way, that they remove the mental layer of complexity at first and only add it back when actually needed. You can start with event sourcing today without any special framework, and it might improve your workflow.

You'll notice that this book won't start with all those complex event sourcing patterns you might have heard about before, and that might have made you take a step back from it. Instead, we're starting with the very essence of event sourcing and looking at the world from an event sourced perspective. Only when we've got those fundamental basics down, we'll start worrying about applying patterns and principles to solve complex problems.







PART 1

THE BASICS

In the first part of this book, we'll focus on all the basic building blocks required to start with event sourcing. We'll cover the mindset behind event-driven design, events and the event bus: projectors, event queries, reactors (or policies), and aggregate roots.

This part will lay the foundation that we can build upon later in this book, where we will learn about complex event sourcing patterns.

CHAPTER 1

EVENT DRIVEN DESIGN

I once had an argument with one of my colleagues about event-driven systems. I told him I didn't like working with events because they are an implicit way of programming. He understood my point of view but still told me it was worth looking into it.

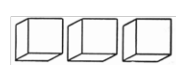
My problem with event driven systems was that whenever you'd dispatch an event, you weren't able to see right on that spot what would happen next. Take this example:

```
dispatch(new CartCheckedOut($cart->uuid));
```

Since one event can be handled by multiple listeners, you'd have to look up all consumers of that event if you wanted to understand what exactly happens whenever `CartCheckedOut` is dispatched. Maybe there's a listener that sends a mail to the customer, maybe an invoice PDF is generated, or maybe the inventory of the products added to your cart is changed. There's no way to tell what exactly will happen without knowing and inspecting all listeners for such an event.

It's a cumbersome way of programming — or so I thought.

Ironically, it's exactly the indirect nature of event-driven systems that opens up a world of possibilities. That's often the case in programming. The more flexibility required, the more complex the solution.



Let's start by exploring that one line of code in depth: "checking out a cart"; we'll start with a normal implementation and work our way towards an event-driven one, discussing the pros and cons along the way. Let's say "checking out a cart" has two side effects:

- The product inventory is updated
- An invoice PDF is sent to the customer

One way of designing the checkout flow would be to make separate functions or classes for each step of that process and bring them together in a class dedicated to "checking out a cart". I like to start from the inside out in such cases and work my way up.

For example, we could make one class that's responsible for updating the inventory:

```
class UpdateInventory
{
    public function __invoke(Cart $cart): void
    {
        foreach ($cart->items as $item) {
            $item->productinventory->subtract($line->amount);
        }
    }
}
```

And another class that's responsible for sending the invoice to the customer:

```
class SendInvoice
{
    public function __construct(
        private GeneratePdf $generatePdf,
    ) {
    }

    public function __invoke(Cart $cart): void
    {
        $pdf = ($this->generatePdf)(
            new InvoiceTemplate($cart)
        );

        $cart->customer->notify(
            new InvoiceForCustomer($cart, $pdf)
        );
    }
}
```

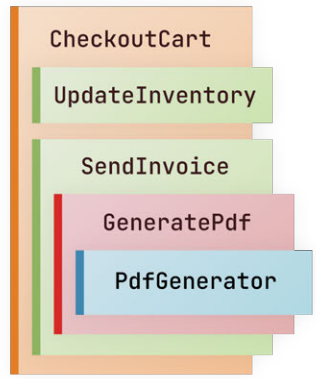
Finally, we can bundle them together in a class that's responsible for "checking out the cart":

```
class CheckoutCart
{
    public function __construct(
        private UpdateInventory $updateInventory,
        private SendInvoice $sendInvoice,
    ) {
    }

    public function __invoke(Cart $cart): void
    {
        $cart->update([
            'status' => CartStatus::CHECKED_OUT,
            'checked_out_at' => now(),
        ]);

        ($this->updateInventory)($cart);

        ($this->sendInvoice)($cart);
    }
}
```

I prefer to split this functionality into separate classes and inject them using Laravel's autowiring because an example like this one tends to be larger and more complex in reality. If we keep all code in one class, that class will grow larger and larger, making it more difficult to test, but also more prone to bugs if we'd need to change something in it.

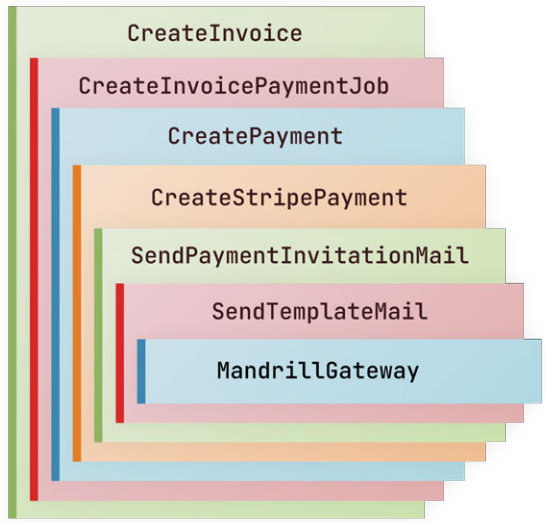
I described this concept before in Laravel Beyond CRUD and called them **actions**: classes that do something business-related.

There's a problem with actions, though: just like putting all code in a single class, actions composed out of other actions can grow quite large as well, but without anyone knowing. If we're not careful, there's the possibility of ending up with a deep dependency chain that's very hard to maintain. Even in our current example, we're already pushing the limits. Let's visualise the dependency tree:



I didn't show the code of the `GeneratePdf` action, but you can imagine how it uses an external PDF generator library to convert, for example, an HTML template to a PDF.

Even with this simplified example, we're already dealing with a four level-deep dependency chain. Now let me show you a real-life example: this is a (slightly simplified) dependency tree of a class in a project I worked on in the past:



The goal of all this code is to store an invoice in the database, asynchronously create a payment via an external payment provider, and send a custom mail with the payment link to the customer via Mandrill.

You would never write code that's nested seven levels deep, but because classes are injected in this example, their nested dependencies are hidden from sight unless you start looking for them. On top of that, these actions are very much hard-wired together. Even though the functionality is split between classes, it's still one blob of tightly knit code, looking at it from the outside.

Seeing a dependency graph like this made me think about the limitations of what I considered the "simpler approach". Suddenly the indirectness of event-driven systems offered a lot of possibilities. If we don't need to worry about micro-managing the next steps in the program flow, we'd have lots more flexibility, and we'd also write more sustainable code.

So let's rethink the design: what if "checking out a cart" is an event whereas many pieces as we'd want could hook into it? Instead of micromanaging all the steps that come with our invoice flow, we can register our "actions" as event handlers instead.

With a little refactoring, our `UpdateInventory` action looks like this:

```
namespace App\Listeners;

// …

class UpdateInventory
{
    public function handle(CartCheckedOut $event): void
    {
        foreach ($event->cart->items as $item) {
            $product = $item->product;

            $product->inventory->subtract($line->amount);
        }
    }
}
```

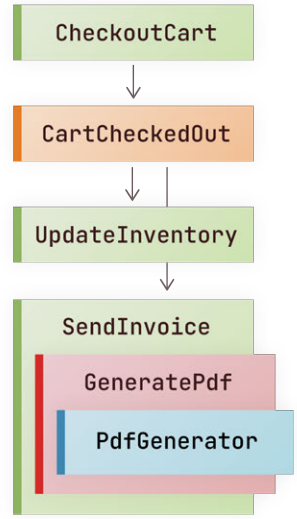
And this is the `SendInvoice` action, now living in the `App\Listeners` namespace:

```
namespace App\Listeners;

// …

class SendInvoice
{
    public function __construct(
        private GeneratePdf $generatePdf,
    ) {
    }

    public function handle(CartCheckedOut $event): void
    {
        $pdf = ($this->generatePdf)(
            new InvoiceTemplate($event->cart)
        );

        $cart->customer->notify(
            new InvoiceForCustomer($event->cart, $pdf)
        );
    }
}
```

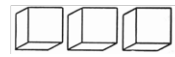
It might seem like we've only added code, but keep in mind that our `CheckoutCart` action can now be written like so:

```
class CheckoutCart
{
    public function __invoke(Cart $cart): void
    {
        $cart->update([
            'status' => CartStatus::CHECKED_OUT,
            'checked_out_at' => now(),
        ]);

        dispatch(new CartCheckedOut($cart));
    }
}
```

But besides moving code around, what have we gained? Well, let's look at our schema once again:



We don’t need to worry about the specific side effects in any given part of our code, and that gives us lots of freedom. What we’ve discussed in this chapter is the foundation for many principles to come. An event-driven architecture will give us freedom and flexibility. It will make what seems like complex problems trivial to solve, it will allow us to naturally scale our project and more. All of this built upon the simple principle of events.



Before we discuss all those possibilities, we need to talk about the two key aspects of our event-driven system: the event bus and events themselves. Those are the topics for the following two chapters.

CHAPTER 2

THE EVENT BUS

Before learning about everything that's possible with an event-driven system, we need to understand the two most basic concepts that underpin it. We'll start with the event bus.

Think of the event bus as the spine of any event-driven system. It's the core that knows about all listeners and which events they handle.

In its simplest form, an event bus is a class with just a few lines of code: it keeps track of which listeners are registered for which events and will notify them every time an event is dispatched. It's really nothing more than this:

```
class EventBus
{
    private array $listeners = [];

    public function addListener(
        string $eventName,
        callable $listener
    ): self {
        $this->listeners[$eventName][] = $listener;
    }

    public function dispatch(object $event): void
    {
        $listenersForEvent =
            $this->listeners[$event::class] ?? [];

        foreach ($listenersForEvent as $listener) {
            $listener($event);
        }
    }
}
```

This implementation is too simple to be a production-ready event bus. A real event bus would probably:

- be able to handle events asynchronously,
- have middleware support so that developers can hook into event dispatching, and
- likely know how to deal with other kinds of listeners besides simple callable objects etc.

While a proper event bus is an interesting piece of infrastructural software in itself, it won't be the focus of this book. Every event sourcing package or framework will already provide an event bus that has most of the features described above. In this book, we're more interested in what we can do *with* an event bus.

So starting from now, we'll assume that the event bus is a given and we don't need to worry about it any longer. If you want to do some code diving to learn more, I'd recommend starting with the `league/event` package. It's a proper event bus implementation from which you can learn quite a lot.

TESTING

Since we'll assume the event bus is always a part of the framework, we won't discuss testing its internal workings.

Throughout this book, we'll see numerous examples of events and listeners, and testing whether they are linked correctly is an important step in developing our application.

Testing listeners in isolation is doable by mocking the event bus itself and observing what listeners were triggered, but we should consider whether such tests add much value. Later in this book, we'll discover how we balance unit and integration testing, and the latter ones will already test our listener configuration.

We shouldn't repeat tests for the same thing, so let's hold off on mocking the event bus for now and see where integration tests get us in later chapters.

CHAPTER 3

EVENTS

In contrast to the event bus, events themselves will have much of our focus throughout this book. It's crucial to understand what an event is, so we'll spend this whole chapter dedicated to it.

In essence, an event is nothing more than a statement of "something happened in the system". It's a message that's broadcasted for anyone who's interested in hearing it. This means that "an event" isn't by definition only bound to our own program: complex systems might need to broadcast events to the outside world, because other systems listen to them.

But we're getting ahead of ourselves; first we need to agree on the details that make an event. In some cases, the event *itself* is enough for listeners to handle it. In other words: an event could be an "empty message". The fact that the message is sent is enough. Martin Fowler gives these kinds of events the fitting name of **Event Notification**: events that are only used as a *ping* for their listeners.

Such an event would be as simple as this:

```
class CartCheckedOut {}
```

You'll immediately notice that it's difficult to come up with a good use case for such events: a `CartCheckedOut` event without knowing *which* cart was checked out is useless. However, you could design systems that would listen to those notifications and in turn look in the database or somewhere else for what exactly changed.

Nevertheless, you probably have the same reservations as I do: events without any context information don't seem that useful.

That's why most events carry some kind of state with them. If events are messages, then their data represent the message's body. Martin Fowler calls this principle **Event-Carried State Transfer**: the event contains some information to make it more useful. In the case of our `CartCheckedOut` event, we could add a cart ID to indicate which cart we're talking about:

```
class CartCheckedOut
{
    public function __construct(
        public int $cartId,
    ) {
    }
}
```

Note that I prefer to use public properties in events. It's been a long standing convention in our projects that event data is read only, and within our team we're fine keeping that rule without programmatically enforcing it. If you don't feel confident with such a design though, you're free to add getters and make your event properties private instead.

Event-carried state transfer poses the question though: what information do we send with events, and more importantly, what information do we not? Couldn't we send the cart model as a whole, instead of only its ID?

Let's look at an example that would make the problem with such an approach clearer. Imagine an `InvoiceGenerated` event that's triggered when the invoice PDF generator has finished its job. Would you send the PDF file itself via the event, or rather its path? If we're using events only as an in-memory mechanism to send messages in our own codebase,

then the answer would be “probably both”. But what if we’d send our messages across to other services, or maybe store a serialized version of events in a database? Then we wouldn’t be able to send the PDF itself via the event as it would lead to significant memory issues.

These kind of problems might seem like an edge case but there’s one way of using events that I didn’t mention yet, and where the question of how to serialize events becomes very important: **event sourcing**.

With event sourcing, we mean that the messages in our system aren’t just broadcasted and forgotten about. Instead, every event is stored in some kind of database or somewhere else on disk. The first advantage of doing so is that we get a log of all events that happened in the system. More importantly though, event sourcing doesn’t use events as a mechanism to handle secondary functionality as we often see in Laravel; instead events themselves become the central mechanism of our system.

Take a look at our previous `CheckoutCart` action:

```
class CheckoutCart
{
    public function __invoke(Cart $cart): void
    {
        $cart->update([
            'status' => CartStatus::CHECKED_OUT,
            'checked_out_at' => now(),
        ]);

        dispatch(new CartCheckedOut($cart));
    }
}
```

Here we dispatch the `CartCheckedOut` event only to handle side effects of the cart being checked out (updating inventory and sending an invoice PDF to the customer). What if we turned things around and "updating the cart's status" is also considered part of those "side effects"?

Our `CheckoutCart` action would look like this:

```
class CheckoutCart
{
    public function __invoke(Cart $cart): void
    {
        dispatch(new CartCheckedOut(
            cartId: $cart,
            checkoutDate: now(),
        ));
    }
}
```

The handler dedicated to updating the cart would look like this:

```
class CheckoutCartProjector
{
    public function handle(CartCheckedOut $event): void
    {
        $cart = Cart::find($event->cartId);

        $cart->update([
            'status' => CartStatus::CHECKED_OUT,
            'checked_out_at' => $event->checkoutDate,
        ]);
    }
}
```

Now, I realise there are a few things happening here that might seem strange. What's that `CheckoutCartProjector` about? Why is the checkout date suddenly passed into the event? We'll cover those questions, and more, soon. First it's important to understand the second crucial part of event sourcing. If events are stored in the database, and if everything in our application is a side effect of those events, what would happen if we threw away all our data except for the events, and *replayed* all events? Dispatch them again, to trigger all side effects once more.

Imagine if there also was a `CartCreated` event and a `CartItemAdded` event, stored in the database with all relevant data. We could throw away our cart, and rebuild it based on the information saved in the stored events. Such a collection of stored events are called an **event stream**: it's a chronological list of events that belong together, for example the events that make up one cart.

Compare this approach to traditional CRUD applications. The most important difference is that every CRUD action will overwrite existing data. Event sourcing only adds new data, by recording new events on top of the old ones. You never lose information about what happened in the past. If you want to achieve the same result in CRUD applications, you'd have to explicitly provide logging or history infrastructure to keep track of what happened, with event sourcing this is a given from the start.

It might seem like there's little to gain from having such a built-in log, but we'll discover several powerful patterns that build on this simple principle throughout this book. I'm sure you'll start to appreciate the power of event sourcing as we progress.

There are some side notes to be made. With event sourcing, there are lots of intricacies to deal with: surely we wouldn't want to send an invoice PDF a second time to the customer when we rebuild our application? How do we keep track of incremental IDs if everything is thrown away? How can we make events serializable so that they can be stored in a database?

There's a lot of ground to cover when it comes to event sourcing and we'll need the better part of this book to manage it all, but it's already important to understand the benefits it brings:

- stored events are a log of everything that happened in your system;
- events can be replayed, meaning you can rebuild old state, or build new state based on old events *after* the fact; and
- modelling with events is a natural way of "thinking about the world" — we'll explore this topic in the next chapter.

TESTING

As long as we respect the strict boundaries that make an event — a simple data class without any logic — there's no reason to write explicit tests for them. That is: only when you use strictly typed properties.

I'm a big proponent of using PHP's type system to its full extent and, as well as that, use whatever's available in the static analysis world to reduce the amount of tests needed. Diving deeply into the topic of static analysis is outside the scope of this book, but if you want to read more about it you can read my book "Front Line PHP".

CHAPTER 4

MODELLING THE WORLD

In traditional CRUD applications, we usually think in terms of "entities" or "models". For example, when our client tells us they want us to build a shopping cart, most of us instinctively think about what models would be related to that feature. There's the `Cart`, there probably will be a `CartItem`, a `Customer` and `Product`. We start with the data schema and work our way from there towards the functionality.

When you think about it, this is actually a strange way of modelling what happens in the real world as code. Humans don't only think in terms of nouns like `Cart` and `Product`. Verbs are equally important as well. If you'd ask someone to describe their real-life shopping process, it would sound more like this:

"I go to the shop, I take a cart, I walk around the aisles, I fill my cart, I make my way towards the cash register, and I pay the cashier."

We think in processes, with nouns and verbs, not entities on their own. It turns out it becomes natural to design our code in ways so that it more closely resembles this way of thinking. And — who could have guessed it — events are a much better starting point than models or entities to do this.

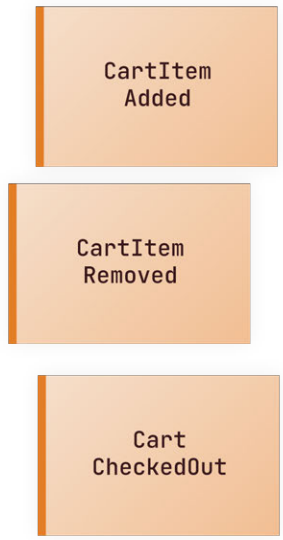
There's `CartCreated`, `CartItemAdded` and `CartCheckedOut`; these events tell so much more about what the system can actually do, compared to just models.

Combine this idea with event sourcing, where the event itself is saved, and everything else is considered a side-effect. That means that models like `Cart` and `CartItem` are side effects — results — of the events that happened in the system. They are *one* interpretation of the event stream. What we'd call "models" in a traditional Laravel application are no longer the source of truth - events are. Models merely give you a window to look into the event stream. That's why from here on out, we'll not use the term "model" anymore but call them **projections** instead.

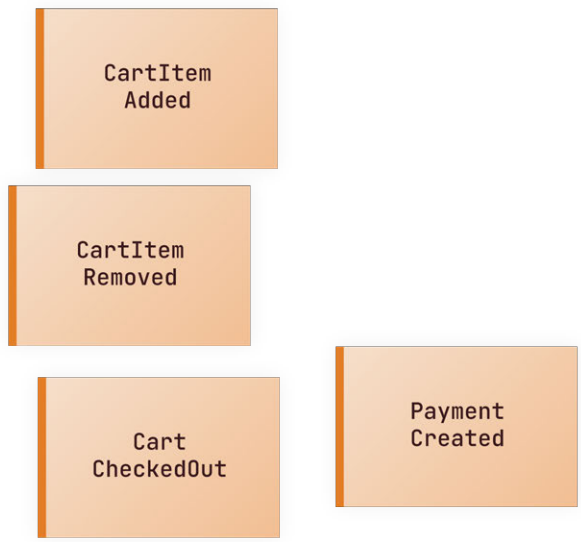
We'll look at projections in depth in the next chapter because first, I want to shift our focus back to the events. Projections are only a side effect of those events, and events are a more accurate way of representing the world (or a small part of it). That means that our starting point should be by using events and worry about all their side effects later.

When a client asks you to build a shopping cart, you'll probably make some kind of analysis before actually writing code. These could be user stories, UML diagrams, whiteboard sketches, etc. I want to put all of those aside for now and only focus on the most important building block: events. We're going to design the cart system together, and we're going to make a diagram of the process flow first.

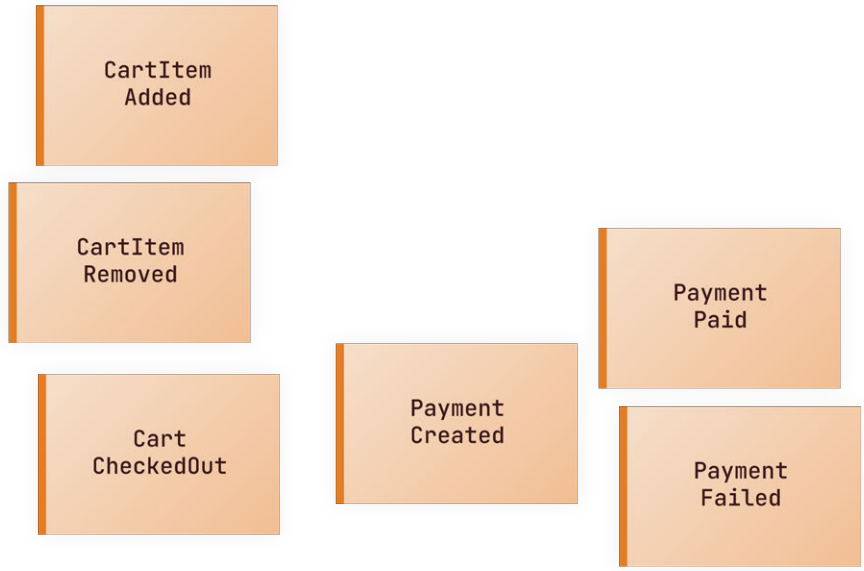
“A shopping cart”, that’s what we’re going to make. Imagine we have a whiteboard in front of us and a bunch of sticky notes that we can use to visualize the process flow. What events could happen with our cart? We could add items to it, which we could also remove again, and we could check it out. Let’s visualize those events.



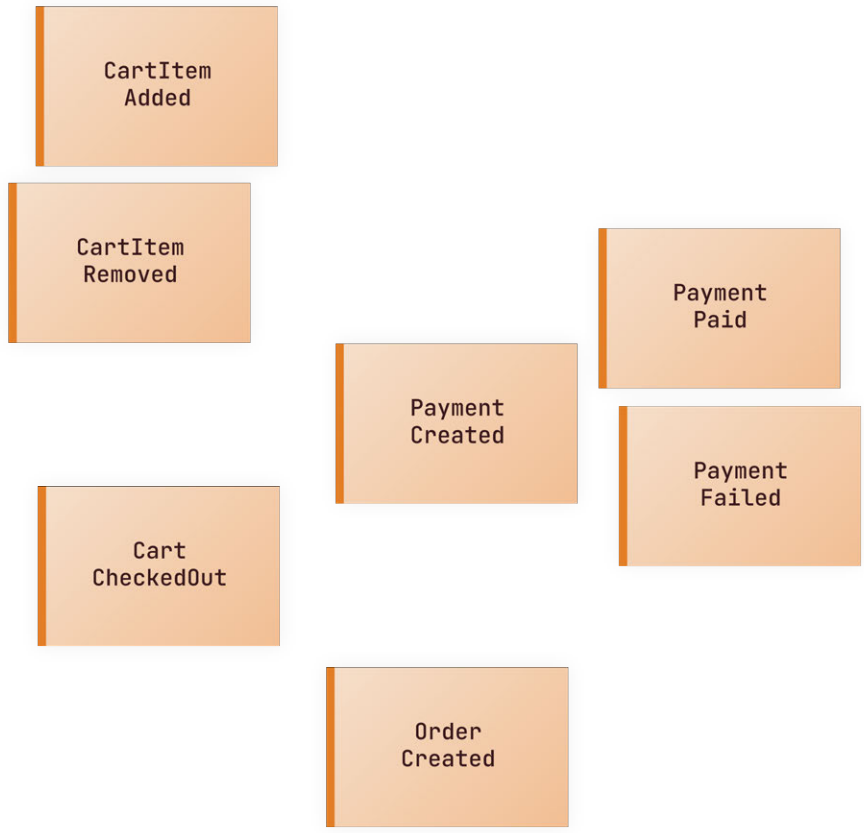
Of course, “checking out a cart” means we’ll also have to make a payment. Let’s add a `PaymentCreated` event as well.



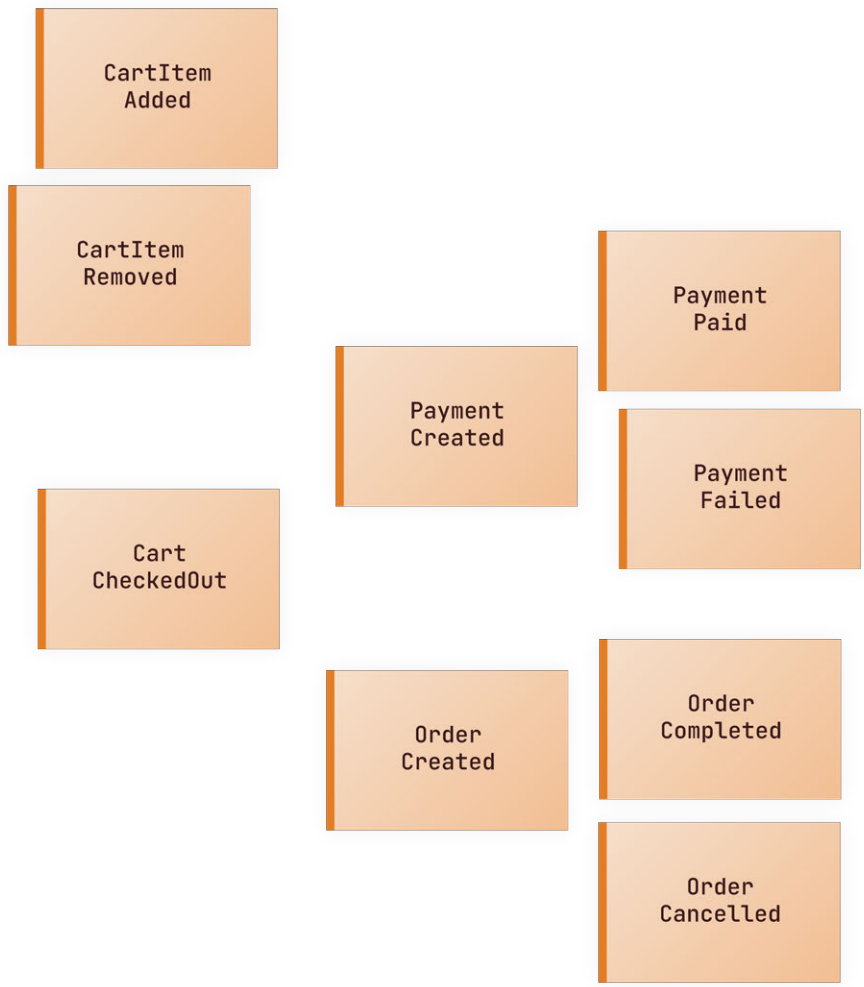
It’s safe to say that payments might not always succeed, so we’ll probably need to make a distinction between a payment being paid, and a payment failing.



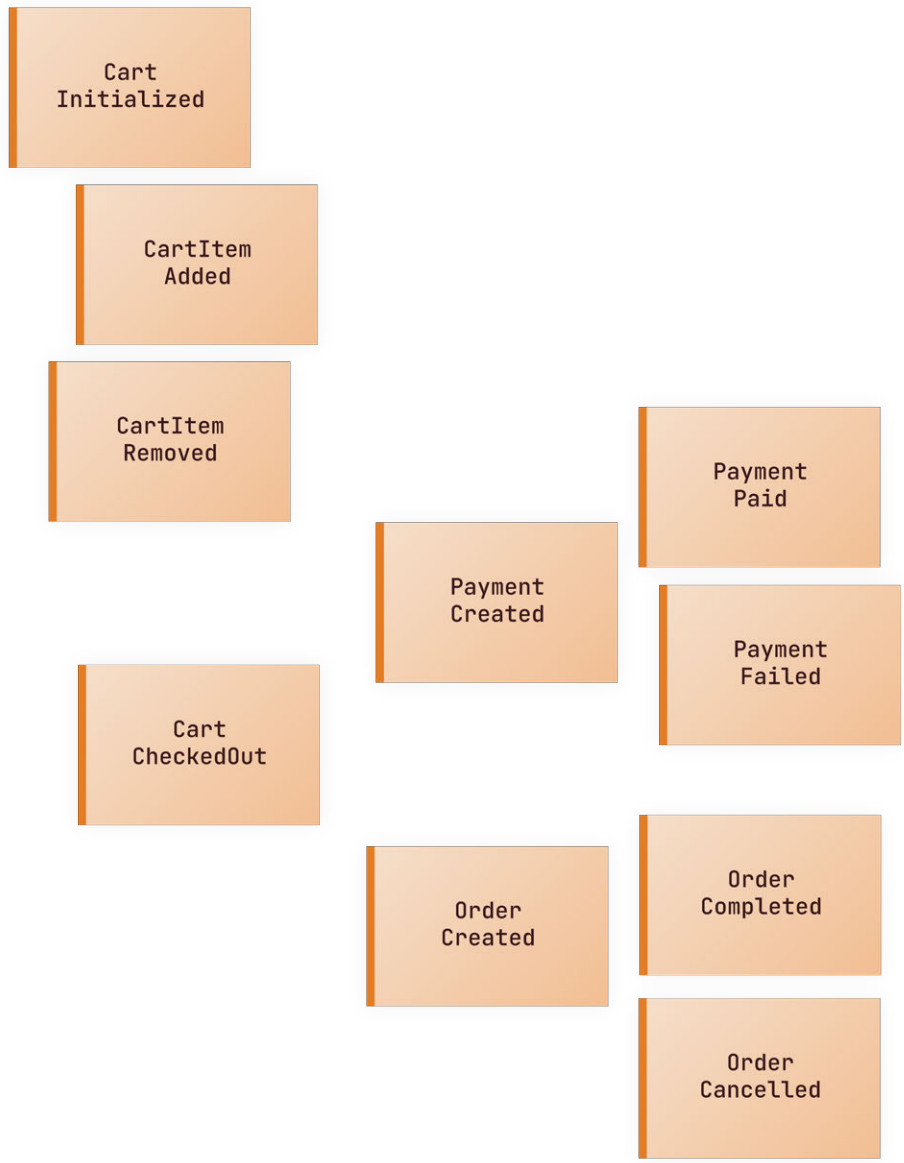
What’s still missing is some kind of end result. When a cart is checked out, it usually results in some kind of order.



It also makes sense that an order can be completed, meaning the client received their products and nothing more should be done. If an order can be completed, it makes sense that it can also be cancelled. For example: if the payment failed, the order shouldn't continue to be processed.



Finally, and this is a more technical one: there needs to be some kind of "starting point", before being able to add items to a cart. A customer will actually need an existing cart before adding items to it. In real life, we'd call this action "taking a cart", but that doesn't really translate well to an online shop. So let's call this event `CartInitialized`; it will probably happen behind the scenes when a user's session starts, but it'll be there nevertheless.



What we've just done with our sticky notes is called **event storming** — or least, the first step of it. It's a brainstorming technique that aims to clarify the process flow using events. Our diagram is still missing things, but starting out with events is the way to go: they are the core building blocks of our application, so it makes sense to begin with them.

Event storming has a few advantages. Normally you'd do event storming sessions with your client in the same room. It's an easy way for non-technical people to describe their expectations while making a diagram that closely matches the actual implementation in code. Furthermore, it can be done completely without any computers or software; you just need a bunch of colored sticky notes, a pen and a wall to stick them to.

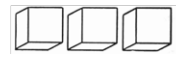
Our event storming session, however, isn't done yet. There's lots of information missing still. We'll continue to work on this diagram throughout this book. But first, we'll implement the events themselves and figure out a way to store them.

CHAPTER 5

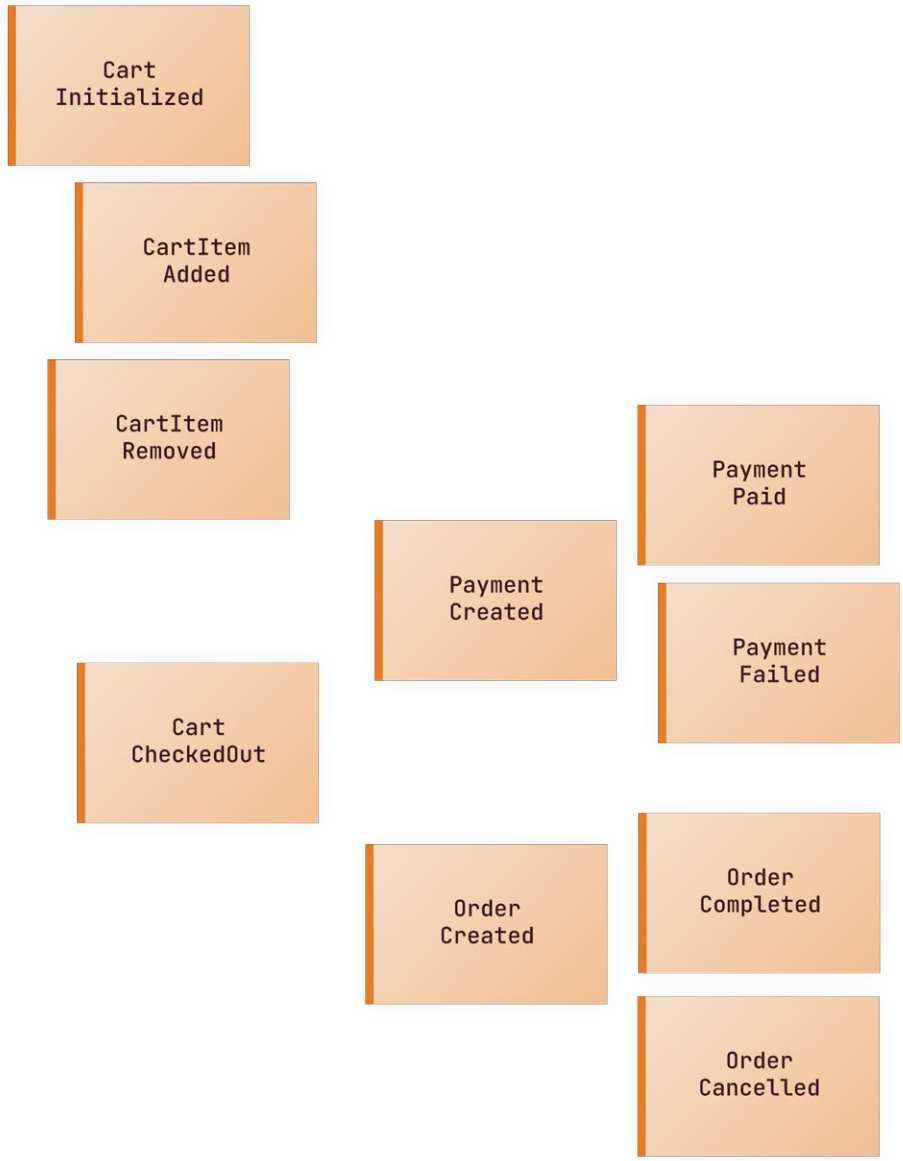
STORING AND PROJECTING EVENTS

When we discussed the event bus back in chapter 3, I described how it can support functionality like asynchronous events, middleware, etc. Something that I didn't mention yet is that the event bus is also the entry point for storing and replaying events in an event-sourced application. In this chapter, we'll take the first steps towards working with stored events.

From here on out, our code samples will be written in a Laravel application, using a package dedicated to event sourcing: `spatie/laravel-event-sourcing`. This package isn't needed for you to follow along; it only gets rid of boring boilerplate code that we'd otherwise need to write ourselves, like an event bus as described above. This package allows us to focus on the part where we can design an application based on events, instead of having to deal with all kinds of infrastructure code.



We ended the previous chapter with a diagram that looked like this:



Before we translate this diagram to code, it's best to discuss how we're going to group code together. Everything in our diagram is related to carts, but there will be much more code added on top of those events. It's good to come up with some sort of categorization, to keep our individual parts maintainable. I like to draw some inspiration from the DDD community and make so-called **aggregates**: an aggregate is a group of code that can be considered "a whole".

A great source on the topic of aggregates is Eric Evans' book called "Domain-Driven Design", there's also Martin Fowler who describes them like so:

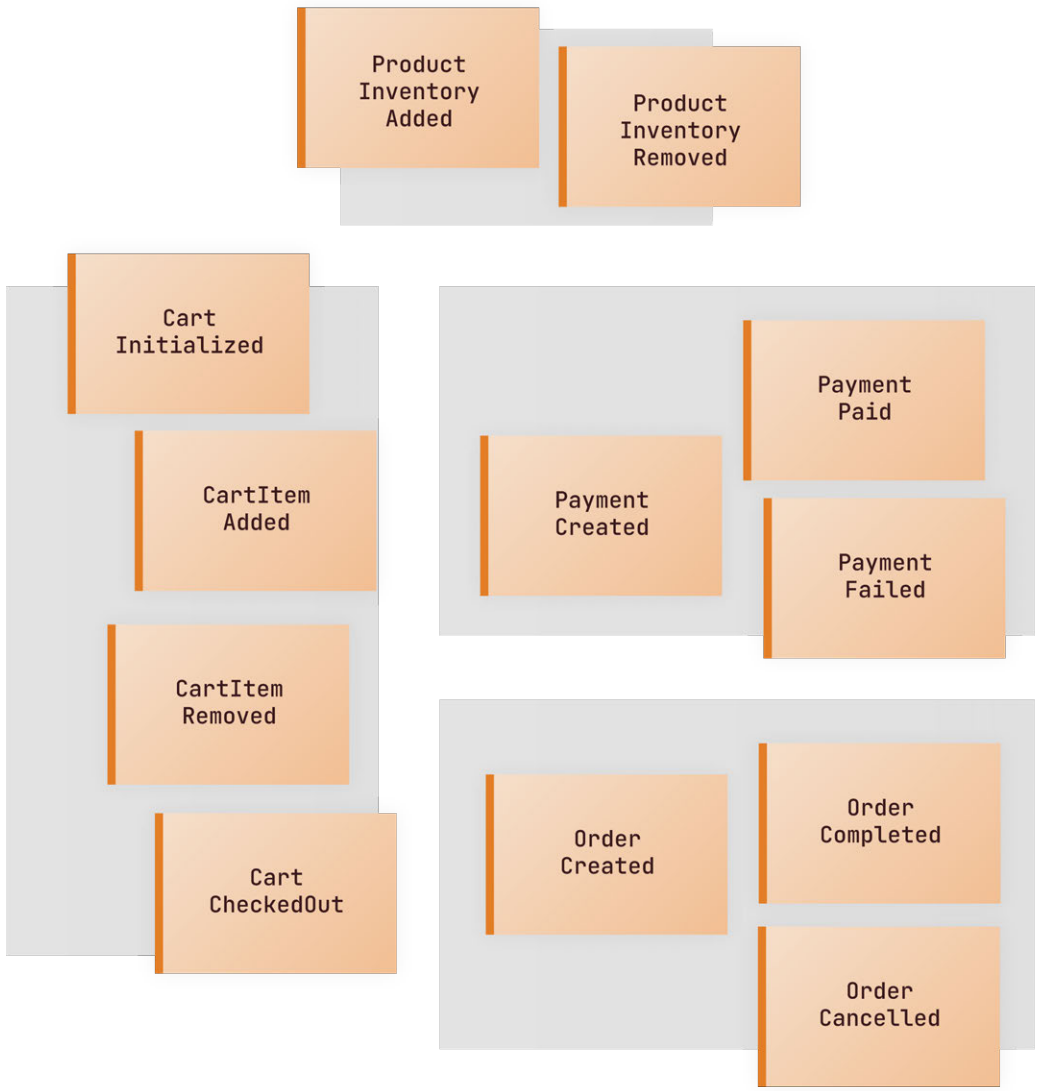
> *A DDD aggregate is a cluster of domain objects that can be treated as a single unit. An example may be an order and its line-items; these will be separate objects, but it's useful to treat the order (together with its line items) as a single aggregate.*

In other words: an aggregate represents a group of concepts that belong together, like orders and their lines. It doesn't make sense to work with order lines outside the context of orders, which is why an order and its lines can be considered an aggregate as a whole.

With that in mind, we're able to identify four aggregates in our diagram:

- the cart, responsible for customers adding and removing items to and from their cart, up until the point of checking it out;
- the order, responsible for the flow that happens after the cart is checked out;
- a product's inventory, which is a secondary concern when customers are adding items to their cart; and
- the payment, which deserves its own process flow, given all the state changes that usually come along with it.

Reflecting these changes on the diagram, we end up with something like this:



For now, we're only using aggregates as a way of grouping code together, but we're going to take it a few steps further in upcoming chapters. We're going to assign an identity to those aggregates so that all events will be related to a specific aggregate instance. Think of it like you'd assign an ID to a `Cart` Eloquent model. Instead of such an ID identifying only one database row, it'll be used as the identity of the group, the aggregate, as a whole.

Likewise, whenever an event is dispatched, it'll originate from one of these aggregates. It'll be crucial to be able to trace back an event to its point of origin. For example: "this `CartItemAdded` event is associated with cart number 1". In other words, this event originated from the `Cart` aggregate, and specifically the one with ID `1`.

Speaking of those IDs, there's an important note to make about them. We established that events will be the source of truth in our application, which means that we should never rely on data that's unknown to the event. IDs are the perfect example: if we're going to initialize a `Cart`, it would normally be assigned an ID given by the database using an auto-incrementing key. However, if we throw away our projections to rebuild them from the event stream, we'd lose the original ID and the database would generate a new one. Imagine there were other resources linking to that cart with its ID, which suddenly changed because we decided to rebuild all carts. This would result in numerous amounts of problems and bugs.

Keep in mind our one crucial rule when it comes to event sourcing: **events are the only source of truth**. If data isn't present on an event, it can't be trusted. In practice, this means that we'll use UUIDs instead of IDs, because they are much simpler to generate from the outside — you'll see this in practice soon.

Next, as we saw in chapter 4, we should carefully think about what data we send with events. Remember, we're going to save these messages in a database, so it's crucial they can be properly serialized. It's best not to store complex objects as a whole but to convert them to an easily storable format.

With all of that being said, let's look at what our first event would look like, the `CartInitialized` event:

```php
<?php

namespace App\Cart\Events;

use Carbon\Carbon;
use Spatie\EventSourcing\StoredEvents\ShouldBeStored;

class CartInitialized extends ShouldBeStored
{
    public function __construct(
        public string $cartUuid,
        public string $customerUuid,
        public Carbon $date,
    ) {
    }
}
```

Notice the `ShouldBeStored` class we're extending from, provided by the event sourcing package. It adds functionality for serializing and deserializing stored events. The rest is fairly simple: it's a class containing some data which is relevant to "when a cart is initialized". One of its properties is the cart's UUID; as we saw earlier, it's important to store it in the event to be able to rebuild our application state when needed.

Let's take a look at another event, the `CartItemAdded` event:

```
namespace App\Cart\Events;

use Spatie\EventSourcing\StoredEvents\ShouldBeStored;
use App\Product\Price;

class CartItemAdded extends ShouldBeStored
{
    public function __construct(
        public string $cartUuid,
        public string $cartItemUuid,
        public string $productUuid,
        public int $amount,
        public Price $currentPrice,
    ) {
    }
}
```

Besides keeping track of the cart's UUID and its own UUID, note that we're also storing the current price and not just a reference to the product itself. Imagine you're adding an item to a cart that has a discounted price, which is changed by an admin right after you've added it. You still want the old price to be used at checkout. At least: that's the business rule I've established for this example. It might be that there are even more complex business rules in real life, which is why the event storming sessions are so important. By talking about the application flow with your client, these kinds of rules are being discussed and become clear before actually writing code.

Next, we need a way for our events to be dispatched and stored. Remember our action classes? That's where we'll need to make some changes. For example, here's the action to initialize a cart:

```
class InitializeCart
{
    public function __invoke(Customer $customer): Cart
    {
        $cartUuid = uuid();

        $event = new CartInitialized(
            cartUuid: $cartUuid,
            customerUuid: $customer->uuid,
            date: now(),
        );

        $event->handle();

        return Cart::find($cartUuid);
    }
}
```

There are a few interesting things going on, so let's go through them step by step.

Dispatching the event

We're calling `$event->handle()`, a method that's provided on the `StoredEvent` class by the package. This is not the best way to dispatch and store events, and we're only doing this temporarily. In later chapters, we'll discover better ways to handle the event flow, and we'll refactor this example accordingly.

Returning a cart

Wherever we use this `InitializeCart` action, chances are we'll want to do something with the cart after it's initialized. Imagine this action is used in a controller, and we'd like to send the cart UUID and status back to the customer.

Remember that `Cart` is a projection, a side effect of the event (we'll take a look at it in depth in a minute). Being a projection means that we're only able to immediately return a `Cart` from this action if we *know* that events are handled immediately when dispatched. As we'll see later in this book, event-driven design is a suitable technique to build asynchronous systems or distributed systems that communicate with each other in an asynchronous way. In those cases, we won't be able to ensure that the projection exists immediately after the event was dispatched.

For now, we'll assume that we're working in a synchronous system and that all events are handled immediately. This allows us to return a `Cart` directly from our action.

What we haven't discussed yet is *how* such a projection is actually made after an event was dispatched.

A projection is a database table (or any other way of storing data) that represents one way of looking at the event stream. It doesn't have to be the only way; it's just one interpretation of the events that happened. In this case we'll call the projection the `Cart`. Since we're going to store projections in a database, they will be normal Eloquent models as you'd know them in a typical Laravel application. Normal, but with one difference: projections are based on the event stream, and we're not allowed to make changes to them from outside that stream. Remember: events are the only source of truth! In other words: every change to our projections should be a side effect of events.

To prevent us from forgetting this rule, the event sourcing package introduces a new class called `Projection`. This class extends from Laravel's Eloquent models as we know them, with the exception that it prevents you from saving data directly to them — I'll show you how they are used in a moment.

Let's start with what the cart projection itself would look like:

```php
namespace App\Cart\Projections;

use Illuminate\Database\Eloquent\Relations\HasMany;
use Illuminate\Database\Eloquent\Relations\HasOne;
use Spatie\EventSourcing\Projections\Projection;
use App\Cart\States\CartState;

/**
 * @property string $uuid
 * @property int $total_price_excluding_vat
 * @property int $total_price_including_vat
 * @property CartState $state
 * @property-read \App\Shop\Customer\Customer $customer
 * @property-read \App\Shop\Cart\Projections\CartItem[] $items
 */
class Cart extends Projection
{
    protected $casts = [
        'total_price_excluding_vat' => 'integer',
        'total_price_including_vat' => 'integer',
        'state' => CartState::class,
    ];

    public function customer(): BelongsTo
    {
        return $this->belongsTo(Customer::class);
    }

    public function items(): HasMany
    {
        return $this->hasMany(CartItem::class);
    }
}
```

As you can see, a normal Eloquent model. Note that I've also added the property doc blocks for convenience. If you're unsure about that `CartState` class, you can read about it in the documentation of `spatie/laravel-model-states` or in Laravel Beyond CRUD; in short it's a class that encapsulates all state-related logic for a `Cart`.

The migration of our projection looks like this:

```
Schema::create('carts', function (Blueprint $table) {
    $table->uuid('uuid');

    $table->uuid('customer_uuid');
    $table->foreign('customer_uuid')
        ->references('uuid')
        ->on('customers');

    $table->string('state');

    $table->integer('total_price_excluding_vat')->default(0);
    $table->integer('total_price_including_vat')->default(0);

    $table->timestamps();
});
```

Now that we have a place to store data, we'll need a place to hook into the events and actually save the projection. The classes responsible for listening to events and projecting data into the database are called **projectors**:

```
namespace App\Cart\Projectors;

use Spatie\EventSourcing\EventHandlers\Projectors\Projector;
use App\Cart\Events\CartInitialized;
use App\Cart\Projections\Cart;
use App\Cart\States\PendingCartState;

class CartProjector extends Projector
{
    public function onCartInitialized(CartInitialized $event): void
    {
        (new Cart)->writeable()->create([
            'uuid' => $event->cartUuid,
            'state' => PendingCartState::class,
            'customer_uuid' => $event->customerUuid,
            'created_at' => $event->date,
        ]);
    }
}
```

Depending on how the event sourcing package is configured, it can automatically discover projectors and hook them up as listeners to events. In this case you can assume that `CartProjector` is automatically registered as a listener class, and that `onCartInitialized` will be called whenever a `CartInitialized` event is dispatched.

Let's note a few things.

Writable projections

Like I said before, projections by default can't be written to as a safety mechanism. The only place you should make changes to projections is within projectors. This is why you need to explicitly call `writeable()` on the cart; otherwise you'd get an error when trying to save it.

Storing data

Also, note that we're not generating any new data like IDs or UUIDs in projections: the event is the source of truth, and it contains the UUID for this cart. The same goes for `created_at`: we're not generating a timestamp when creating the projection. The cart was created the moment the event was dispatched, not when one of its side effects were handled.

Imagine if we threw away this `Cart` projection and rebuilt it two weeks after the cart was actually initialized. We'd still want its `created_at` date to be whenever the event was originally dispatched, not when the projection was rebuilt.

Rebuilding projections

I've already mentioned "rebuilding projections" several times, but I haven't explained how it works yet. We'll discuss this in the next chapter.

Before ending this chapter, let's look at one more projection: the one for adding a cart item. Every time a cart item is added, we should add a row in the `cart_items` table and link it to the cart, but we should also update the cart's total price. We could calculate this kind of information on the fly whenever we're working with a cart, but I'd say it's better to store it in the database instead. Having this kind of information allows us to query it directly in the database and has very little impact on our code.

In essence, we could say there are *two* side effects when adding an item to a cart: the cart item itself should be added, and the cart's price should be updated.

Since `CartItem` is another projection, we'll add a dedicated `CartItemProjector` for it. This projector doesn't do much besides storing the data from the `CartItemAdded` event as a projection in the database:

```
namespace Spatie\Shop\Cart\Projectors;

use Spatie\EventSourcing\EventHandlers\Projectors\Projector;
use Spatie\Shop\Cart\Events\CartItemAdded;
use Spatie\Shop\Cart\Projections\Cart;
use Spatie\Shop\Cart\Projections\CartItem;

class CartItemProjector extends Projector
{
    public function onCartItemAdded(CartItemAdded $event): void
    {
        (new CartItem)->writeable()->create([
            'uuid' => $event->cartItemUuid,
            'cart_uuid' => $event->cartUuid,
            'product_type' => $event->productType,
            'product_id' => $event->productId,
            'amount' => $event->amount,
            'price_per_item_excluding_vat' =>
                $event->currentPrice->pricePerItemExcludingVat(),
            'price_per_item_including_vat' =>
                $event->currentPrice->pricePerItemIncludingVat(),
            'vat_percentage' =>
                $event->currentPrice->vatPercentage(),
            'vat_price' =>
                $event->currentPrice->vatPrice(),
            'total_price_excluding_vat' =>
                $event->getTotalPriceExcludingVat(),
            'total_price_including_vat' =>
                $event->getTotalPriceIncludingVat(),
        ]);
    }
}
```

Updates to the `Cart` should be handled by our `CartProjector`:

```
class CartProjector extends Projector
{
    // …

    public function onCartItemAdded(CartItemAdded $event): void
    {
        $cart = Cart::find($event->cartUuid);

        $totalPriceExcludingVat = $cart->total_price_excluding_vat
            + $event->getTotalPriceExcludingVat();

        $totalPriceIncludingVat = $cart->total_price_including_vat
            + $event->getTotalPriceIncludingVat();

        $cart->writeable()->update([
            'total_price_excluding_vat' => $totalPriceExcludingVat,
            'total_price_including_vat' => $totalPriceIncludingVat,
        ]);
    }
}
```

Here we can see a glimpse of the power of the indirect nature of event-driven architectures. We didn't need one large class that knows about all the steps there are to adding a cart item. Instead, we end up with several small and isolated listeners that handle their specific piece of the program flow.

The more you work with event sourcing, the more you'll appreciate this behaviour. It's an extremely flexible way to model complex flows while at the same time keeping individual pieces of the puzzle small and concise.

TESTING

I've already mentioned before that you should aim to only store simple values in events. Not only to make serializing them easier but also to avoid having to test several getters or static constructors.

There are, of course, cases where those methods really help out, but remember to keep them at a minimum. If you're adding any functionality to your event classes, make sure you write a test for it. It's usually only a few lines of code:

```
class CartItemAdded extends ShouldBeStored
{
    // …

    public function getProduct(): Product
    {
        return Product::find($this->productUuid);
    }
}
```

```php
/** @test */
public function test_get_product(): void
{
    $product = ProductFactory::new()->create();

    $event = new CartItemAdded(
        /* … */,
        productUuid: $product->uuid,
    );

    $this->assertTrue($product->is($event->getProduct()));
}
```

Regarding testing projectors, we'll first cover them more in-depth and look at testing them afterwards.

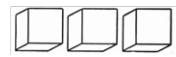
If you're following along, feel free to implement the removal of cart items yourself. We're going to end this chapter here and continue with more advanced projector techniques in the next one.

CHAPTER 6

PROJECTORS IN DEPTH

Up until this point, we've covered the technical basics of event sourcing: how to store events and project them. It might leave you wondering if we've actually gained much by using an event-driven approach. On the surface, it might seem like we've added several more steps to achieve the same goal.

That perception will change with this chapter: we're going to look at what a simple concept like projectors allow us to do.



Have you ever had a client ask about a new feature, where you had to tell them it would only work on data that's added after the feature was deployed? An example: imagine your client wanting to do some in-depth analysis on customer behaviour. Maybe they'd like to identify customers that took less than 10 minutes to check out their cart. Now, what if we didn't keep track of the checkout date and time yet? You would need to add a new checkout date field on the cart, which would be `null` for all carts existing before this feature.

With event sourcing however, we've got more flexibility. The fact that "an event happened" is information that's usually missing in a traditional application where you don't keep track of every change. An event-sourced system, on the other hand, keeps track of when events were originally dispatched. This essentially gives us a chronological log of everything that happened in our system.

So without having to change the database schema, we can solve our client's problem, because event sourcing captures the essence of time as well. From my experience, it's those time-sensitive questions that often offer the most value to the business. They allow us to monitor and report on our application flow throughout time and help us make decisions based on those metrics after the facts.

To solve our problem of customers with a quick checkout, there are in fact two solutions: we could either add a field on the `Cart` projection, or we could make a dedicated projection for this kind or report. Let's start with the first.

Our code is running in production and we need to add a new field to an existing table. We'd start with a migration:

```
public function up()
{
    Schema::table('carts', function (Blueprint $table) {
        $table->dateTime('checked_out_at')->nullable();
    }
}
```

Next, we need to make sure this field is projected as well. Let's go to our `CartProjector`. The `onCartCheckedOut` method was already implemented:

```
class CartProjector extends Projector
{
    // …

    public function onCartCheckedOut(
        CartCheckedOut $event
    ): void {
        $cart = Cart::find($event->cartUuid);

        $cart->writeable()->update([
            'state' => CheckedOutCartState::class,
        ]);
    }
}
```

Now we need to know when the event happened. Our event itself doesn't keep that data (since we didn't add a field for it), but there still is a way to determine its creation date. Whenever an event is serialized to the database, it's wrapped in an `EloquentStoredEvent` model — a normal Eloquent model. It doesn't only store the serialized event though; it also keeps track of some meta data, including its creation date. If we can retrieve the stored event model for this event, we can access its data. Luckily every event class automatically gets some meta data attached to it, including the ID that's used to store it in the database.

Knowing all of that, retrieving the stored event is nothing more than a normal database query:

```
$storedEvent = EloquentStoredEvent::find($event->storedEventId());
```

Bringing it all together, we'd refactor our projector like so:

```
class CartProjector
{
    // …

    public function onCartCheckedOut(
        CartCheckedOut $event
    ): void {
        $cart = Cart::find($event->cartUuid);

        $storedEvent = EloquentStoredEvent::find(
            $event->storedEventId()
        );

        $cart->writeable()->update([
            'state' => CheckedOutCartState::class,
            'checked_out_at' => $storedEvent->created_at,
        ]);
    }
}
```

However, we're not done yet! If we'd push these changes to production, they would only affect carts that were checked out *after* our deploy.

Remember how I explained that event sourcing enables us to rebuild our application state from scratch? We could essentially throw away all of our data, except the events themselves, and rebuild everything again. That's exactly what we're going to do: throwing away all carts and replay the events. We could, in theory, throw away and rebuild everything, but you can imagine how that would take lots of time. So instead of starting from scratch, we're only going to replay the events related to carts.

Before replaying, we'll need a way to reset our carts table. Projectors have a built-in solution for this: the `resetState()` method:

```
class CartProjector extends Projector
{
    public function resetState(): void
    {
        Cart::query()->delete();
    }

    // …
}
```

This method is run whenever we replay the projector. The final thing to do after deploying our changes is to actually replay this projector. This is done with the following artisan command:

```
php artisan event-sourcing:replay \
  "App\\Cart\\Projectors\\CartProjector"
```

After this command is finished, the events will have been replayed and the cart projector would have written the checked out date to all carts, including those in the past.

Throwing away all carts and rebuilding them can become quite expensive over time. Imagine a production system with years of data; rebuilding all carts could take hours, maybe even days. You wouldn't want your system to be down in "maintenance mode" all that time.

We're going to cover deployment strategies for such cases in a later chapter, but it's also worth mentioning another solution.

Instead of changing the cart itself, we could make a new projection, dedicated to the report our client is asking. They want to know what customers checked out their cart in less than 10 minutes, so let's make a dedicated table that keeps track of the amount of time a cart was active. Here's the migration:

```
Schema::create('cart_durations', function (Blueprint $table) {
    $table->uuid('uuid');
    $table->uuid('cart_uuid');
    $table->dateTime('created_at');
    $table->dateTime('checked_out_at')->nullable();
    $table->integer('duration_in_minutes')->nullable();
});
```

And here's the projection:

```
class CartDuration extends Projection
{
    protected $guarded = [];

    public $timestamps = [];

    protected $casts = [
        'created_at' => 'datetime',
        'checked_out_at' => 'datetime',
        'duration_in_minutes' => 'integer',
    ];
}
```

Finally, we'll have to make a projector. We're going to make a new class, but we'll listen to the existing `CartInitialized` and `CartCheckedOut` events:

```
namespace App\Cart\Projectors;

// …

class CartDurationProjector extends Projector
{
    public function onCartInitialized(
        CartInitialized $event
    ): void {
        $createdAt = EloquentStoredEvent::find(
            $event->storedEventId()
        )->created_at;

        (new CartDuration)->writeable()->create([
            'uuid' => $event->cartUuid,
            'cart_uuid' => $event->cartUuid,
            'created_at' => $createdAt,
        ]);
    }
```

```
    public function onCartCheckedOut(
        CartCheckedOut $event
    ): void {
        $checkedOutAt = Carbon::make(
            EloquentStoredEvent::find(
                $event->storedEventId()
            )->created_at
        );

        $cartDuration = CartDuration::find($event->cartUuid);

        $durationInMinutes = $checkedOutAt
            ->diffInMinutes($cartDuration->created_at);

        $cartDuration->writeable()->update([
            'checked_out_at' => $checkedOutAt,
            'duration_in_minutes' => $durationInMinutes,
        ]);
    }
}
```

Whenever a cart is initialized, this projector will create a new `CartDuration` projection, and store the date the event was triggered. Note that we're repurposing the cart's UUID as the duration's UUID. Since `Cart` and `CartDuration` are a one-to-one relation, it's OK to reuse it. Technically you wouldn't even need a dedicated UUID for this projection, but Laravel expects it to be there nevertheless.

Next, when a `CartCheckedOut` event is triggered, we update the existing `CartDuration` for that cart, and calculate the duration in minutes. When this projector is run for previously dispatched events, it will create a new `CartDuration` for all existing carts because of the `CartInitialized` event, as well calculate the duration when those carts were finally checked out.

The only thing left to do is deploy these changes, and replay the events on our newly created projector:

```
php artisan event-sourcing:replay \
  "App\\Cart\\Projectors\\CartDurationProjector"
```

You can see how we've created another interpretation of the same event stream by creating a new projector that listens to the same events. It's once again an illustration of how events are the only source of truth, and projections are only a side effect.

Besides being able to act upon things that happened in the past, there's another benefit we've gained: once our `CartDuration` projections are generated, it's very easy to read from them. We don't have to do any in-memory calculations or database joins: the data is right there, whenever we need it, in the right format. This also means that you could make dedicated projections for specific pages, even when those projections are based on the same event stream. No need to dynamically build an admin dashboard based on the projected carts. Let's just make a dedicated projection for that dashboard.

Finally, there's no more fear of missing out: you can add however many projections you want after your application has gone into production. As long as the events are stored, you're free to replay them as many times and in whatever way you'd like. We'll cover the power of projections even further when we discuss CQRS, a topic for a later chapter.

TESTING

Projectors usually don't do much more than moving data from events to projections. A classic projection test would consist of manually sending an event to a projector, and asserting whether the right projection was made.

```php
/** @test */
public function cart_initialized(): void
{
    $projector = new CartDurationProjector();

    $event = CartInitializedFactory::new()->create();

    $projector->onCartInitialized($event);

    $cartDuration = CartDuration::find($event->cartUuid);

    $this->assertNotNull($cartDuration);
}
```

In case of the cart duration calculation, there's a bit more to test:

```
/** @test */
public function cart_checked_out(): void
{
    $projector = new CartDurationProjector();

    $cartDuration = CartDurationFactory::new()
        ->createdAt(Carbon::make('2021-01-01 10:00'))
        ->create();

    $event = CartCheckedOutFactory::new()
        ->withCart($cartDuration->cart)
        ->createdAt(Carbon::make('2021-01-01 10:25'))
        ->create();

    $projector->onCartCheckedOut($event);

    $cartDuration->refresh();

    $this->assertEquals(
        25,
        $cartDuration->duration_in_minutes
    );
}
```

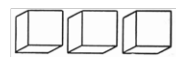
What we haven't tested yet is whether these projectors actually listen to the right events. That's something we'll come back to later when we're writing integration tests.

Caveats

Before ending this chapter, I need to mention a few caveats.

The first one I mentioned already: replaying events can be time-consuming. There are cases where this isn't a problem, but sometimes you need to be aware that your deployment strategy might become more complex.

Next, there's also the matter of which data to store in events. While making changes to projections is easy, making changes to events (like adding a property or removing one) is very difficult. Changing an event might have big consequences when replaying them. This is also a topic we'll spend a chapter on later in this book.



Despite these caveats, I think there's lots of value in using an event-sourced system. While some problems are harder to solve because of event sourcing, others become trivial. We'll dedicate the future chapters to further exploring everything that's to be gained from using event sourcing. Even more is possible, thanks to the simple concept of storing events.

CHAPTER 7

EVENT QUERIES

While projections are a powerful tool, they also have their shortcomings. Imagine your client asking for different kinds of reports. In that case, there might be some overhead to making a dedicated projection for every single report. If reports need to be filterable on top of that, you might end up with lots of iterations of the same projection.

Take this example: your client wants a report that shows earnings for products for a given period. In other words: both `product` and `period` are two variables. It would be impossible to generate a row for every possible period and product combination.

One way of solving such a problem with projections is to have a normalized data set, and group them together on the fly. You could keep a projection that lists earnings per product per day and filter and aggregate that data based on the user's request.

That is a valid approach to the problem, though there's an easier solution that skips the need for using an intermediate projection table: **event queries**.

An event query is a class that gathers a bunch of events and replays them on the fly. The difference with replaying projections is that an event query keeps track of "application state" *exclusively* in-memory: it won't store anything in a database. Every time you perform an event query, it will rebuild its internal state again. This principle allows you to build a

highly customizable query that can expose results based on the user's needs.

Let's start with our event query class:

```
namespace App\Reports;

use Illuminate\Support\Collection;
use Spatie\EventSourcing\EventHandlers\Projectors\EventQuery;
use Spatie\EventSourcing\StoredEvents\Models\
        EloquentStoredEvent;
use Spatie\Period\Period;

class EarningsForProductAndPeriod extends EventQuery
{
    public function __construct(
        private Period $period,
        private Collection $products
    ) {
        // …
    }
}
```

Whenever we create a new event query, we'll want to load the related events into memory. We've already seen that events are stored in the `EloquentStoredEvent` model, and that we can use it to query our stored events:

```
public function __construct(
    private Period $period,
    private Collection $products
) {
    EloquentStoredEvent::query()
        ->whereEvent(OrderCreated::class)
        ->whereDate(
            'created_at', '≥', $this->period->getStart()
        )
        ->whereDate(
            'created_at', '≤', $this->period->getEnd()
        )
        ->cursor()
        ->each(
            fn (EloquentStoredEvent $event) ⇒
                $this->apply($event->toStoredEvent())
        );
}
```

Since there are potentially millions of events stored in our database, it's best to keep performance in mind:

- we filter events based on their type - we're only interested in `OrderCreated` events; next
- we filter on the period to further narrow our query; and finally
- we use a cursor to load stored events incrementally instead of loading all of them into memory at once.

With all relevant events available, we loop over them and apply them using `$this->apply()`. Under the hood, this method looks for event listeners on the query class itself, and calls those listeners as if the event was dispatched just now; much like replaying events for a projector. Such a listener is a protected method on the event class itself, and looks like this:

```
class EarningsForProductAndPeriod extends EventQuery
{
    // …

    private int $totalPrice = 0;

    protected function applyOrderCreated(
        OrderCreated $orderCreated
    ): void {
        $orderLines = collect(
            $orderCreated->orderData->orderLineData
        );

        $totalPriceForOrder = $orderLines
            ->filter(function (OrderLineData $orderLineData) {
                return $this->products->first(
                    fn(Product $product) =>
                        $orderLineData->productEquals($product)
                ) !== null;
            })
            ->sum(
                fn(OrderLineData $orderLineData) =>
                    $orderLineData->totalPriceIncludingVat
            );

        $this->totalPrice += $totalPriceForOrder;
    }
}
```

Let's go through this step by step. The first thing you should note is that whenever a cart is checked out, an order is created. That's why we listen to the `OrderCreated` event: it's the proof the sale actually happened.

If events tend to grow large in data, it might be a good idea to use data transfer objects in them. That's exactly what happens with those `OrderData` and `OrderLineData` objects. They are simple objects containing data that can be serialized.

Next, we filter our collection of order lines by the products in them. We're only interested in products that match our predefined filter.

Finally, we add all the earnings from individual order lines together, and increase our internal `totalPrice` value. Remember that we're applying event after event, so we need to keep track of the grand total using an instance variable.

After all events are applied, the only thing left to do is expose the internal `totalPrice` to the outside world:

```
class EarningsForProductAndPeriod extends EventQuery
{
    // …

    public function totalPrice(): int
    {
        return $this->totalPrice;
    }
}
```

With all of this in place, we're now able to generate these reports on the fly, in memory:

```
$report = new EarningsForProductAndPeriod(
    Period::make('2021-01-01', '2021-02-01'),
    $collectionOfProducts,
);

$report->totalPrice();
```

In a way, you can think of event queries as projectors and projections combined: instead of manually replaying events into a database, we're replaying them on the fly in memory.

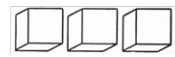
Event queries aren't the solution to all problems, but they are quite useful to solve specific questions that only need a relatively small amount of events and lots of flexibility. Also note that you're free to listen to multiple events within the same event query, and that you can expose whatever state you'd like.

TESTING

Event queries are isolated components, making them easy to test. All we need to do is seed some events and assert that the correct result is calculated:

```
/** @test */
public function total_price_is_calculated(): void
{
    $eventFactory = OrderCreatedFactory::new();

    $eventFactory
        ->withTotalPrice(10_00)
        ->onDate('2021-01-11')
        ->create();

    $eventFactory
        ->withTotalPrice(15_00)
        ->onDate('2021-01-10')
        ->create();

    $eventFactory
        ->withTotalPrice(15_00)
        ->onDate('2021-02-10') // Should not be counted
        ->create();

    $collectionOfProducts = collect([/* … */]);

    $report = new EarningsForProductAndPeriod(
        Period::make('2021-01-01', '2021-02-01'),
        $collectionOfProducts,
    );
```

```
    $this->assertEquals(25_00, $report->totalPrice())
}
```

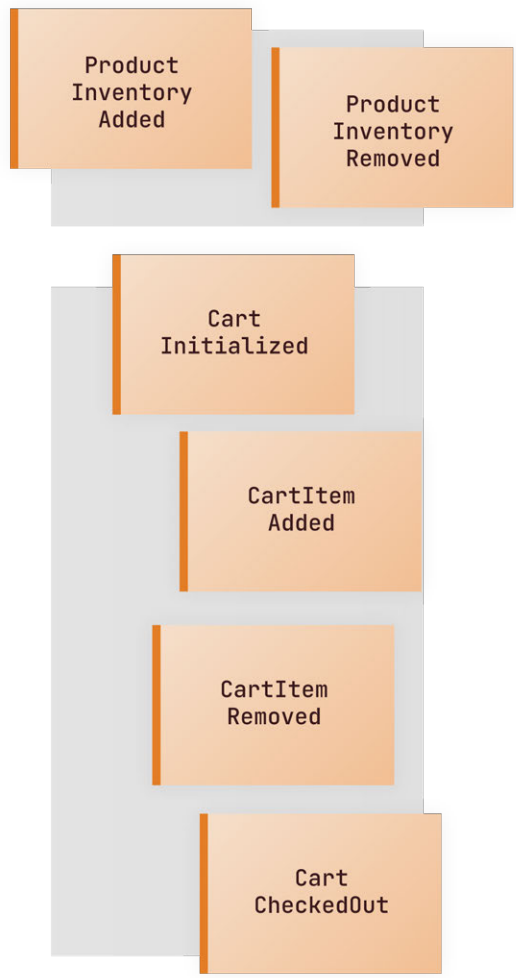


We'll build upon the idea of event queries even further in a future chapter, but first we're going to look at a crucial missing piece in order to manage event flow.

CHAPTER 8

REACTORS

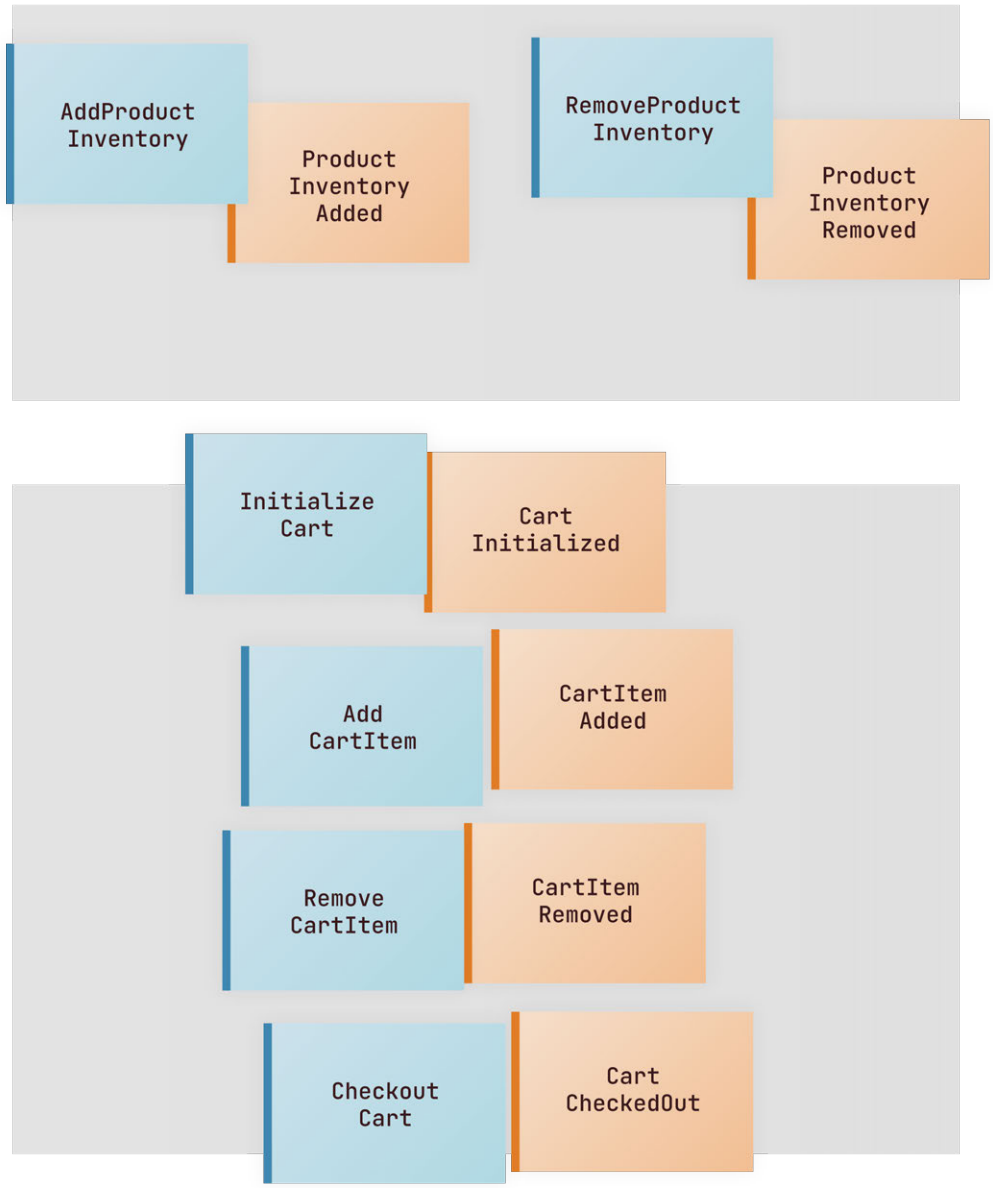
With stored events, projectors and event queries, we already have a basic toolset to design an event-driven system. There's still something important missing, though. Let's look at part of our event flow diagram and discover what's wrong:



Previously, we said that product inventory would be updated whenever a customer adds an item to a cart. That poses the question: how do we make sure this actually happens? In essence, we need a way to automatically react when an event happens to continue the flow. As you've guessed from the title, these kinds of classes are called **reactors**. Other popular names for the same concept are "process manager" or "policy"; sometimes you can see them being used to create "sagas", a topic we'll touch in another chapter. I'm making sure to list all these names, because depending on what literature you read or framework you use, they are named differently.

The idea is always the same, though: these classes provide a mechanism to keep the event flow going without user interactions. In this chapter, we'll build some reactors together and discover what they allow us to do. First, we'll have to make a few changes to our diagram.

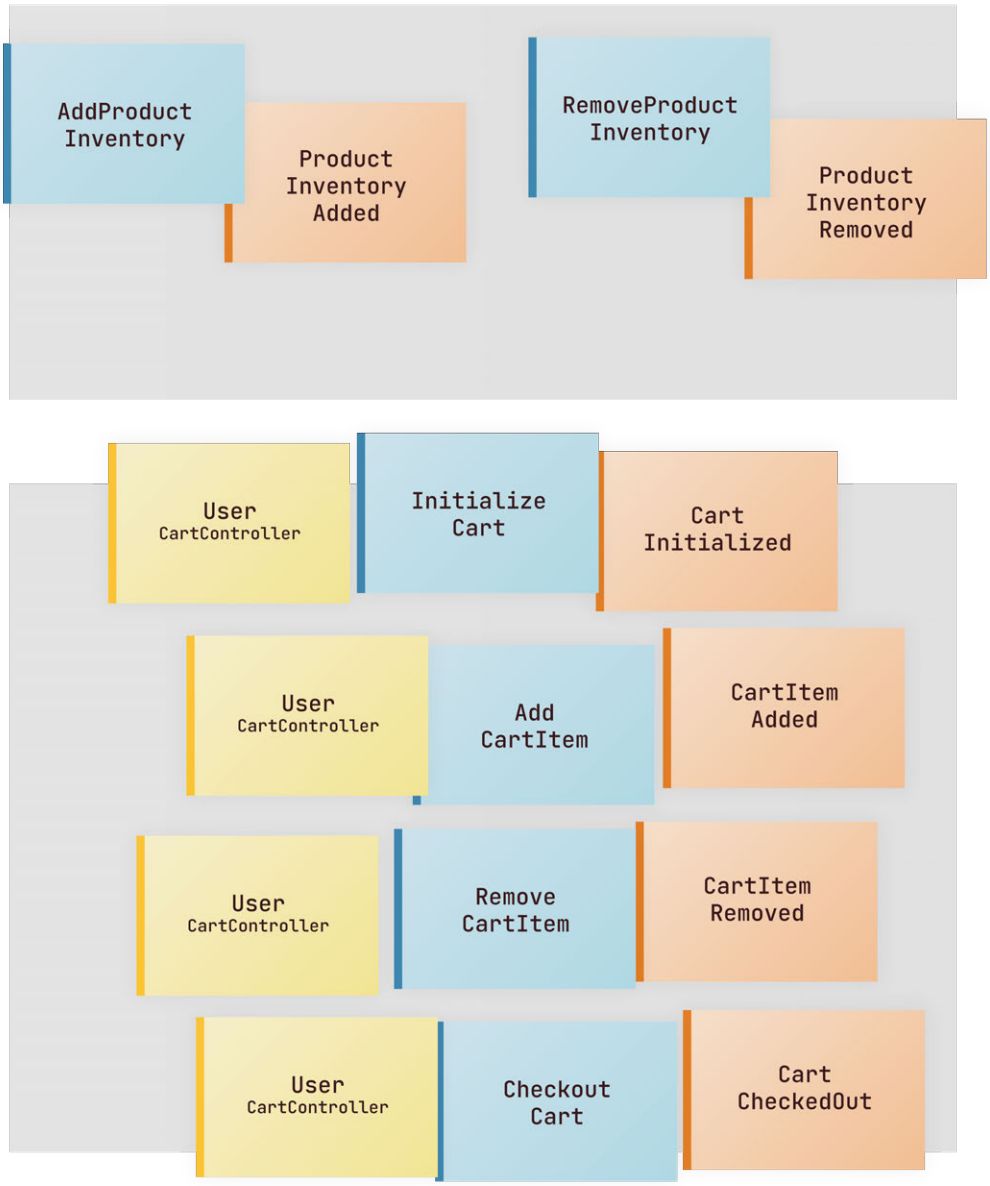
Let's start by adding actions to it: the classes that triggered the events. You'll note that often the action name and event name are similar to each other.



Even though the names are related, it's important to understand the differences between them: actions are used to set up and prepare the scene, while events are a statement that something happened in our system. You could say that actions are the "user interface" for developers who are building controllers, jobs, etc.; it's a way to reach into the underlying system.

Also note that according to the official event storming grammar, the blue cards should be called “commands”. We’ll discuss the differences between actions and commands in chapter 14. For now, we can assume they are the same.

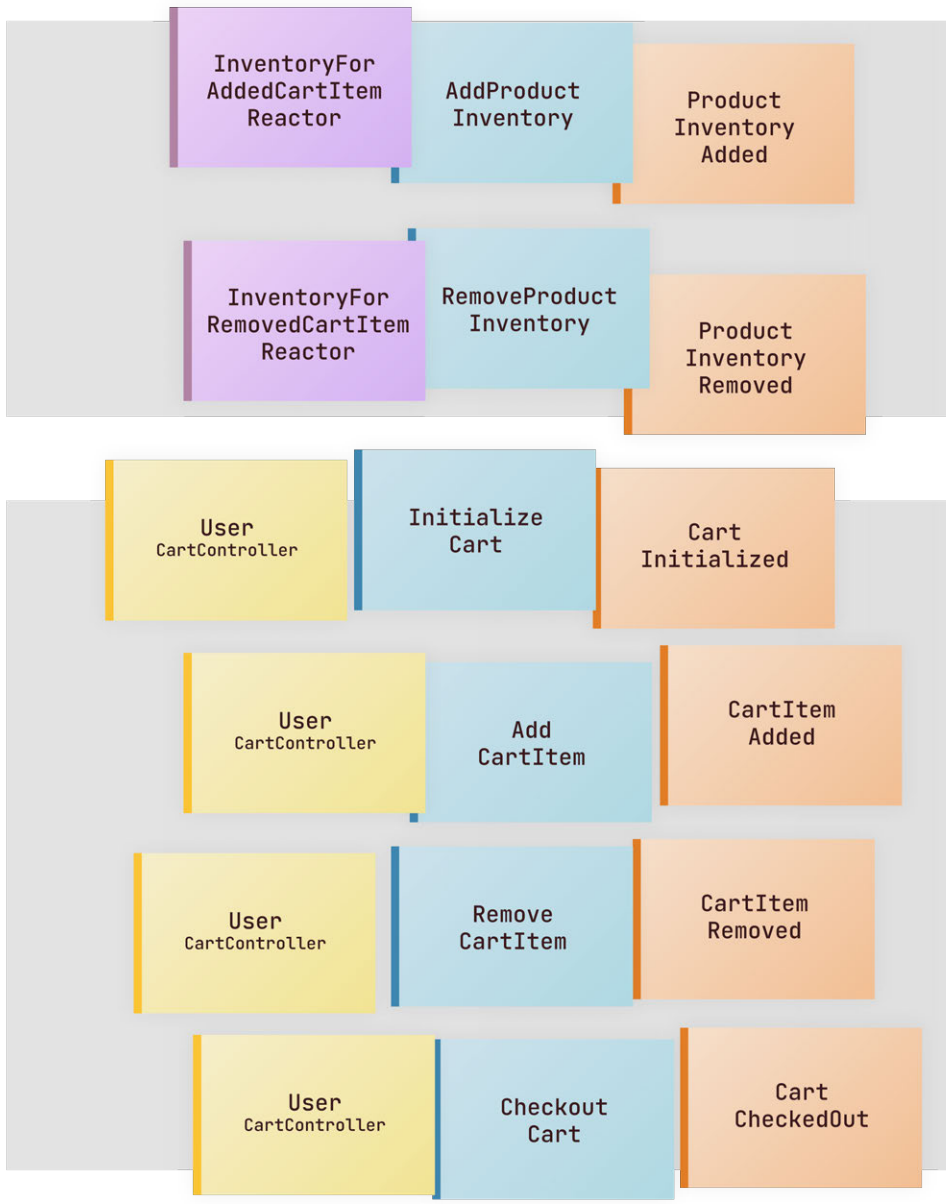
Speaking of developers using actions in controllers: there’s another type of sticky note we’re going to add to our diagram. These ones are used to visualize where or by who actions are triggered. This could, for example, be a user sending an HTTP request via a controller:



It’s pretty clear that all cart events are the direct result of a user clicking a button somewhere on the website. So what about managing product inventory? Every time an item gets added or removed, we should update the inventory accordingly. These are the places we’ll use reactors.

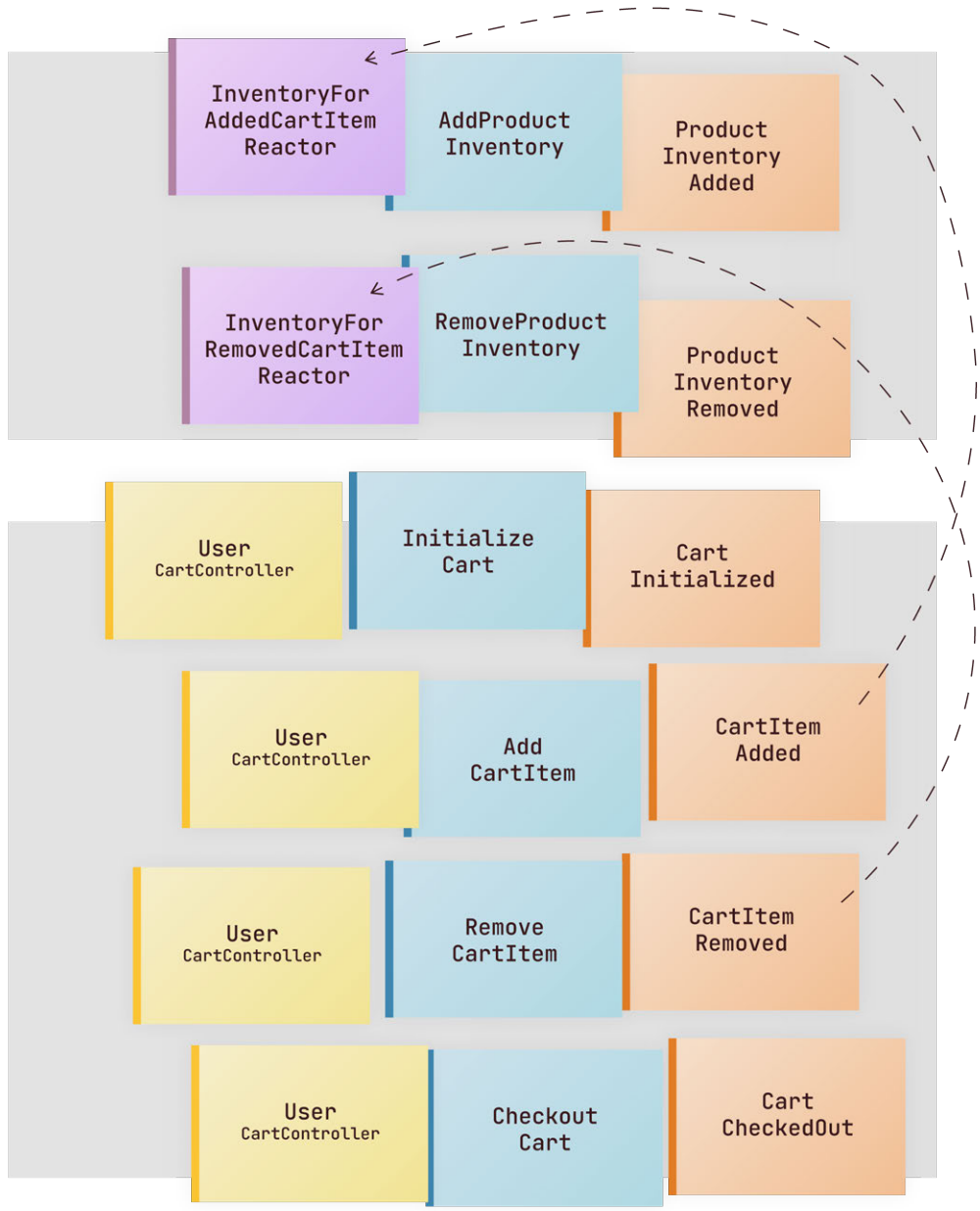
Reactors are actually nothing more than normal event listeners. Unlike projectors, reactors won’t be replayed, so they are only run once after an event is dispatched. This is the perfect mechanism for triggering an event automatically after another one, in our case: updating the inventory after an item was added or removed.

First, we're going to complete our diagram further, and then we'll code our first reactor. The convention within event storming is to use purple notes for these types of automatic processes that trigger something else. If you say "when this happened, then that should happen next", you should use a purple note.



As you can see, I prefer to use very clear names here, even though they are a bit longer. I find it essential for a name to be absolutely clear about what it does: "this is the reactor that handles inventory updates when a cart item is added" seems to me like the best description possible.

We'll also add some arrows between the events and the corresponding reactors to clarify this flow:



Now that we know the flow of our event diagram, we're going to write the actual code. Let's start with an empty reactor class:

```
namespace App\Inventory\Reactors;

use Spatie\EventSourcing\EventHandlers\Reactors\Reactor;

class InventoryForAddedCartItemReactor extends Reactor
{
}
```

Technically, it's possible to have several listeners in one reactor, but because they are such a crucial part in managing the process flow, I find it best to only have one listener per reactor. In this case: we want to listen for the `CartItemAdded` event, and remove available inventory as the result.

```
class InventoryForAddedCartItemReactor extends Reactor
{
    public function onCartItemAdded(
        CartItemAdded $cartItemAdded
    ): void {
        $product = Product::find($cartItemAdded->productUuid);

        (new RemoveProductInventory)(
            $product,
            $cartItemAdded->amount,
        );
    }
}
```

If we were to describe this reactor, we'd say something like "when a cart item is added, we'll look up the product associated with that cart item and remove its inventory".

This is a very simple example, though. Let's make it a bit more interesting. Let's say not every product has inventory, instead some have an endless supply (think about digital products). You can check for these kinds of conditions in a reactor and act accordingly:

```
class InventoryForAddedCartItemReactor extends Reactor
{
    public function onCartItemAdded(
        CartItemAdded $cartItemAdded
    ): void {
        $product = Product::find($cartItemAdded->productUuid);

        if (! $product->hasInventory()) {
            return;
        }

        (new RemoveProductInventory)(
            $product,
            $cartItemAdded->amount,
        );
    }
}
```

While moving the event flow forward is a crucial task of reactors, they can handle other things as well. Imagine you have to send a mail to the customer as the result of the `OrderCompleted` event; you wouldn't want this mail to be re-sent every time events were replayed. It only needs to happen once, and that's exactly what we'd use a reactor for. Such a reactor would look like this:

```
class MailCompletedOrderReactor extends Reactor
{
    public function onOrderCompleted(
        OrderCompleted $orderCompleted
    ): void {
        $order = Order::find($orderCompleted->orderUuid);

        $order->customer->notify(new OrderCompletedMail($order));
    }
}
```

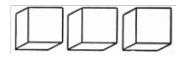
TESTING

Testing reactors on their own is very similar to unit testing projectors: we can manually make a reactor instance, and pass it an event. It could look something like this:

```
public function a_mail_is_sent_after_completing_the_order()
{
    Notification::fake();

    $event = OrderCompletedFactory::new()->create();

    $reactor = new MailCompletedOrderReactor();

    $reactor->onOrderCompleted($event);

    $order = Order::find($orderCompleted->orderUuid);

    Notification::assertSentTo(
        [$order->customer],
        OrderCompletedMail::class
    );
}
```

Note that reactors will probably always deal with side effects and that's why faking or mocking will be crucial with them. There's a specific issue with that when it comes to using actions: they are difficult to mock. We'll discuss these shortcomings in the command bus chapter.

Besides testing the reactor itself, we also want to test whether it handles the right event. I prefer to write a few integration tests to check whether the whole flow works. I'll show how to write such tests at the end of the next chapter, because it brings everything together.



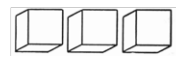
As you can imagine, reactors are an indispensable part of our event-driven toolset. They make complex process flows possible and still allow for a decoupled, event-driven system.

There is *one* more missing piece we need to look at in-depth, and that one we'll cover in the next chapter.

CHAPTER 9

AGGREGATE ROOTS

A common use case in any application is to make decisions based on state. For example: “a cart cannot be checked out when it’s empty” or “a cart should at least have $20 in it before free shipping”. We should discuss where we’re going to handle these kinds of business rules.

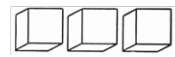


Until now, we’ve been interacting with aggregates and their related projections via actions, so they are probably the first idea that comes to mind on where to handle business logic. It’s not the ideal place, though: actions are already concerned with translating user input to structured data that our events can work with. As well as that, this approach would spread core business functionality across several classes and often result in code duplication.

Let’s entertain the idea of having one entry point into our aggregate, one that knows about all business rules and can interact with the underlying code. The DDD community calls this concept an **aggregate root**. Here’s Martin Fowler again, explaining them:

> *An aggregate will have one of its component objects be the aggregate root. Any references from outside the aggregate should only go to the aggregate root. The root can thus ensure the integrity of the aggregate as a whole.*

We can still use actions as a way to interact with the aggregate root, but the aggregate root is the core component to interact with.



Let's look at an example: "a cart cannot be checked out if there are no items in it". Originally, the `CheckoutCart` action would have looked something like this:

```
class CheckoutCart
{
    public function __invoke(Cart $cart): Cart
    {
        (new CartCheckedOut(
            cartUuid: $cartUuid,
            date: now(),
        ))->handle();

        return $cart->refresh();
    }
}
```

Let's start by creating our first aggregate root and move some code around. This is the `CartAggregateRoot`:

```
namespace App\Cart;

use Spatie\EventSourcing\AggregateRoots\AggregateRoot;

class CartAggregateRoot extends AggregateRoot
{
    public function checkout(): self
    {
        $this->recordThat(new CartCheckedOut(
            cartUuid: $this->uuid(),
            date: now(),
        ));

        return $this;
    }
}
```

The first thing to note is that `CartAggregateRoot` extends from the abstract `AggregateRoot` class. Similar to projectors and reactors, this class adds basic functionality in dealing with events. Specifically, it adds a `recordThat()` method which we should use instead of calling `handle()` directly on the event.

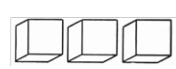
On top of storing and dispatching events, `recordThat()` does one more thing: it automatically adds a UUID to the events it's recording. This UUID has a very important purpose, so let's discuss it in depth.

Every cart has its own unique identifier. We've seen before that we generate a UUID when calling the `InitializeCart` action, and pass that UUID to our `CartInitialized` event. In the end there's a projector that uses this UUID and creates a new `Cart` projection with it.

Does this UUID only represent the identity of the `Cart` projection, or does it represent something more? When we think about the meaning of "a cart", we could say it's more than only its projection. "A cart" is the result of a stream of events, where a `Cart` projection is simply one of the side effects of that event stream. Our cart isn't just one row in a database with a UUID - it's a collection of events and projections. It's this collection that we call the "aggregate", so naturally the UUID for "a cart" doesn't only identify one projection, but the aggregate as a whole. It's the aggregate root that stores this UUID and automatically attaches it to all recorded events.

So, where does this UUID come from? Exactly the same place as before: it's generated in the action when initializing the cart. Instead of directly passing it to events, we use it to retrieve an aggregate root:

```
class InitializeCart
{
    public function __invoke(Customer $customer): Cart
    {
        $cartUuid = uuid();

        $cartAggregateRoot = CartAggregateRoot::retrieve(
            $cartUuid
        );

        // …

        return Cart::find($cartUuid);
    }
}
```



If the UUID of an aggregate root is attached to all of its events, that means we can also retrieve the previously stored events from the database for that aggregate root. Those events can be replayed internally every time an aggregate root is retrieved. Does that sound familiar? It's exactly what we did with event queries.

There's one important difference, though: event queries are used to query the existing events; they represent a read-only model of an event stream. However, aggregate roots will use those previous events to make new decisions, which means that an aggregate root's state shouldn't be tampered with from the outside.

Let's look again at our checkout method: we're not allowed to continue if a cart is empty. Instead of querying projections to determine whether our business rules are valid, we can look at the events themselves. Every time an item gets added, we'll keep track of it in our aggregate root. Every time one gets removed, we'll do the same.

Tracking such changes to build its internal state is done in separate protected methods on the aggregate root. You don't need to handle every event if it's not necessary, and you have the flexibility to make changes to an aggregate root's internal state whenever you want: nothing is persisted, so you don't have to worry about data migrations. Let's imagine two more methods on our `CartAggregateRoot`: `addItem` and `removeItem`:

```
class CartAggregateRoot extends AggregateRoot
{
    // …

    public function addItem(
        string $cartItemUuid,
        Product $product,
        int $amount
    ): self {
        $this->recordThat(new CartItemAdded(
            cartItemUuid: $cartItemUuid,
            productUuid: $product->uuid,
            amount: $amount,
        ));

        return $this;
    }

    public function removeItem(CartItem $cartItem): self
    {
        // Check whether the item actually exists in this Cart…

        $this->recordThat(new CartItemRemoved(
            cartItemUuid: $cartItem->uuid,
        ));

        return $this;
    }
}
```

These two methods record the events, but they don't apply to any state. Remember that an aggregate root is rebuilt from scratch every time it's retrieved, so we must split "recording events" and "managing state". When replaying events, only the state-related methods should be called, not the original event recording.

These listener methods are exactly what you'd expect them to be, just like with event queries:

```
class CartAggregateRoot extends AggregateRoot
{
    // …

    private array $cartItems = [];

    protected function onItemAdded(
        CartItemAdded $event
    ): void {
        $this->cartItems[$event->cartItemUuid] = true;
    }

    protected function onItemRemoved(
        CartItemRemoved $event
    ): void {
        unset($this->cartItems[$event->cartItemUuid]);
    }
}
```

In our case, we only need to know whether a cart is empty or not. Keeping track of added items via a simple array does the job.

Let's go back to our `checkout()` method now. Here we can do a check on whether the cart is empty or not:

```
class CartAggregateRoot extends AggregateRoot
{
    // …

    public function checkout(): self
    {
        if (count($this->items) === 0) {
            throw new EmptyCartException();
        }

        $this->recordThat(new CartCheckedOut(
            date: now(),
        ));

        return $this;
    }
}
```

And let's not forget to check whether a cart item actually exists before removing it too (we skipped over that part at first because we didn't have our local state yet):

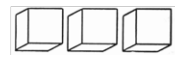
```
class CartAggregateRoot extends AggregateRoot
{
    // …

    public function removeItem(CartItem $cartItem): self
    {
        if (! isset($this->items[$cartItem->uuid])) {
            throw new CartItemNotFound();
        }

        $this->recordThat(new CartItemRemoved(
            cartItemUuid: $cartItem->uuid,
        ));

        return $this;
    }
}
```

Note how we throw exceptions from our aggregate roots to indicate something is wrong. It isn't an aggregate root's task to determine *how* an invalid operation is handled. It should only signify that something is wrong. The developer should handle these exceptions on another level.

In some cases, exceptions are merely there to prevent a truly invalid operation from happening, which only would occur if the developer made a mistake during coding. These kinds of exceptions are fine to hard-crash since they should be properly handled before deploying. You can think of those exceptions as a last line of defence against ourselves, when we need to come back to a codebase months, maybe even years after we originally worked in it. It's plausible that we forget some intricate details of our system over time, and it's good to know there's a safety net in place.

Other exceptions have valid feedback for the user, though. For example, a cart that cannot be checked out because it's empty, should be communicated to the user. Controllers can catch those exceptions, and add, for example, validation errors or flash messages to get the feedback back to the user. What's most important is that the aggregate root detected the problem and didn't have to worry about how to handle it.

In essence, we can think of our aggregate root as a state machine: it keeps track of state internally, and can make decisions based on that state, such as whether an exception is thrown or whether an event is recorded. Whenever we retrieve an aggregate root, its state is restored based on the previous events. This is also called **reconstitution**, which could be considered as "rebuilding".



I mentioned that the aggregate root's UUID is automatically added to its recorded events. Let's return to our `CartProjector` and see how we can access it:

```
class CartProjector extends Projector
{
    // …

    public function onCartCheckedOut(
        CartCheckedOut $event
    ): void {
        $cart = Cart::find($event->aggregateRootUuid());

        $cart->writeable()->update([
            'state' => CheckedOutCartState::class,
            'checked_out_at' => $event->date,
        ]);
    }
}
```

Every event that's recorded within an aggregate root will have this `aggregateRootUuid()` method. While it's a handy shorthand, it sometimes might become difficult to know where a specific event was recorded and thus what aggregate root it's linked to.

Personally, I find it easier to explicitly store the aggregate root UUID on the event and give it a proper name, like so:

```
class CartCheckedOut extends ShouldBeStored
{
    public function __construct(
        public string $cartUuid,
        public Carbon $date,
    ) {
    }
}
```

You can fill in the cart UUID from within the aggregate root when recording the event by using the `uuid()` method.

```
class CartAggregateRoot extends AggregateRoot
{
    // …

    public function checkout(): self
    {
        if (count($this->items) === 0) {
            throw new EmptyCartException();
        }

        $this->recordThat(new CartCheckedOut(
            cartUuid: $this->uuid(),
            date: now(),
        ));

        return $this;
    }
}
```

Feel free to use whatever style fits you and your team the best.

There's one more thing to mention: how to use these aggregate roots from the outside. It's actually pretty straightforward; here's an example of using it in the `CheckoutCart` action:

```
class CheckoutCart
{
    public function __invoke(Cart $cart): Cart
    {
        $cartAggregateRoot = CartAggregateRoot::retrieve(
            $cart->uuid
        );

        $cartAggregateRoot->checkout()->persist();

        return $cart->refresh();
    }
}
```

Note the `persist()` method on the aggregate root: the aggregate root will hold off on storing events and dispatching them to the outside until `persist()` called. Persisting makes sure you're only storing events when all business rules have been checked. Take, for example, an action to add multiple items at once:

```
class AddCartItems
{
    public function __invoke(
        Cart $cart,
        Product ...$products
    ): Cart {
        $cartAggregateRoot = CartAggregateRoot::retrieve(
            $cart->uuid
        );

        foreach ($products as $product) {
            $cartAggregateRoot->addItem(
                cartItemUuid: uuid(),
                product: $product,
                amount: 1,
            );
        }

        $cartAggregateRoot->persist();

        return $cart->refresh();
    }
}
```

If we were adding ten products and there was a problem when adding the third one, we wouldn't want some events to have been stored and some not. That's the safety we get from using `persist()`.

TESTING

Testing aggregate roots can be done on several levels. Since they contain your application's core functionality, it's crucial to test them right.

First, there's unit testing: testing aggregate roots in isolation. The event sourcing package has built-in support for these kinds of tests by allowing you to create an aggregate root fake. These kinds of fakes add useful test methods to do assertions on recorded events, but also won't trigger any side effects like reactors or projectors.

The idea of such tests is as follows:

- set up the aggregate root fake with a set of previously recorded events;
- do something with the aggregate root;
- assert that either exceptions were thrown or new events were recorded.

Here's one example, but remember to check out the test suite in the shop package as well:

```
/** @test */
public function can_add_item(): void
{
    CartAggregateRoot::fake(self::CART_UUID)
        ->given([
            $this->cartInitializedFactory->create(),
        ])
        ->when(function (CartAggregateRoot $cartAggregateRoot) {
            $cartAggregateRoot->addItem(new AddCartItem(
                self::CART_UUID,
                self::CART_ITEM_UUID,
                $this->product,
                1
            ));
        })
        ->assertRecorded([
            $this->cartItemAddedFactory->create(),
        ]);
}
```

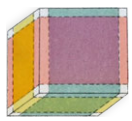
Aggregate root fakes are a powerful tool for writing tests, so make sure to read the documentation of the event sourcing package when you're going to use them.

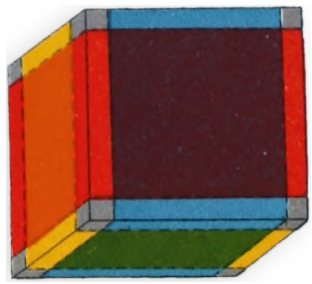
Next, there are integration tests that check whether the flow works as a whole. I like to start with actions in these kinds of tests and assert that projections and other expected side effects are correctly handled.

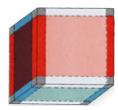
```
/** @test */
public function test_add_item(): void
{
    $product = ProductFactory::new()->create();

    $cartItem = (new AddCartItemAction)(
        cart: $this->cart,
        product: $product,
        amount: $amount,
    );

    $this->assertEquals(1, $this->cart->refresh()->items->count());
    $this->assertTrue($cartItem->product->is($product));
}
```

Remember that you don't need redundant tests but also that your aim is to cover as much code as possible. You'll need to balance the number of unit tests and integration tests throughout your project. Unfortunately, there's no fixed formula to determine whether you need to write a unit test, integration test, or a combination of both. You'll have to make these kinds of decisions on a case-by-case basis.

I find though, that aiming for as much coverage with as little as possible overlap between unit tests and integration tests is often a great way to start.







PART 2

ADVANCED PATTERNS

Now that we understand our basic building blocks — the event bus, events, projectors, event queries, reactors and aggregate roots — it's time to start applying them to real-world examples. You know from experience that the real-world is always significantly more difficult to comprehend compared to examples in isolation.

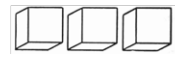
We'll still work with our example project, but we're going to address real-world problems I've encountered in real-world projects.

This is the part where you'll actually learn what it feels like to use event sourcing in practice, outside of a book.

CHAPTER 10

STATE MANAGEMENT IN AGGREGATE ROOTS

Using aggregate roots as the place to bundle business logic has many benefits, but they also have the potential to become large and complex. During the next two chapters, we'll discuss ways to prevent such problems from happening and to keep aggregate roots concise and maintainable.



As we've discovered in the previous chapter, aggregate roots are, in essence, a state machine, and one common issue that causes aggregate roots to grow is state management. Something that often comes with state management is checking whether an action may be performed in the current state which is a common use case in aggregate roots. For example, you might want to know whether a cart is checked out or not or whether it is empty or not. Early in my programming career, I often used booleans to represent this kind of state:

```
class CartAggregateRoot extends AggregateRoot
{
    private bool $pending = true;

    private bool $checkedOut = false;

    private bool $failed = false;

    private bool $paid = false;

    // …
}
```

Managing such boolean flags becomes a bit of a hassle. Even this small example already allows for 2^4 — that is 16 possibilities. In a recent project of mine, we identified 7 possible boolean flags in one aggregate root, resulting in 128 different combinations. Mind you, not all 128 combinations made sense, but can you imagine having to deal with all of those possibilities in your head?

You can see why it's best to avoid boolean flags to represent state, especially in aggregate roots. We're going to discuss two alternative ways to deal with state and to avoid the problems that come with boolean flags.

The first solution is to represent state as a single field with several possible values. You could use a string field or an enum. With enums available as of PHP 8.1, I'm going to use them in our example:

```
enum CartStatus {
    case pending;
    case checkedOut;
    case failed;
    case paid;
}
```

Our aggregate root would look like this:

```
class CartAggregateRoot extends AggregateRoot
{
    private CartStatus $status;

    public function __construct()
    {
        $this->status = CartStatus::pending;
    }
}
```

By setting a state in the constructor, we make sure this aggregate root always has a valid state as soon as it exists. Whenever events are applied, we can change the state of this aggregate root accordingly:

```
class CartAggregateRoot extends AggregateRoot
{
    // …

    protected function applyCheckedOut(
        CartCheckedOut $event
    ): void {
        $this->status = CartStatus::checkedOut;
    }

    protected function applyFailed(
        CartFailed $event
    ): void {
        $this->status = CartStatus::failed;
    }

    protected function applyPaid(
        CartPaid $event
    ): void {
        $this->status = CartStatus::paid;
    }
}
```

Every time an aggregate root is retrieved and its events are reconstituted, our state will be automatically be updated to the current one.

Thanks to this state field, we can now write simple checks instead of micromanaging boolean flags:

```
class CartAggregateRoot extends AggregateRoot
{
    // …

    public function addItem(/* … */): self
    {
        if (! $this->status === CartStatus::pending) {
            throw new InvalidCartStatus();
        }

        // …
    }

    public function checkout(/* … */): self
    {
        if (! $this->status === CartStatus::pending) {
            throw new InvalidCartStatus();
        }

        // …
    }

    public function pay(/* … */): self
    {
        if (! $this->status === CartStatus::checkedOut) {
            throw new InvalidCartStatus();
        }

        // …
    }
```

```
    public function fail(/* … */): self
    {
        if (! $this->status === CartStatus::checkedOut) {
            throw new InvalidCartStatus();
        }

        // …
    }
}
```

While a single state field works fine in many situations, there are cases that require a more complex solution. Imagine our cart has a special kind of state: it can be locked by an admin user. Being locked means that no items can be added or removed anymore, but the cart can still be checked out. In addition, if a cart payment fails when it's locked, it should stay in its failed state whereas it should transition back to pending if the cart wasn't locked.

We could model this locking mechanism as a boolean, combining that one with our existing state field and it would probably work fine in this case. But can you imagine a few more such rules popping up? If that's the case, you'll be needing a more robust solution to prevent boolean flags from polluting your aggregate root once more. To do this, we'll use the state pattern. Using the state pattern, we represent each state with a dedicated class, and each of those classes can model the business rules unique to that state.

We'll start by making changes to our aggregate root. By doing so, we'll get an idea of what kind of functionality our state classes should provide. I like working backwards in this case to prevent making the solution more complex than it needs to be. So first, we'll write some failing code in our aggregate root and afterwards fix it by building our state classes.

There are a few interesting use cases that our aggregate root should support:

- locking a cart;
- determining whether adding an item is allowed based on the lock state; and
- deciding what the next state should be after failing a cart, also based on the lock state.

We'll look at those three scenarios in isolation. First, let's set up our aggregate root and take it from there:

```
class CartAggregateRoot extends AggregateRoot
{
    private CartState $state;

    public function applyLock(CartLocked $event): void
    {
        // We should mark our current state as locked
    }

    public function addItem(/* … */): self
    {
        // Are we allowed to make changes to the cart?
        if (! $this->state->changesAllowed()) {
            throw new CannotChangeCart();
        }
    }
```

```
    protected function applyFailed(CartFailed $event): void
    {
        // Do we transition to pending or failed?
        $this->state = $this->state->nextState();
    }
}
```

Having started from our aggregate root, it's clear that our state needs to provide at least two methods: `changesAllowed()` and `nextState()`. We're at the point where we can create an abstract `CartState` class:

```
abstract class CartState
{
    abstract public function changesAllowed(): bool;

    abstract public function nextState(): CartState;
}
```

Next, we implement this class for all our states, four in total:

```
class Pending extends CartState
{
    public function changesAllowed(): bool
    {
        return true;
    }

    public function nextState(): CartState
    {
        return new CheckedOut();
    }
}
```

```
class CheckedOut extends CartState
{
    public function changesAllowed(): bool
    {
        return false;
    }

    public function nextState(): CartState
    {
        return new Paid();
    }
}
```

```
class Paid extends CartState
{
    public function changesAllowed(): bool
    {
        return false;
    }

    public function nextState(): CartState
    {
        return new Paid();
    }
}
```

```
class Failed extends CartState
{
    public function changesAllowed(): bool
    {
        return false;
    }

    public function nextState(): CartState
    {
        return new Failed();
    }
}
```

Some implementations are straightforward: a paid cart should never be allowed to change anymore and neither should a checked out cart. But in the pending state, we should keep track of the locked state as well; likewise, the failed state needs to determine the next one based on this locked state.

Let's model the locked state with a class as well and call it `CartLockState`. It has two implementations: `Locked` and `Unlocked` with the same methods as `CartState`:

```
abstract class CartLockState
{
    abstract public function changesAllowed(): bool;

    abstract public function nextState(): CartState;
}
```

```
class Locked extends CartLockState
{
    public function changesAllowed(): bool
    {
        return false;
    }

    public function nextState(): CartState
    {
        return new Failed();
    }
}
```

```
class Unlocked extends CartLockState
{
    public function changesAllowed(): bool
    {
        return true;
    }

    public function nextState(): CartState
    {
        return new Pending();
    }
}
```

Next we combine `CartState` and `CartLockState` together: every `CartState` will get an internal `CartLockState` as well:

```
abstract class CartState
{
    public function __construct(
        protected CartLockState $lockState
    ) {
    }

    // …
}
```

Don't forget to change all existing `nextState()` implementations as well because we now need to pass the lock state whenever creating a new cart state.

Finally, we can use the lock state to determine what should happen in `Pending` and `Failed`:

```
class Pending extends CartState
{
    public function changesAllowed(): bool
    {
        return $this->lockState->changesAllowed();
    }

    // …
}
```

```
class Failed extends CartState
{
    // …

    public function nextState(): CartState
    {
        return $this->lockState->nextState();
    }
}
```

We're not done yet, though. Let's go back to our aggregate root to glue everything together:

```
class CartAggregateRoot extends AggregateRoot
{
    private CartState $state;

    public function __construct()
    {
        $this->state = new Pending(new Unlocked());
    }

    public function applyLock(CartLocked $event): void
    {
        // We should mark our current state as locked

        $this->state = $this->state->locked();
    }
}
```

Our initial state will be `Pending` and `Unlocked`. Whenever we apply the `CartLocked` event, we update the current state and change its lock state to `Locked`. In order to do so, we've added one more method to `CartState` as a way to toggle the locked state:

```
abstract class CartState
{
    // …

    public function locked(): self
    {
        $clone = clone $this;

        $clone->lockState = new Locked();

        return $clone;
    }
}
```

TESTING

The state flow in aggregate roots is one of the most crucial parts of your application and should be tested thoroughly. Whether we use state enums or apply the state pattern, it's still the aggregate root that requires most testing; implementing states is merely a small implementation detail.

So either way, you'd write normal aggregate root tests like we've seen before; and test which events are triggered and which not, based on the aggregate root's state.

You can imagine how this indirect approach to testing states isn't always ideal. We'll come back to states in the chapter after the next one and discuss how to model and test dedicated state machines. But we need to cover another topic first in order to understand state machines.

In these examples, I've only shown the relevant parts to understand how states and the state pattern are used. If you want to see the full picture, make sure to check out the demo application.

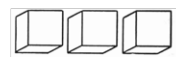
Whether you use simple enum-like statuses or state classes is up to you, both have their own up or downsides. I'd definitely recommend starting with the simpler approach and use an enum state and refactor when needed.

In the next chapter, we'll continue to explore how we can simplify complex aggregate roots by splitting them into several parts.

CHAPTER 11

AGGREGATE PARTIALS

This chapter will focus on splitting an aggregate root into several smaller pieces. At all times, we'll respect the rule to only have one entry point into our aggregate — the aggregate root — but we'll allow an aggregate to be split internally into different parts. Those parts are called **aggregate entities** or **aggregate partials**. Please note that, in the latest version of the event sourcing package, we've decided to use the name "aggregate partials" in order to avoid confusion with entities in DDD.



Sooner or later, you'll notice that aggregate roots start to grow, even when state management is moved to separate classes. The reality is that many business processes are complex and require us to write more than a few lines of code. Even within the same aggregate (a cart for example) there's already the potential for it to grow larger than you'd want.

Aggregate partials help solve this problem by moving away parts of an aggregate root to dedicated classes. So what exactly can be extracted away? The good news is that you can always make these decisions later in the development process. It's extremely easy to refactor an aggregate root to partials, even when the project is already in production.

As an example, we're going to make a dedicated partial class that manages cart items. It would look something like this:

```php
namespace Spatie\Shop\Cart\Partials;

use Spatie\EventSourcing\AggregateRoots\AggregatePartial;

class CartItems extends AggregatePartial
{
    private array $cartItems = [];

    public function isEmpty(): bool
    {
        return count($this->cartItems) === 0;
    }
}
```

The power of partials is that they work almost exactly the same way as aggregate roots: they can record and apply events and build their internal state from reconstituted events.

This means that we can move all item-related code from our `CartAggregateRoot` to our newly created `CartItems` class:

```
class CartItems extends AggregatePartial
{
    // …

    public function addItem(
        string $cartItemUuid,
        Product $product,
        int $amount
    ): self {
        $this->recordThat(new CartItemAdded(
            cartItemUuid: $cartItemUuid,
            productUuid: $product->uuid,
            amount: $amount,
        ));

        return $this;
    }

    protected function applyCartItemAdded(
        CartItemAdded $cartItemAdded
    ): void {
        $this->cartItems[$cartItemAdded->cartItemUuid] = null;
    }
```

```
    public function removeItem(
        CartItem $cartItem
    ): self {
        $exists = array_key_exists(
            $cartItem->uuid,
            $this->cartItems,
        );

        if (! $exists) {
            throw new UnknownCartItem();
        }

        $this->recordThat(new CartItemRemoved(
            cartUuid: $this->aggregateRootUuid(),
            cartItemUuid: $cartItem->uuid,
        ));

        return $this;
    }

    protected function applyCartItemRemoved(
        CartItemRemoved $cartItemRemoved
    ): void {
        unset($this->cartItems[$cartItemRemoved->cartItemUuid]);
    }
}
```

Let's note a few things:

- `$this->recordThat()` is available, just like on aggregate roots;
- apply methods also work just like aggregate roots; and finally
- since a partial is linked to an aggregate root (the aggregate root is passed to the partial when constructing it), it can access its UUID using `$this->aggregateRootUuid()`.

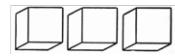
Next we need to change our aggregate root. Since it is still the entry point for all code living outside the aggregate, we'll need to provide the `addItem()` and `removeItem()` methods on it as well, but they will do nothing more than to use the underlying `CartItems` partial instead:

```
class CartAggregateRoot extends AggregateRoot
{
    protected CartItems $cartItems;

    public function __construct()
    {
        $this->cartItems = new CartItems($this);
    }

    public function addItem(
        string $cartItemUuid,
        Product $product,
        int $amount
    ): self {
        if (! $this->state->changesAllowed()) {
            throw new CartCannotBeChanged();
        }

        $this->cartItems->addItem(
            $cartItemUuid,
            $product,
            $amount,
        );

        return $this;
    }
```

```
    public function removeItem(
        CartItem $cartItem
    ): self {
        if (! $this->state->changesAllowed()) {
            throw new CartCannotBeChanged();
        }

        $this->cartItems->removeItem($cartItem);

        return $this;
    }

    // …
}
```

Again, let's make a few observations:

- the `$cartItems` variable needs to be protected in order for the package to be able to access it;
- for this example, we're keeping state checks in our aggregate root. You're allowed to create a dedicated state for individual partials as well if needed.



The final piece of the puzzle is exposing state between partials and aggregate roots. As we've discussed before, it's not desirable to expose an aggregate root's state to the outside, but it's perfectly fine to use it within its own internal aggregate. Unfortunately, PHP has no way of enforcing "friend classes" that can, for example, access protected properties or methods. That means that we'll need to provide public getters in order to read state between an aggregate root and its partials. We've already added such a method called `isEmpty()` on `CartItems`, and can use it in our aggregate root like so:

```
class CartAggregateRoot extends AggregateRoot
{
    // …

    public function checkout(
        CartCheckoutData $cartCheckoutData
    ): self {
        if ($this->cartItems->isEmpty()) {
            throw new CartIsEmpty();
        }

        // …
    }
}
```

If you want to pass state the other way around (from the aggregate root to a partial) you could either add getters on the aggregate root and remember to only use them internally, or you can pass an aggregate's state to its partials every time you're calling one of the partial methods. Personally, I prefer the second approach so that we can avoid public getters from being abused.

TESTING

Since aggregate partials are an internal implementation detail of the aggregate, your aggregate root tests will keep working just like before. You can still fake the aggregate root and observe recorded events like you're used to. This makes refactoring to partials extremely simple since you don't need to change any tests.

```php
/** @test */
public function can_add_item()
{
    CartAggregateRoot::fake(self::CART_UUID)
        ->given([
            $this->cartInitializedFactory->create(),
        ])
        ->when(function (CartAggregateRoot $cartAggregateRoot) {
            $cartAggregateRoot->addItem(
                self::CART_ITEM_UUID,
                $this->product,
                1
            );
        })
        ->assertRecorded([
            $this->cartItemAddedFactory->create(),
        ]);
}
```

It's also possible to test partials in isolation by using `AggregatePartial::fake()`. By doing so, you don't need to worry about setting up a real aggregate root and can use the partial as-is.

```
class CartItemsTest extends TestCase
{
    private const CART_ITEM_UUID = 'cart-item-uuid';

    /** @test */
    public function test_is_empty()
    {
        $cartItems = CartItems::fake();

        $this->assertTrue($cartItems->isEmpty());

        $product = ProductFactory::new()->create();

        $cartItems->addItem(
            self::CART_ITEM_UUID,
            $product,
            1
        );

        $this->assertFalse($cartItems->isEmpty());
    }
}
```

Faking partials can be useful if there are lots of little details within a partial's flow that would take lots of setup work if you'd test them on the aggregate root level. You'll still have to test the correct use of partials from the aggregate root's perspective though.

Partials are a useful technique in keeping aggregates maintainable. The way events are applied on them can reduce lots of boilerplate code, especially if you're using commands, which we'll cover soon. Remember that you shouldn't overcomplicate your design upfront. It's super simple to refactor an aggregate root to several partials throughout the project. Because we have such a simple and versatile solution, it allows our code to grow together with the project over time, while keeping it all maintainable.

Let's once more stress that partials are nothing more than a way of structuring code in order to keep it maintainable. You could also use traits, entities and value objects or standalone classes to keep your aggregate roots small. What I've found useful with partials is that they are also event-driven by design, making them easy to understand in an event-driven context.

CHAPTER 12

STATE MACHINES WITH AGGREGATE PARTIALS

We will bring the knowledge of the previous two chapters together and use aggregate partials to build a state machine. While the state pattern is useful in many scenarios, you might not like how it spreads state-related logic across several classes. Now that we know how partials work, we can create one dedicated to managing an aggregate's state instead.

We start by making a `CartStateMachine`; this is an partial class:

```
class CartStateMachine extends AggregatePartial
{
    private CartStatus $cartStatus;

    private CartLockStatus $lockStatus;

    public function __construct(AggregateRoot $aggregateRoot)
    {
        parent::__construct($aggregateRoot);

        $this->cartStatus = CartStatus::pending;
        $this->lockStatus = CartLockStatus::unlocked;
    }
}
```

Internally we keep track of the two relevant states we identified previously: `CartStatus` and `CartLockState`; this time they are simple enums again, since all functionality will be handled by our state machine:

```
enum CartStatus {
    case pending;
    case checkedOut;
    case failed;
    case paid;
}
```

```
enum CartLockStatus {
    case locked;
    case unlocked;
}
```

Next, we need a mechanism to move the state forward. When we used the state pattern, we had a `nextState()` method on each state class. By using partials, however, we can listen to aggregate events to change the state:

```
class CartStateMachine extends AggregatePartial
{
    // …

    public function applyCheckout(CartCheckedOut $event): void
    {
        $this->cartStatus = CartStatus::checkedOut;
    }

    public function applyLock(CartLocked $event): void
    {
        $this->lockStatus = CartLockStatus::locked;
    }

    public function applyUnlock(CartUnLocked $event): void
    {
        $this->lockStatus = CartLockStatus::locked;
    }

    public function applyFail(CartFailed $event): void
    {
        $this->cartStatus = $this->isLocked()
            ? CartStatus::failed()
            : CartStatus::pending();
    }
}
```

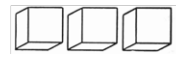
Next, we need some methods to read state from our state machine and they look like this:

```
class CartStateMachine extends AggregatePartial
{
    // …

    public function changesAllowed(): bool
    {
        return $this->cartState === CartState::pending
            && $this->lockState === CartLockState::unlocked;
    }

    public function isLocked(): bool
    {
        return $this->lockState === CartLockState::locked;
    }

    public function canFail(): bool
    {
        return $this->cartState === CartState::checkedOut;
    }
}
```

And finally, we hook everything up in our aggregate root:

```
class CartAggregateRoot extends AggregateRoot
{
    protected CartStateMachine $state;

    public function __construct()
    {
        $this->state = new CartStateMachine($this);

        // …
    }

    public function checkout(/* … */): self
    {
        if (! $this->state->changesAllowed()) {
            throw new CartCannotBeChanged();
        }

        // …
    }

    public function removeItem(/* … */): self
    {
        if (! $this->state->changesAllowed()) {
            throw new CartCannotBeChanged();
        }

        // …
    }
```

```
    public function addItem(/* … */): self
    {
        if (! $this->state->changesAllowed()) {
            throw new CartCannotBeChanged();
        }

        // …
    }

    public function lock(): self
    {
        if ($this->state->isLocked()) {
            throw new CartCannotBeLocked();
        }

        // …
    }

    public function unlock(): self
    {
        if (! $this->state->isLocked()) {
            throw new CartCannotBeUnlocked();
        }

        // …
    }

    public function fail(): self
    {
        if (! $this->state->canFail()) {
            throw new CartCannotFail();
        }

        // …
    }
}
```



We haven't done anything new compared to using the state pattern, but using a partial as a state machine has some advantages:

- it's conceptually easier to understand since everything related to states is kept in the same class;
- it's easier to test without dependencies; and
- it moves all state logic away from the aggregate root while still being able to listen to events.

TESTING

As mentioned before: testing state machines in isolation is very simple. You often have to test many state combinations, but each test will be only two or three lines of code. Here are a few examples:

```
/** @test */
public function changes_allowed_when_created()
{
    $state = CartStateMachine::fake();

    $this->assertTrue($state->changesAllowed());
}

/** @test */
public function changes_not_allowed_when_checked_out()
{
    $state = CartStateMachine::fake();

    $state->applyCheckout(
        CartCheckedOutFactory::new()->create()
    );

    $this->assertFalse($state->changesAllowed());
}
```

```php
/** @test */
public function failed_cart_that_is_locked_stays_failed()
{
    $state = CartStateMachine::fake();

    $state->applyLock(CartLockedFactory::new()->create());

    $state->applyFail(CartFailedFactory::new()->create());

    $this->assertTrue($state->isFailed());
}

/** @test */
public function
    failed_cart_that_is_unlocked_transitions_back_to_pending()
{
    $state = CartStateMachine::fake();

    $state->applyFail(CartFailedFactory::new()->create());

    $this->assertTrue($state->isPending());
}
```

Testing the state flow in isolation is very powerful. It's easy to cover all possible scenarios and be sure your state machine works as intended when it comes to state management. Remember that you'll also have to test correct usage of your state machine within your aggregate root using aggregate fakes, but you don't have to test all transition functionality on the aggregate anymore - that's already handled by the state's unit test.

We've now seen three ways to dealing with state: using a status enum, using the state pattern or modelling an aggregate partial as a state machine. I deliberately wanted to share all three with you since different problems require different solutions.

Keep in mind that state management is one of the most crucial parts of your application, so you should carefully think about which solution fits your project's needs the best. You can, of course, switch strategies between aggregates since there's no rule saying you should apply the same pattern everywhere.

What's more, you'll always be able to switch between state management strategies later if required easily. Since aggregate roots are reconstituted on the fly and because their state lives in memory, you don't have to worry about a migration path when switching from the enum approach to, for example, partials.

This in-memory flexibility is actually one of the benefits you'll come to love with event sourcing.

CHAPTER 13

THE COMMAND BUS

Up until this point, we have avoided a well-known paradigm in event-driven systems. The reason is that we simply didn't have a need for it, mostly because all chapters up until now assumed we're working in a single application with a linear program flow. In this chapter we'll look at what we've missed - both the benefits that brings and also the added difficulties. We're going to discuss **commands** and **command handlers**.

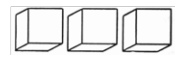
In a way, we've already used a concept similar to commands and handlers since the beginning of this book - we called them "actions". Both commands and handlers have the same goals:

- to move business functionality away from entry points like controllers or CLI commands; and
- visualise the *intent* of our code.

An action essentially encapsulates everything that's needed for *something* to happen in the application. From the outside — controllers or CLI applications — we're not interested in *how* an action is performed, rather we just want it to happen and return the correct result. As an example, it's easier to say "add an item to a cart" instead of "retrieve the cart aggregate root, add an item to it, persist it and then retrieve the projected results".

Commands serve the same purpose, but they differ to actions in that the command and its handler are split into separate classes. Just like events, commands are simple data objects, while their handler knows how to deal with them. To map a command to its handler, there's a **command bus** in between, just like the event bus.

Why the added complexity? This indirect approach offers the same advantages as with events and an event bus: it splits our system in more loosely coupled components that are easier to scale. Of course, your project might not need this additional flexibility, which is why I waited to bring it up until now. It's entirely possible that you'll never need a command bus that, for example, can handle commands asynchronously or dispatch them to other servers if you're building a distributed system.



Before further discussing the benefits and difficulties of a command bus, let's see what it looks like in practice. Here's a command that adds an item to a cart:

```
namespace Spatie\Shop\Cart\Commands;

use Spatie\Shop\Support\EventSourcing\Attributes\AggregateUuid;
use Spatie\Shop\Support\EventSourcing\Attributes\HandledBy;
use Spatie\Shop\Support\EventSourcing\Commands\Command;

#[HandledBy(CartAggregateRoot::class)]
class AddCartItem implements Command
{
    public function __construct(
        #[AggregateUuid] public string $cartUuid,
        public string $cartItemUuid,
        public Product $product,
        public int $amount,
    ) {
    }
}
```

Note that you can checkout the `command-bus` branch of the demo app to see it implemented for the cart aggregate.

You might have noticed that commands can be mapped automatically to an aggregate root using attributes. This essentially makes the aggregate method the command handler. This implementation was inspired by Axon, a popular event sourcing framework in the Java world. It's a shortcut for not having to manually write lots of handlers that always do the same thing: retrieving an aggregate, calling a method and persisting the changes.

Behind the scenes this command will be mapped to the `CartAggregateRoot`, using the `$cartUuid` as the identifier. It will search for a method on the aggregate root that handles this command based on its type.

Here we arrived at an important advantage of using a command bus. Commands also work with aggregate partials, which means that we don't need a manual mapping between the aggregate root methods and the partial methods anymore. Remember how we provided "proxy methods" on the aggregate root in the previous chapter? They looked like this:

```
class CartAggregateRoot extends AggregateRoot
{
    protected CartItems $cartItems;

    // …

    public function addItem(
        string $cartItemUuid,
        Product $product,
        int $amount
    ): self {
        // …
        $this->cartItems->addItem(
            $cartItemUuid,
            $product,
            $amount,
        );

        return $this;
    }

    public function removeItem(
        CartItem $cartItem
    ): self {
        // …
        $this->cartItems->removeItem($cartItem);

        return $this;
    }
}
```

The command bus provided with the shop package is smart enough to know that commands should also be passed to aggregate partials, meaning we don't need to provide those proxy methods anymore. Our `CartItems` partial would look like this:

```
class CartItems extends AggregatePartial
{
    // …

    public function addItem(
        AddCartItem $addCartItem
    ): self {
        // …
    }

    public function removeItem(
        RemoveCartItem $removeCartItem
    ): self {
        // …
    }
}
```

And `CartAggregateRoot` wouldn't need those `addItem()` and `removeItem()` anymore.

Whenever a command is dispatched, the command bus will automatically resolve the correct handler method based on its attributes and the type hint in the aggregate root (or one of its partials). So dispatching a command becomes as trivial as this:

```
class CartController
{
    public function addCartItem(
        Request $request,
        CommandBus $commandBus
    ): void {
        $commandBus->dispatch(new AddCartItem(
            cartUuid: /* … */,
            cartItemUuid: /* … */,
            product: /* … */,
            amount: /* … */,
        ));
    }
}
```

With the basics out of the way, there are a few more things we need to discuss.

First, there's always a one-to-one mapping between commands and their handlers. That's because a handler is allowed to return a result to indicate that the command was successfully handled or not. However, working with event-sourced aggregate roots as command handlers makes this a little more difficult since aggregate roots themselves shouldn't return

anything. For example, we won't be able to directly return a result like a projection; but we can throw exceptions to indicate that something went wrong. You already know about the indirect character of an event-driven design, so you know there are both benefits and downsides.

In practice, this means we'll have to resolve the expected result after running a command manually:

```
class CartController
{
    public function pay(
        Request $request,
        CommandBus $commandBus
    ): void {
        $paymentUuid = PaymentUuid::make();

        $commandBus->dispatch(new PayCart(
            cartUuid: /* … */,
            paymentUuid: $paymentUuid,
        ));

        $payment = Payment::find($paymentUuid);
    }
}
```

Also, keep in mind that side effects of commands can be queued (like projections or reactors, for example). That means we might not be able to immediately return a result like with did in the example above. For example, imagine that creating a payment involves communication with an external payment provider. This might take a couple of seconds, maybe even minutes.

Solving these kinds of issues require us to build our frontend with an async mindset. A somewhat naive solution could be to wait in the controller before returning:

```
class CartController
{
    public function pay(
        Request $request,
        CommandBus $commandBus
    ): void {
        $paymentUuid = PaymentUuid::make();

        $commandBus->dispatch(new PayCart(/* … */));

        while (! $payment = Payment::find($paymentUuid)) {
            continue;
        }

        return $payment;
    }
}
```

Though you could argue that this is a rather sketchy solution: what if the external payment provider is currently unavailable? Will we simply wait for a timeout error? That's not a user-friendly solution. The better solution, in this case, will depend on the business requirements, but most likely will involve one of two things:

- client-side polling combined with a user-friendly loading screen until the payment is processed; or
- using a form of async communication like email to let the client know their payment is ready and can be paid.

You can see how there isn't a "technically correct" answer to these questions, as all will depend on the business requirements. However, we *do* know that our controller shouldn't wait to return a result. Instead, we'll return the payment uuid so that the frontend can decide what to do with it:

```
class CartController
{
    public function pay(
        Request $request,
        CommandBus $commandBus
    ): void {
        $paymentUuid = PaymentUuid::make();

        $commandBus->dispatch(new PayCart(/* … */));

        return $paymentUuid;
    }
}
```

Another detail worth noting is that commands should always have a handler. If a command goes unhandled, we'll throw an exception. This is a safety measure for programmers, as the indirect flow of these kinds of systems might make it difficult to spot whether a command is actually handled or not.

With the basics out of the way, there are some more things to tell about using command buses. First, there's always a one-to-one mapping between commands and their handlers. That's because a handler is allowed to return a result to indicate that the command was handled or not. If we're working with event-sourced aggregate roots, we won't be

able to return a result like a projection directly; but we can throw exceptions to indicate that something went wrong.

Next, command busses also support **middleware**. Middleware allows us to hook into the command bus flow and add all kind of functionality. For example, you could log every command that's dispatched. Such a middleware would look something like this:

```
use Spatie\EventSourcing\Commands\Middleware;

class LogMiddleware implements Middleware
{
    public function handle(
        object $command,
        Closure $next
    ): mixed {
        $commandClass = $command::class;

        \Log::info("Command {$commandClass} was dispatched.");

        return $next($command);
    }
}
```

Adding a middleware to the command bus should be done when registering it in the dependency container:

```
class AppServiceProvider extends ServiceProvider
{
    // …

    public function register(): void
    {
        $this->app->singleton(CommandBus::class, function () {
            $commandBus = new CommandBus();

            $commandBus->middleware(new LogMiddleware());

            return $commandBus;
        });
    }
}
```

Another, more complex middleware is provided by the event sourcing package: the `RetryMiddleware`. This one will retry any failed command a given amount of times before permanently failing it. This is especially useful if there's a high chance of concurrent requests happening to the same aggregate root instance.

Imagine that our cart could be altered by multiple people at the same time. We need to make sure that operations on our cart aggregate root are performed in sync, otherwise, our chronological event log might become corrupted. So what happens if two people want to make a change to this aggregate root at the same time?

Well, one of the commands will fail because the package guards against these kinds of race conditions. But now what? You probably want to retry the failed command because chances are high it'll succeed, even with the changes made to the aggregate by the previous command. So we want to re-retrieve the aggregate root (to have the most up-to-date version) and dispatch the failed command once more.

This is what the `RetryMiddleware` does for you behind the scenes. It's a very simple approach to handling concurrency conflicts, but in many cases also effective.

TESTING

Just like events, commands themselves shouldn't be tested if they only contain data. Handlers of course should, and having a command bus allows some tests to be simplified.

Imagine a complex process that consists of several commands. You would test each command handler individually, but you'd also want to know whether all commands are triggered in the right order. With the command bus, you have one central place where all commands arrive, and thus a place where you can assert which commands were triggered and which were not.

You could, for example, have a testing command bus that not only dispatches them but also keeps a log of all the commands that passed in order. You could read that log (it could be an array of command names) and make assertions based on it.

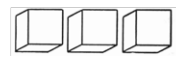
```
public function test_complex_process
{
    $this->commandBus->dispatch(new StartProcess(/* … */));

    $this->assertEquals(
        [
            StartProcess::class,
            SubCommandA::class,
            SubCommandB::class,
            SubCommandC::class,
            EndProcess::class,
        ],
        $this->commandBus->getLog(),
    );
}
```

It is possible that command buses are overkill for your project. If you're working in a monolithic project, there's little added value to gain. Also, keep in mind that there's a cost to using commands: there are more classes and more indirectness. For some projects, this cost is warranted, but not for all.

CHAPTER 14

CQRS

We've arrived at the chapter where we're going to discuss this peculiar term many have probably heard of: CQRS. CQRS stands for **command-query responsibility segregation**. It's a term that many think of as very complex, but we've actually been doing CQRS all along throughout this book.

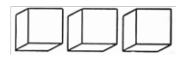


Greg Young first came up with the term CQRS and has often said that you can't do event sourcing without it (https://www.youtube.com/watch?v=JHGkaShoyNs). More precisely, in fact, he says that you *can* do CQRS without event sourcing, but not the other way around.

Let's look at its name for a moment, "command-query responsibility segregation". In other words: CQRS is about segregating (or separating) the responsibilities of commands and queries into two. We've seen that commands and actions are all about making changes to the system, while projections and event queries are about presenting data from the system to users. The "command" part of CQRS refers to the **write** side of our application - the code that makes decisions and persists data; while the "query" part refers to the **read** side - the code that interprets data and state in ways that makes it accessible to the user. Here's Fowler's definition:

> *At its heart is the notion that you can use a different model to update information than the model you use to read information.*

I find that using terms like “reading” and “writing” very much clarify what the pattern is about: separating code that writes data from code that reads data. CQRS claims there are benefits for doing so, so let’s take a closer look.



Considering everything we’ve discussed in this book up until now shows that Greg is right: you cannot create an event-sourced application without CQRS; separating writes (storing events) from reads (projections) is one of the core principles of an event-driven system.

And while there’s no choice about using CQRS or not in event-driven systems, it’s good to know the advantages it offers. Let’s compare CQRS to how Laravel models work: a model represents data from a database and also exposes a rich query builder API. Many Laravel developers will go one step further though and add affordance methods on models to make them easier to work with: `$cart→checkout(/* … */)` or `$cart->addItem(/* … */)`. While such an approach might work in small projects, it could quickly get out of hand. Model classes could:

- grow to hundreds of lines of code;
- combine both read and write operations;
- lead to complex relations and side effects between them, and
- result in performance issues because of nested relations, and more.

I began to appreciate the power of CQRS when I understood that there shouldn’t be a one-to-one mapping between the write model and the read model. On the write side, we have events that originate from one aggregate root, for example the events related to the cart; but we can have several *different* projections of those cart events, as well as projections spanning several different aggregates.

We already know about one projection: a table that keeps track of the cart and its items so that customers can see their cart via a web interface. But what if we also need an admin panel showing reports about how many carts were checked out per day, together with their total value? Should we aggregate all existing carts at runtime into a data set? No: we can make a new `CartAdminReport` projection or use an event query and replay all previous events on it.

What if you needed a very performant search index on all your products? An integration with, for example, Algolia or ElasticSearch could be another projection, especially for that goal. Such a projection doesn't live in your database, but still, it's one interpretation of the event stream and one read-model amongst many.

The best part is that everything that you want to add to the read side can be done without any breaking changes: the events that are the source of truth of our application have been stored all along. We're only making new interpretations of those events after the fact.

Another question that developers often struggle with is whether the write side is allowed to use data from the read side to make decisions on. For example: can the cart aggregate root use a projection to determine whether an operation is allowed or not?

As we've seen, the power of aggregate roots lies in that its internal state is reconstituted from events at runtime. But there can be cases where it feels like you're just copying the same data structure you already had as a read-model in order to build an aggregate root's internal state.

Some people claim that using read-models in the write side is never allowed. I'd say the answer is a little more nuanced, though. For example: if your projections are always a synchronous operation after events are dispatched, you know for sure that read-models can't contain outdated information. Synchronous projections aren't always possible though, but even then, there might be cases where it's better to deal with some inconsistency issues instead of trying to keep track of all state internally in the write side.

Greg Young says the same, by the way: there are cases where querying the read side from the write side is simply much easier; it doesn't make sense to get hung up on it if you know what you're doing.

It's questions like these that scare developers from using CQRS and event sourcing. They fear *the big list of rules* that comes with the pattern, a list that originated in large projects for specific use cases. Mind you: this list of rules is very useful in many cases, but you should carefully determine which should be used where and not just blindly follow them. Ross Tuck has written a great post on common misconceptions about commands and command buses on rosstuck.com, in his conclusion, he says:

> *But remember, the point isn't to cargo-cult [...], it's to understand the constraints and decisions that lead to [a] tactical choice.*

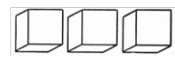
I hope you're as excited as I am at this point: everything we've discussed about event sourcing has led us to discover that we've been doing CQRS all along and that it's an incredibly powerful and flexible way to structure our code.

Of course, as with everything, CQRS isn't the solution to all problems. We've established this idea in the very beginning of this book as well: event sourcing isn't for all projects. But when you're dealing with a project that's large enough, with enough complex process flows, it's a life changer to be able to have these kinds of patterns to help guide you.

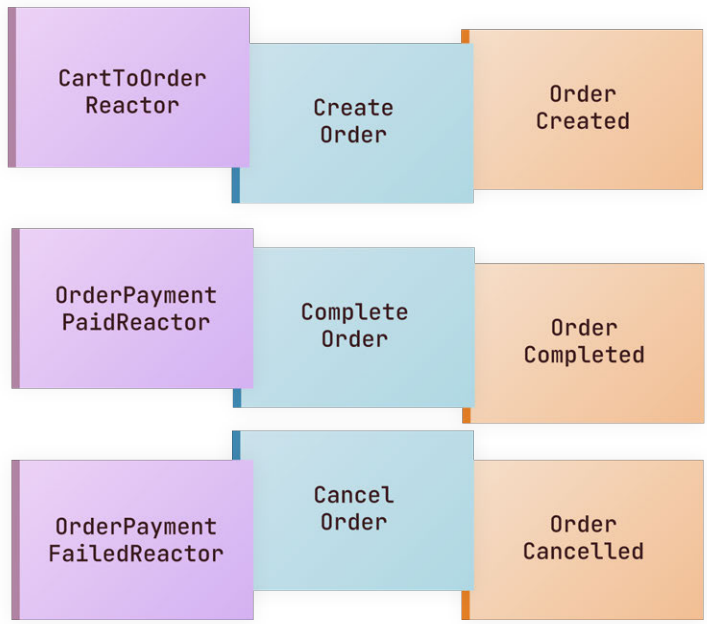
CHAPTER 15

SAGAS

Back in chapter 9, we discussed the concept of reactors to manage the program flow. I mentioned that reactors can be used to create so-called **sagas** or complex process flows. This chapter will be dedicated to such flows.



Let's start by looking at the order part of our event diagram. You can see everything related to an order is triggered automatically via reactors.

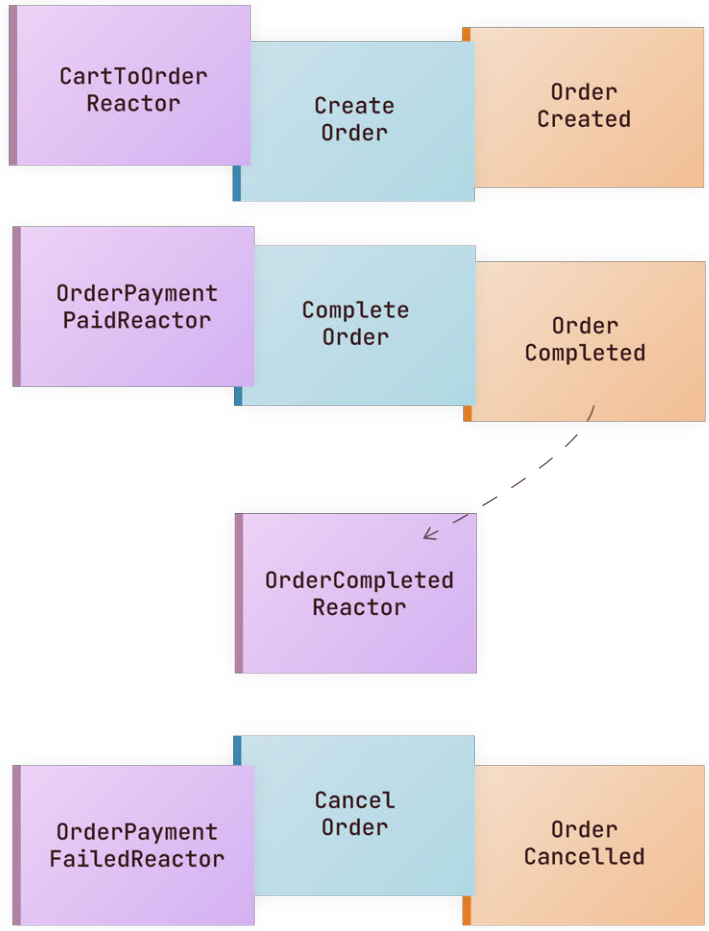


Yet, there's still something missing in this diagram: what should happen when an order completes? We should actually do two things: generate an invoice PDF and send it via mail to the customer.

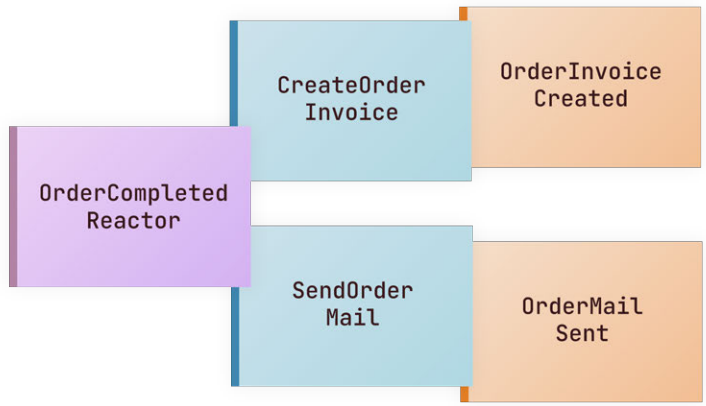
Since we can't send a mail before the PDF is generated, this process should be treated as "a whole". That poses an interesting question though: what if PDF generation is done asynchronously? Maybe there's even a dedicated service, independent of our main program. While "generating a PDF" and "sending a mail" is a single process, it turns out that this process itself is divided into several smaller pieces.

This is the kind of problem a saga aims to solve.

We'll start by adding an `OrderCompletedReactor`, this is the reactor that will oversee the process as a whole.



What this reactor should do is issue two commands: first create a PDF, next send the mail. In our diagram, it would look something like this:



Maybe you can already see where we're going with this: our reactor will listen, not only to the `OrderCompleted` event, but also to the `OrderInvoiceCreated` and `OrderMailSent` events: from the outside, these events represent one process, so it's OK to handle them in the same reactor.

Our reactor would look something like this:

```
class OrderCompletedReactor extends Reactor
{
    public function onOrderCompleted(
        OrderCompleted $event
    ): void {
        $order = Order::find($event->orderUuid);

        (new CreateOrderInvoice)($order);
    }

    public function onInvoiceCreated(
        OrderInvoiceCreated $event
    ): void {
        $order = Order::find($event->orderUuid);

        (new SendOrderMail)($order, $event->invoicePath);
    }

    public function onMailSent(
        OrderMailSent $event
    ): void {
        Log::info("Mail for order {$event->number} was sent.");
    }
}
```

As you can see, the basic idea of sagas isn't new. We're just structuring code a little differently. By doing so, we're getting a few advantages.

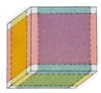
First, sagas can manage their internal state, just like aggregate roots. You could listen for events, store some of their data as class variables and make decisions based on that state later in the process.

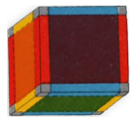
Second, you could introduce a way of serializing sagas and persisting them to disk. Imagine a process that could take days to complete and that sometimes needs a decision to be made to move along. You could attach a unique identifier to a saga and persist it so that you can retrieve it at a later point in time.

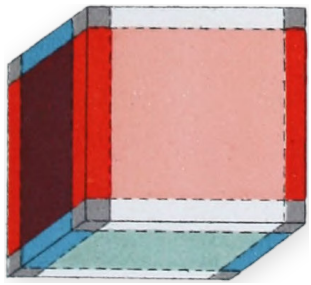
Finally, sagas could handle rollback functionality if something goes wrong somewhere in the process. If those errors are represented as events, you could hook into them and, for example, issue some commands to reset the database state back to where it was before the process started or maybe notify the right person about the failure.

Remember that there are ways to solve our initial example without using sagas: we could have a single reactor method that listens for the `OrderCompleted` event, and that will generate a PDF and send a mail in sequence. This solution is definitely enough if you're working in a centralized codebase.

Sagas do have their place in complex applications, especially if you're working in a distributed system where there's no other way but the handle events asynchronously. They are a good solution for a specific set of problems.







PART 3

CHALLENGES WITH EVENT SOURCING

We've covered the basics and how to use them in practice. Up till now, we've mainly focused on the good parts of event sourcing. We need to be realistic though: event sourcing comes with a set of challenges that don't exist in normal CRUD applications. We get lots of value in return, don't get me wrong, but we need to rationally address those challenges and find ways of dealing with them.

This part of the book is a collection of standalone topics that you can refer to whenever you're dealing with a specific problem. The beauty is that almost all problems with event sourcing are already solved. You just need to know the right solutions for them.

CHAPTER 16

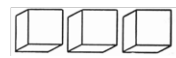
EVENT VERSIONING

One of the only certainties in software development is that most things will change, and almost nothing is permanent. The same goes for an event-sourced system. We've discovered that making changes to the read side of our application is trivial; it's even one of the large benefits of event-sourced systems. Making changes to the write side is another story though.

I want to clarify something upfront: the problem of dealing with change in an event-sourced system is already solved. In fact, there are many solutions to deal with change and versioning, and it's only a matter of picking the right one for the right circumstances. It is true that dealing with changes in an event sourced system is different compared to a CRUD application. Different doesn't mean better or worse though, and it's important to keep that in mind.

Some techniques in this chapter might seem counterintuitive if you're coming at the problem from a CRUD perspective, but remember what we gain from using event sourcing. Depending on the kind of project, event sourcing delivers much more than it costs.

First, we'll set the scene to discover what kind of problems we might run into.



Imagine your application has been in production for two years, and suddenly a major change to the shopping cart is required. Now, consider for example, that the new GDPR guidelines just came into effect, and all customers that don't have an account — in other words: guests who checked out a cart — must give explicit permission that their data may be used according to the privacy guidelines listed on the website.

In other words, we need to add an `$agreedWithPrivacyPolicy` property on our `CartCheckedOut` event:

```
class CartCheckedOut extends ShouldBeStored
{
    public function __construct(
        public string $cartUuid,
        public Carbon $date,
        public CustomerData $customerData,
        public bool $agreedWithPrivacyPolicy,
    ) {
    }
}
```

So we make the changes to our event, update the projectors and reactors to understand those changes accordingly, but find our code will break.

The issue lies with all previously stored events: older serialized versions of the `CartCheckedOut` class miss this `$agreedWithPrivacyPolicy` property. Trying to unserialize older events (which happens when reconstituting an aggregate root, for example) will fail because some data is missing.

Approaching this problem with a CRUD mindset, we know that we can use migrations as a solution. We could change the stored events in the database using a script when deploying our code to production, and we're on our way again.

But remember how we've always said that events are the only source of truth. If we're going to change that truth for past events, there will be a number of problems.

First, we can't trust our event log to be immutable anymore. Immutability is a beautiful thing in event sourcing. The fact that events are immutable and never changed is the cornerstone principle that allows us to make a number of assumptions throughout our codebase. The fact that we can throw away all our application state and trust it to be rebuilt from events to the same result, is one such guarantee. Another one is that we can incrementally add projections and know that they will be based on exactly the same data as previous projections.

The second problem with changing stored events isn't obvious in smaller projects but is a painful reality in distributed systems (we'll cover this topic in depth soon). There could be numerous consumers listening for the same event on different servers, and we don't have direct control over all of them. We can't just change previously recorded events and expect all of those consumers to know about that change, let alone expect them to handle that change correctly.

Finally, adding a single property is a relatively small change, but can you imagine what kind of work would need to be done if it was more complex than that? What would happen with properties being renamed or property data stored in another format?

The bottom line is: the `$agreedWithPrivacyPolicy` property didn't exist before deploying this new feature, and we shouldn't fight that idea. The old events will stay exactly as they were when originally dispatched.

Our original events still can't be reconstituted, so how do we solve the problem then?

We'll start with the simplest approach to deal with these kinds of changes: **versioning events**. If we assign a version to our event classes, and bump that version once we made changes to an event, we can use that version number to transform old events to new ones on the fly. This practice of transforming is also called **upcasting**: we're creating newer versions of our old events in memory.

In practice this means that our `CartCheckedOut` event would look like this after adding `$agreedWithPrivacyPolicy`:

```
use Spatie\EventSourcing\Attributes\EventSerializer;
use Spatie\EventSourcing\Attributes\EventVersion;

#[
    EventVersion(2),
    EventSerializer(CartCheckedOutSerializer::class),
]
class CartCheckedOut extends ShouldBeStored
{
    public function __construct(
        public string $cartUuid,
        public Carbon $date,
        public CustomerData $customerData,
        public bool $agreedWithPrivacyPolicy,
    ) {
    }
}
```

By default, all events are stored with version number `1`; you can override that value by using the `EventVersion` attribute. If we change the version to `2`, all `CartCheckedOut` events dispatched after this changed will be marked as version `2`.

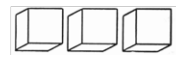
Marking an event with another version isn't enough though. The second piece of the puzzle is to provide a custom serializer for this event, one that knows how to deal with different versions. We can use the `EventSerializer` attribute to configure it.

Such a serializer would look like this:

```
use Spatie\EventSourcing\EventSerializers\JsonEventSerializer;

class CartCheckedOutSerializer extends JsonEventSerializer
{
    public function deserialize(
        string $eventClass,
        string $json,
        int $version,
        string $metadata = null
    ): CartCheckedOut {
        $data = json_decode($json, true);

        if ($version === 1) {
            $data['agreedWithPrivacyPolicy'] = false;
        }

        return new CartCheckedOut(...$data);
    }
}
```

When deserializing an event - meaning converting it from JSON to an actual class - we check the version that was stored with it. If in our case, it's version 1, we know it's an old payload and should manually provide a conversion.

Upcasting can feel like we're supporting legacy code indefinitely and that we have to write code to deal with that explicitly. Remember though, the benefits we get from ensuring our events are the only source of truth: they outweigh the maintenance cost associated with versioning.



A variation on upcasting is what Greg Young calls **event negotiation**. The idea is that stored events can be represented as several versions, depending on who's consuming them. This pattern becomes especially useful when you're dealing with distributed systems and when there are several event consumers that you don't directly control.

In practice it means that the `CartCheckedOut` would be able to be retrieved as version 1 *and* version 2. Imagine there's another service listening to the events your system is emitting (the details of how those events are emitted don't matter for now, you can assume there's some kind of HTTP layer between those two services.) Now imagine we, the producers of events, want our system to move forward, but don't want to wait until all consumers have caught up.

We could provide a mechanism for consumers to specify which version of events they are compatible with, and provide them the payload in the correct format. This approach means that you'd need to be able to convert events between all possible versions you're going to support. It's definitely not an easy task.

You can feel we're venturing into the microservices world. You already know that we won't cover microservices in detail, but we'll spend a dedicated chapter on the general ideas. We'll leave event negotiation here for what it is, but if you want to learn more about it, I'd recommend reading Greg Young's book called "Versioning in an Event Sourced System".

There are some cases where you really need a way of converting old events to new ones. If you can guarantee that all consumers of those events are properly updated as well (projectors, reactors, event queries, aggregate roots, etc.), you can consider this approach.

The advantage of migrated events is that you don't have to deal with legacy code like with upcasting, but the downside is that you'll need a way of migrating your events. Such migrations shouldn't be done with SQL queries. After all, you not only need to update your events, but also replay all related projectors, which can take a lot of time — hours, maybe even days.

We'll need a way to automate our migration process and keep our original application running while it's rebuilding its state with the new changes. You can feel where we're going with this: migrating will be a multi-step process.

First, we'll need to migrate the events themselves. I've already mentioned the concept of event streams early in this book, and it's this mechanism that we can use to our advantage. You can think of an event stream as a collection of events that belong together. A common example are the events related to an aggregate root: they form a stream based on the aggregate's UUID. We can use those streams to copy events from old streams to new streams, transforming them along the way to their new format. When all the work is done, we can remove the old stream and use the new one. This technique is aptly named **copy-replace**.

Since a stream always represents a collection of events that belong together, we should be able to remove a stream as a whole safely. For example, you shouldn't be allowed to remove a `CartItemAdded` event from a stream. That would result in an incorrect cart at the end, but you could remove the stream as a whole, essentially acting as if the cart never existed.

We start by marking old events as belonging to a stream called “v1”. If you know at the start of your application that these kinds of migrations will certainly happen in the future, it might be a good idea to tag all events from the start. A stream tag is nothing more than a meta property on every event, it’s a category we can filter on.

Next, everything needs to be in place for our projectors to handle the new type of events. If high availability is crucial in your system, or if you know rebuilding will take more than a couple of minutes, you could provide a separate version of projectors to deal with the new events and keep the old ones for now as well.

When everything is in place, we can deploy our changes and nothing actually alters. Production events are still tagged on the v1 stream for now, and our old projectors still work as before.

Now it’s time for the actual migration. As long as new events are coming in the v1 stream, we’ll keep running our migration script. This script will take all events from the v1 stream, transform them to new events, and dispatch them *again* on a new stream called v2. We must ensure that only our newly added projectors handle events from this stream, and not the existing reactors or projectors.

When our script has handled all events, we’ll finally make the switch so that new events are now dispatched on the v2 stream, and make sure that all reactors and projectors now work with the v2 stream instead of the old one.

In the end, we can delete the old events since they have been moved and replayed, all while the old system was still running.

This is by no means an easy task to do and in fact, Greg Young even calls it “the nuclear option of versioning”. It’s a powerful pattern, but you need a very good reason to even consider it. Also, note that the event-sourcing package we’re using doesn’t block you from doing this, but it also doesn’t provide a framework to help you with it. If you’re doing these kinds of

migrations, you'll have to write a migration script tailored to your needs.

With all of that being said, the most important thing to keep in mind is that it's better to prevent than to cure when it comes to event versioning.

Sometimes it's better to create new events and new projectors instead of trying to version or migrate events to new streams. If there are too many changes between the old and new version of a specific event, it could be an indicator that you're better off leaving the old events and their handlers and start with a clean(er) slate.

Also, try to keep events small. It's better to have several specific events like `ProductNameChanged` and `ProductedInventoryAdded` instead of `ProductUpdated`. The more ground covered by an event, the more likely you'll have to deal with complex versioning problems in the future.

Finally, keep relevant data *on* events, even when it seems like that data will never change. For example, shipping costs might always have a 6% tax rate, but it's still valuable to store that data with the event. Imagine if that rule changed somewhere in the future and you were doing all tax calculations in projectors themselves; you wouldn't be able to correctly rebuild your state if the old tax rate should be applied to old orders and the new rate to new ones.

A common pitfall you need to watch out for in those cases is that projectors start to have time-specific conditions:

```
class CartProjector
{
    public function onCheckedOut(CartCheckedOut $event): void
    {
        $shippingVatPercentage = $event->created_at > '2020-01-01'
            ? 0.12
            : 0.06;

        $totalShippingCost = $event->shippingCost
            * $shippingVatPercentage;

        // …
    }
}
```

You're going down a very deep rabbit hole if you start to do time-sensitive projections. It would have been much easier to have avoided this versioning problem altogether and store the VAT percentage with the event:

```
class CartProjector
{
    public function onCheckedOut(CartCheckedOut $event): void
    {
        $totalShippingCost = $event->shippingCost
            * $event->shippingVatPercentage;

        // …
    }
}
```

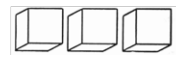
Chances are you won’t be able to avoid event versioning completely. Keep calm and know about the tools available to tackle the problem. Like I said at the start of this chapter: the problem of event versioning is already solved. You just need to identify the right solution for your specific situation.

CHAPTER 17

SNAPSHOTTING

A problem you should rarely deal with (though worth discussing) is **event snapshotting**. In some cases, aggregate roots become performance bottlenecks because of the number of events they have to load in memory. If the event stream associated with an aggregate root becomes too large, your code might be slowed down by it.

I want to make clear upfront that snapshotting should be your last resort when it comes to solving performance problems; there are other ways to deal with these issues without relying on snapshots. In fact, those solutions are probably always preferable, so let's cover them first.



To start, we should determine what "an excessive number of events" is within the context of a single aggregate root. A few hundred events is fine; if we're speaking of thousands, we should start to worry.

Next, we should design our aggregates in a way that they won't accumulate an infinite amount of events over time. We should design them so that they represent a finite process. A cart is the perfect example: it's a process with a clear start and end; after it's checked out, it won't grow any further.

Moreover, it's reasonable to assume that a cart will never accumulate more than a few hundred events during its life cycle, so we're safe to say our cart won't give us any performance problems. If, on the other hand, we're modelling a log of all carts as an aggregate root, things might get ugly. Above all, this should be a warning sign to reconsider the architecture of that part of your application. Also, keep in mind that aggregate partials are still part of the aggregate and that splitting such complex aggregate roots would face the same performance issues.

Finally, we should be wary not to do any expensive operations when applying events and reconstituting the aggregate root. Bottlenecks could go from expensive mathematical operations to handling I/O — which should definitely be avoided at all costs in aggregate roots! Aim to move those parts outside the aggregate root and only keep simple data in it.

If all else fails, there's still the option of snapshotting. The idea isn't difficult: take a snapshot of an aggregate root's state and store it in the database. The next time an aggregate root is retrieved, it'll start from that snapshot and only apply events that happened after the snapshot was taken.

Snapshotting, and thus serializing an aggregate root, comes with the same challenges as serializing an event (or any kind of class): you can't make changes to your current implementation without taking all previously-stored versions into account. This results in problems like aggregate root versioning, which can become complex over time. While previously only our events were the source of truth, we now also need to carefully think about making changes to our aggregate roots.

In all fairness, aggregate roots are a little less complex compared to events when it comes to versioning: you could always throw away an old snapshot, reconstitute the aggregate root from scratch and create a new snapshot. This means you'd need a dedicated rebuild step during deployments for snapshotted aggregate roots, which in turn adds a new layer of complexity.

All that to say that snapshotting is rarely the solution to performance problems. Still, for completeness sake, it's worth mentioning that it is an option.

So with all those caveats and disclaimers, let's look at how to use snapshots in practice. Taking a snapshot of an aggregate root is as easy as calling one method on it:

```
$cartAggregateRoot = CartAggregateRoot::retrieve($cartUuid);

$cartAggregateRoot->snapshot();
```

When and where you're taking snapshots is up to you. You can make daily snapshots of all aggregate roots as a cron job, or you could set up very specific triggers for individual aggregate roots. Your strategy should be guided by the project's needs.

There are two methods you can implement on your aggregate root to deal with the versioning issue: `getState()` and `useState()`:

```
class CartAggregateRoot extends AggregateRoot
{
    protected function getState(): array
    {
        $class = new ReflectionClass($this);

        return collect($class->getProperties())
            ->mapWithKeys(fn (ReflectionProperty $property) => [
                $property->getName() => $this->{$property->getName()},
            ])
            ->toArray();
    }

    protected function useState(array $state): void
    {
        foreach ($state as $key => $value) {
            $this->{$key} = $value;
        }
    }
}
```

By implementing these methods, you can make changes to the payload when serializing and deserializing aggregate roots, just like we're used to when versioning events.

To summarize: snapshots themselves aren't difficult to use, but rarely are they the best solution for performance problems. It's good to know they are available and how they work, but you'll rarely need them.

CHAPTER 18

MICROSERVICES

Throughout this book, we've approach event sourcing mainly from a single application's point of view. This is because the majestic monolith approach is still the most popular within the PHP and Laravel community, but also because microservices and distributed systems come with a whole set of problems that are beyond the scope of this book.

Thanks to its indirect and decoupled characteristics though, event sourcing is an excellent way of modelling microservices. Skipping the topic altogether would mean missing out on experiencing the strength of event sourcing within a distributed system, so we will spend one chapter discussing some high-level ideas.

What makes event sourcing and microservices such a good match is that both focus on being decoupled and scalable. A microservice architecture consists of several standalone services, each living in their isolated bubble, communicating with each other via messages .HTTP is a common way of doing so). These services are small applications with their own database, their own code (often using different languages between them), and their own internal state.

Replace "HTTP messages" with "events", and you start to see how both paradigms perfectly fit together. Microservices often have a central message bus that they use to send messages to, and receive notifications from, each other. Exactly how we described our event bus.

Since communication between services happens over the wire (using HTTP or any other kind of protocol), we'll need ways to serialize messages, just like we need to serialize events to store them.

As well as that, event sourcing not only offers a way of indirect communication, but it also provides a whole set of patterns for dealing with managing state based on such communication.

You can see why the two are such a great match.

It's not all fun and games: microservices add a lot of challenges for programmers and devops to deal with. In fact, many well-known software architects like Eric Evans and Greg Young warn and talk about the complexity of microservices.

Let's take one example to illustrate the kind of complexity you have to deal with.

We'll use the product inventory system we've been using in the examples, and we'll imagine we're Amazon for a moment. It's impossible to have one central place serving all Amazon's global traffic to manage inventory, so we *need* to scale horizontally across the globe.

What happens when a user adds an item to a cart or checks it out, and the product inventory must be updated? The inventory service closest by will handle that request, but it also needs to be kept in sync with other services. Remember: each node has its own database to ensure there's no bottleneck. Now imagine another customer adding the same item at exactly the same point in time but on another node. In reality, the item was already sold and out of stock because of the purchase on node A, but node B wasn't aware of those changes yet.

How will you deal with such conflicts? This problem is called the problem of **eventual consistency**: if we decide to scale our system horizontally to improve availability, it means we're paying the price with consistency. Making sure everything is consistent at all times would mean introducing a central service managing the only source of truth, and thus creating a new bottleneck.

You either choose between availability or consistency - you can't have both.

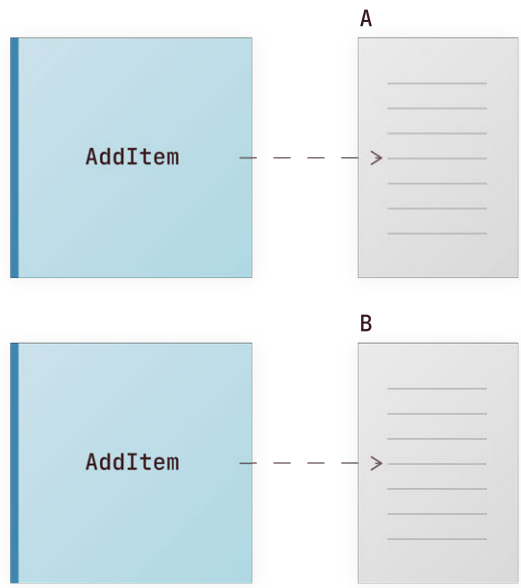
Greg Young describes a great way of dealing with this problem: he says to embrace it and deal with the consequences instead of fighting it.

Let's look at the real world for a moment. Imagine there are two physical shops using a shared warehouse, and two customers order the same product at exactly the same time. How would such a problem be resolved? Once the problem is detected by either store manager A or store manager B, one customer's order is cancelled or delayed *after* the fact.

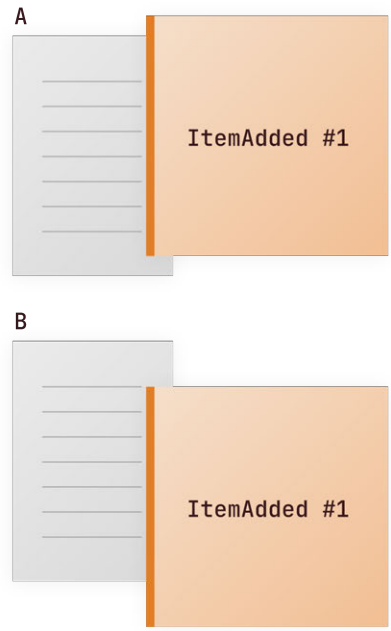
Greg Young says that the key in dealing with eventual consistency isn't to prevent it, but to be able to detect it in time, and to deal with it when it occurs, just like we'd do in real life.

With event sourcing, we have a solution at hand: detect *after* the fact whether two conflicting events occurred, and have a separate flow in place for when that happens. If a conflict happens once a month, it's probably not worth building an automated flow to handle it. If, on the other hand, it happens 50 times every second, it's crucial to handle it automatically. The great part is, event sourcing gives us all the building blocks we need to be able to detect such conflicts, and to handle them accordingly.

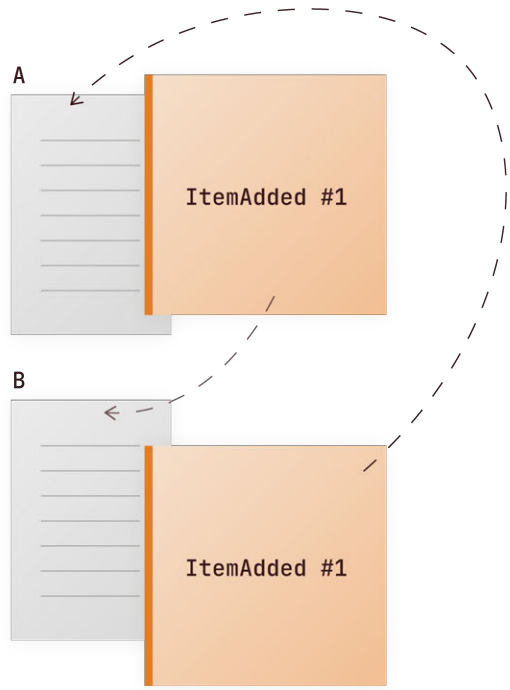
Let’s continue with our example. There are two inventory nodes, each with their own copy of the product inventory. When an `ItemAdded` event comes into node A, it will be stored, and the customer will be told everything is OK. At the same time, an `ItemAdded` event for the other customer with the same product arrives at node B, which doesn’t know yet about the changes that happened in node A.



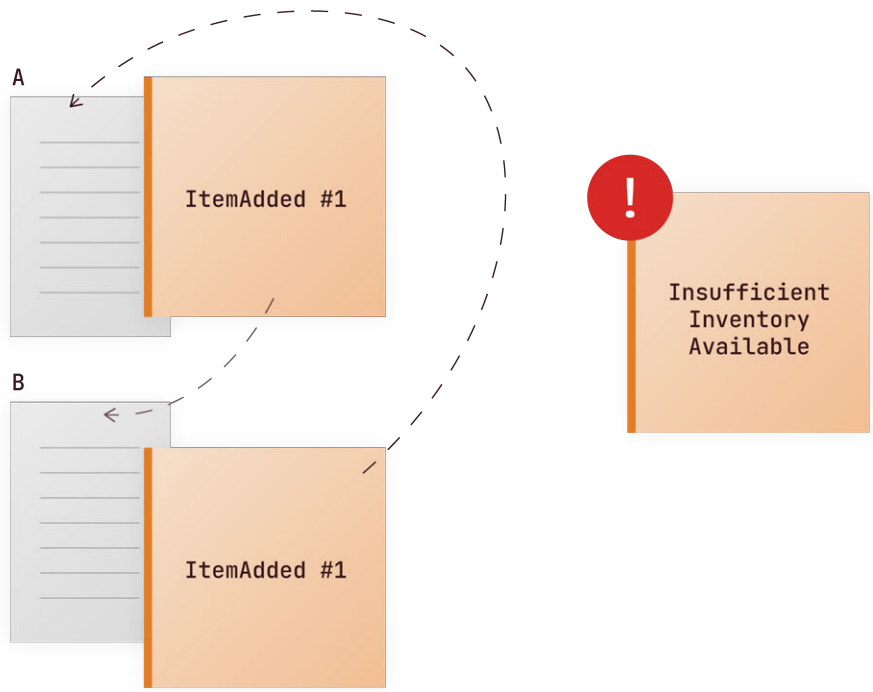
Now, each recorded event is given a sequence number: 1, 2, 3, 4, ... Whenever an event is recorded by a node, this sequence number increases by one.



Both nodes sync events and their sequence numbers between each other, but not exactly at the time the customer adds their item. Adding items should be a quick action, syncing events can be handled as a background task. When these two nodes sync their events though, they'll notice a conflict in sequencing: there were two events, on separate nodes, that didn't know about the full state of the other node when they were recorded. This means there's a *potential* problem.



If there's enough stock available to handle both events, then nothing happens. The events are synced between the nodes and all is fine. If, however, there is a mismatch that results in a negative amount of stock, we probably *should* do something. We dispatch a new event to indicate that a conflict happened, and have a separate handler for it. This event could be called `InsufficientInventoryAvailable`, for example.



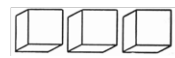
There would be a reactor listening for this event, and it would know how to handle such a conflict. Maybe one user should receive a cancellation mail, and maybe one order should be postponed; it all depends on what the business expects.

You can imagine the complexity of dealing with these kinds of problems in real-life applications. Luckily event sourcing is a great tool to help support you in handling them. If you're interested in reading more about microservices, I can recommend Matthias Noback's book called "Microservices for Everyone". We'll leave it at this though, since other microservice topics are outside the scope of this book.

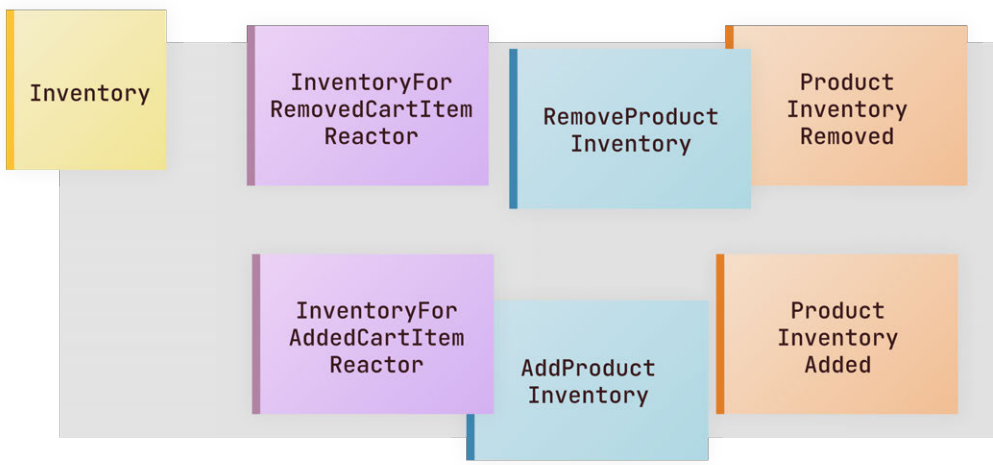
CHAPTER 19

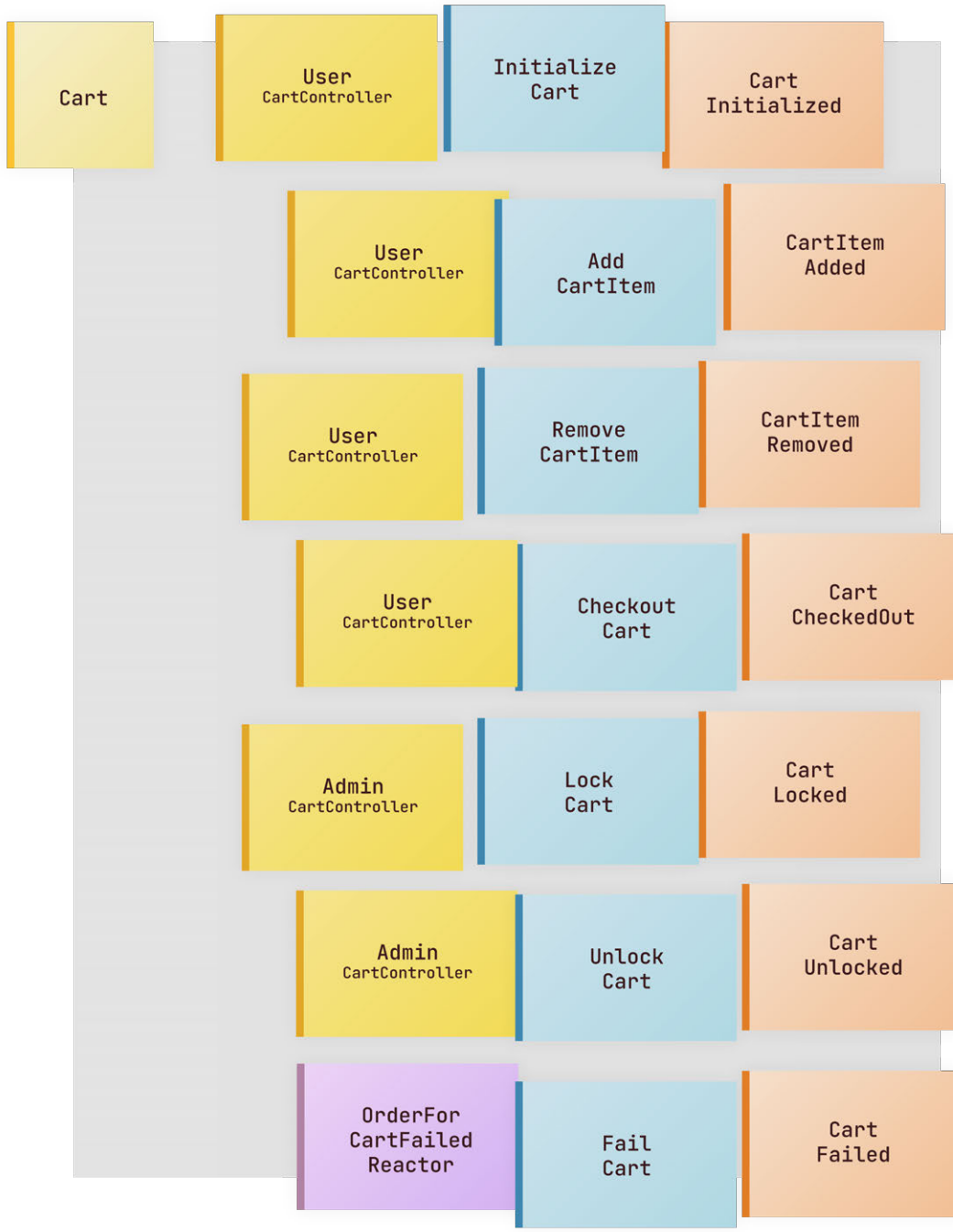
PARTIAL EVENT SOURCING

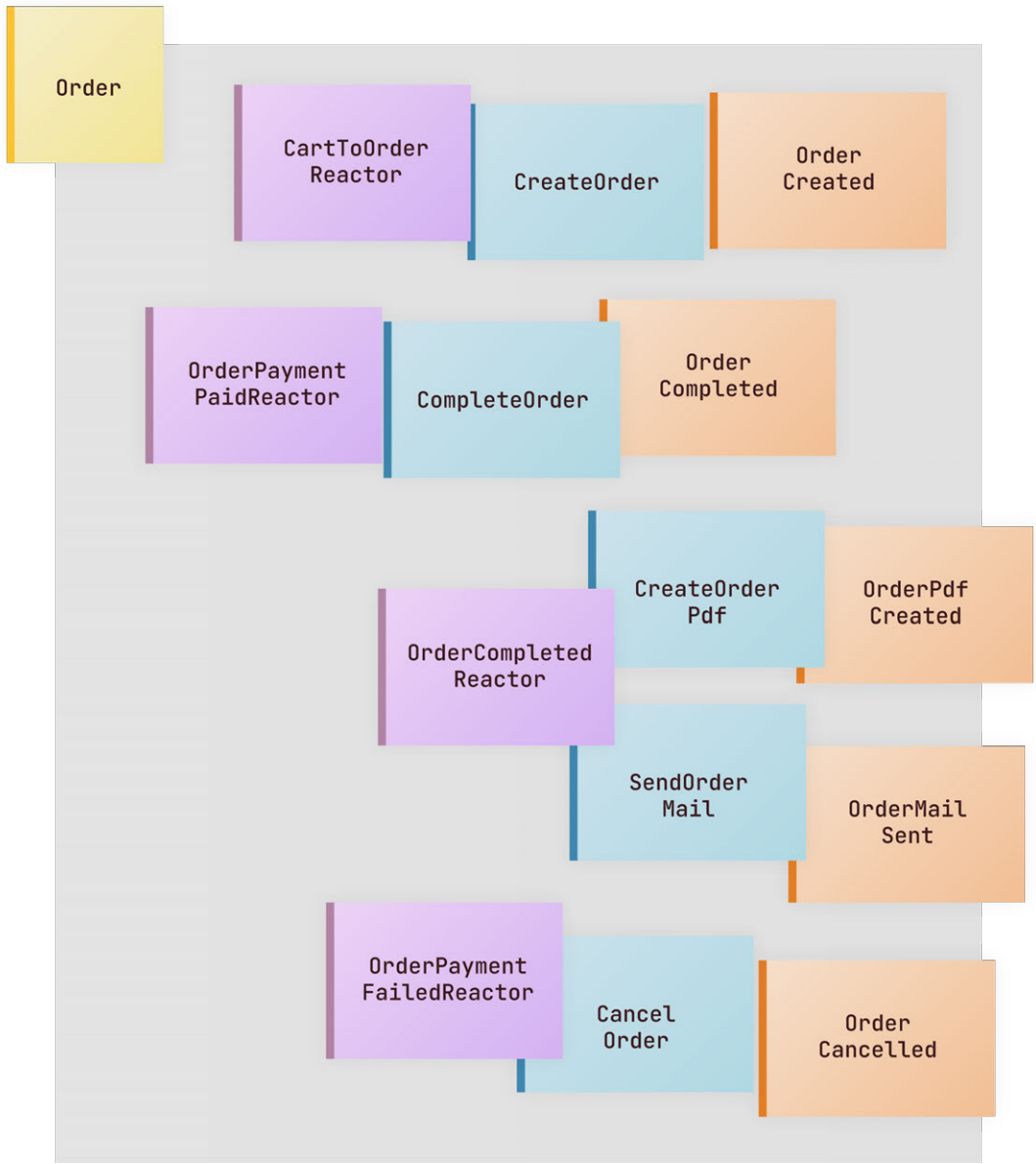
Throughout this book, we've always made the assumption that our whole system is event-sourced, and we've never considered otherwise because this book is about event sourcing, after all. Nevertheless, it's an important issue to address: is event sourcing your whole project a good idea? What are the alternatives?

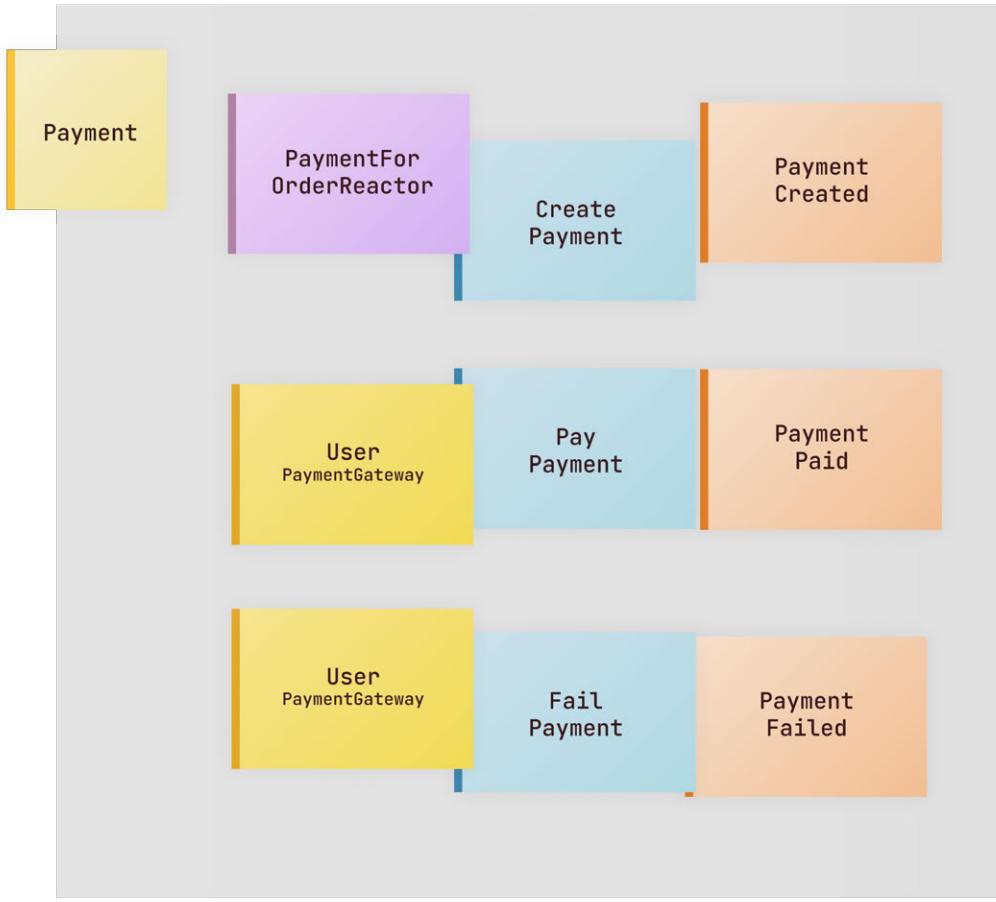


Let's revisit our event flow diagram and add a few more sticky notes to name our aggregates:




Cart
User
CartController
Initialize
Cart
Cart
Initialized
User
CartController
Add
CartItem
CartItem
Added
User
CartController
Remove
CartItem
CartItem
Removed
User
CartController
Checkout
Cart
Cart
CheckedOut
Admin
CartController
Lock
Cart
Cart
Locked
Admin
CartController
Unlock
Cart
Cart
Unlocked
OrderFor
CartFailed
Reactor
Fail
Cart
Cart
Failed


Order
CartToOrder Reactor
CreateOrder
Order Created
OrderPayment PaidReactor
CompleteOrder
Order Completed
CreateOrder Pdf
OrderPdf Created
OrderCompleted Reactor
SendOrder Mail
OrderMail Sent
OrderPayment FailedReactor
Cancel Order
Order Cancelled



Note that I removed the arrows between events and reactors for now because they add more noise and aren't important in this chapter.

Have you considered what's missing from this diagram? There's at least one important thing I can think about: a `Product` aggregate. Should that aggregate use event sourcing as well?

The answer to that question will depend on the scope of your project and its requirements. I was working on a project once where the client didn't need all the benefits event sourcing gave when it came to products. Knowing there's a significant overhead to event sourcing when it's not actually necessary, we carefully considered alternatives.

Ideally, we'd want only part of our system to be event-sourced, and the other part to be a simple CRUD.

That poses an issue:

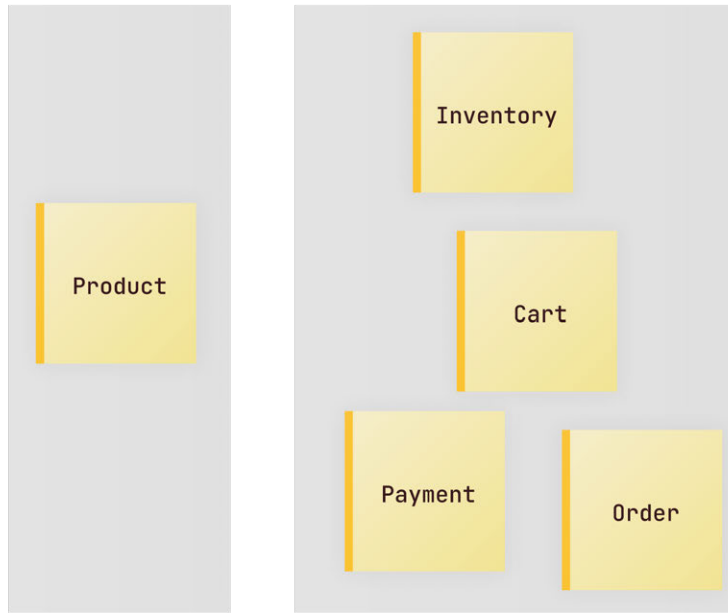
If products weren't event-sourced, but the inventory management and the shopping cart are, what would replays of our system look like? What happens if we throw away all cart-related projections and want to rebuild them, but there's a cart item referring to an old product that has since been deleted? How would we refer back to data that doesn't exist anymore in the CRUD part of our application?

The solution is actually right before us; it's the one rule we've been repeating over and over again: events are the *only* source of truth.

If we want to use data from a non-event sourced part of our application, we can't trust that data to exist forever. So we copy data from our CRUD system to our event-sourced system using events. At first hand, this might seem like a lot of data duplication: the `CartItemAdded` event now not only contains a reference to the product, but also its name, price, description and everything that might be relevant if a product were to be removed in the future. Let's consider what we get in return though:

- we can use event sourcing only in parts of our application where it makes sense;
- other parts where CRUD is sufficient don't need any overhead that comes with event sourcing;
- both can be combined in a way that ensures our code will keep working, even when we rebuild the event-sourced part.

Take a look at this diagram:



Maybe you've already noticed the parallels, but we've essentially applied a microservice mindset to what's actually a monolithic system. We can see a clear boundary between the CRUD and event-sourced part, and the only way of communication between them is by using events.

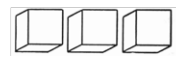
Thinking with a microservice mindset (even in a monolithic project) will be an advantage in larger projects. Greg Young warns about the dangers of trying to event source everything within a project: there's a significant overhead that doesn't always offer enough benefits, and you need to be careful about the added cost of using event sourcing. Projects that apply event sourcing everywhere without giving it a second thought often fail because of the unnecessary complexity. Greg says event sourcing isn't a framework but rather a pattern you should apply only in those parts of your system where it makes sense.

Sure, that means there's some additional overhead to managing the communication between event-sourced and CRUD parts, but it's a price worth paying to keep your system sustainable in the long run.

CHAPTER 20

DEPLOYMENT STRATEGIES

A part of event sourcing that might seem daunting at first is dealing with actually deploying code to production. We've already touched this topic a little bit when discussing event versioning, but there are some more things to be said.



First of all, event sourcing doesn't make *everything* more difficult when it comes to deploying changes to production. In fact, some parts become significantly easier.

Consider the write model for a moment. Everything that happens before events are actually stored: commands or actions, aggregate roots and their state management or reactors. All code in this layer (most likely representing the better part of our business logic) can be changed and deployed without any significant problems. So long as we're able to interpret the original events, we can change logic about inner state management in the aggregate root or rewire commands whatever way we want.

On the other hand, making changes to events or projections do require careful thought about a migration process.

Event sourcing makes it easier to reason about migrations because there's such a clean boundary, and we're always guaranteed that events are the only source of truth. Compare that to the CRUD approach and imagine there's an important change to the state flow. Chances are we'll need to write actual database migrations because state is (in part) determined by data stored in a CRUD model's table.

Still, there's no denying: when it comes to events or projections, migrations can be quite a bit more difficult compared to CRUD. Consider a system that's been running in production for several years, with millions of events, and some projections have to be rebuilt from scratch. Such a migration could potentially take hours, sometimes even days to finish.

Those cases require a more elaborate approach than simply putting your application offline as long as migrations are running.

We've already discussed one robust solution before: using a copy-replace strategy to rewrite events into a new stream with new projectors, while keeping the application running. Another approach could be to spin up a copy of the whole application, do all the migrations on that one, and send new events to both "nodes" while the migration is in progress. You can see why a microservice architecture might be useful here. If your application is already built to be horizontally scalable, there's little to no overhead in using such an approach.

Before considering those complex solutions though, maybe it's already good enough to carefully craft a migration path. If you know that a projector's only interested in a specific set of events, you could filter them before replaying. If the change only applies to events created in the last year, you could throw away all projections for that period and replay only the events within that year. By carefully thinking about what data to migrate, you can significantly reduce the migration cost and time.

Frank de Jonge maintains EventSauce, a framework-agnostic event sourcing library. He shares some interesting thoughts about rebuilding projections on eventsauce.io:

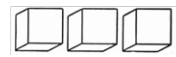
> *Rebuilding projections is one of the more complicated subjects in event sourcing. EventSauce takes an uncommon approach to tackle this problem: it does not tackle the problem.*
>
> *Generic rebuild tooling is very complex and imposes some additional constraints. Even then, generic tooling will probably cover about 80% of the use-cases.*

Frank is giving the most valuable piece of advice when it comes to dealing with change in event-sourced systems: don't look for a generic solution because there isn't one. You'll save more time by carefully laying out a migration path that fits your needs instead of being pushed into a corner by a framework.

In practice, this could be as simple a script like this one. In it, we first clean the projection tables, next retrieve the relevant events, and then replay them on the projectors:

```php
class CartReplay
{
    public function __invoke(): void
    {
        DB::table((new CartItem())->getTable())->delete();
        DB::table((new Cart())->getTable())->delete();

        StoredEvent::query()
            ->whereIn('event', [
                CartInitialized::class,
                CartItemAdded::class,
                CartItemRemoved::class,
                CartCheckedOut::class,
                CartFailed::class,
                CartPaid::class,
                ShippingCostsAdded::class,
                ShippingCostsRemoved::class,
                CouponUsed::class,
                CouponRemoved::class,
            ])
            ->orderBy('created_at')
            ->chunk(1000, function (Collection $data): void {
                foreach ([
                    app(CartItemProjector::class),
                    app(CartProjector::class),
                ] as $projector) {
                    $data->each(
                        fn (StoredEvent $storedEvent) =>
                            $projector->apply($storedEvent)
                    );
                }
            });
    }
}
```

I find that such custom migration scripts aren't difficult to write and offer you much more flexibility to do exactly what you want to, without any overhead.



When it comes to deployment strategies with distributed systems, I once again would recommend Matthias Noback's book "Microservices for Everyone". Taking a deep dive into this topic would be out of the scope of this book, but Matthias has lots of good things to say about microservices in general and uses them with an event-sourced architecture.

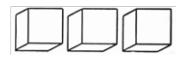
My recommendation is to be wary not to over-engineer your deployment process, as the easiest solution is, in many cases, the best.

CHAPTER 21

NOTES ON EVENT STORMING

One of the key building blocks in learning about event sourcing and building event-sourced systems was our event flow diagram. It purposefully grew throughout this book. My goal was to mimic some of the characteristics of real-world event storming: incremental discovery over time.

There are some notes to be made about how we used event storming though, and I want to spend this chapter sharing them.



We used event storming from a technical point of view: we already knew what solution we were working towards and what kind of architecture and patterns we'd use.

The original goal of event storming is to not think about technical implementations, but only about the business processes. Ideally, you'd gather with all stakeholders in a room and spend a day or several mapping out everything that's relevant to the business and its stakeholders.

It wasn't the intention of Alberto Brandolini, the inventor of event storming, to have so many similarities between his diagrams and event-sourced systems; that just happened to be a side effect of "thinking with events".

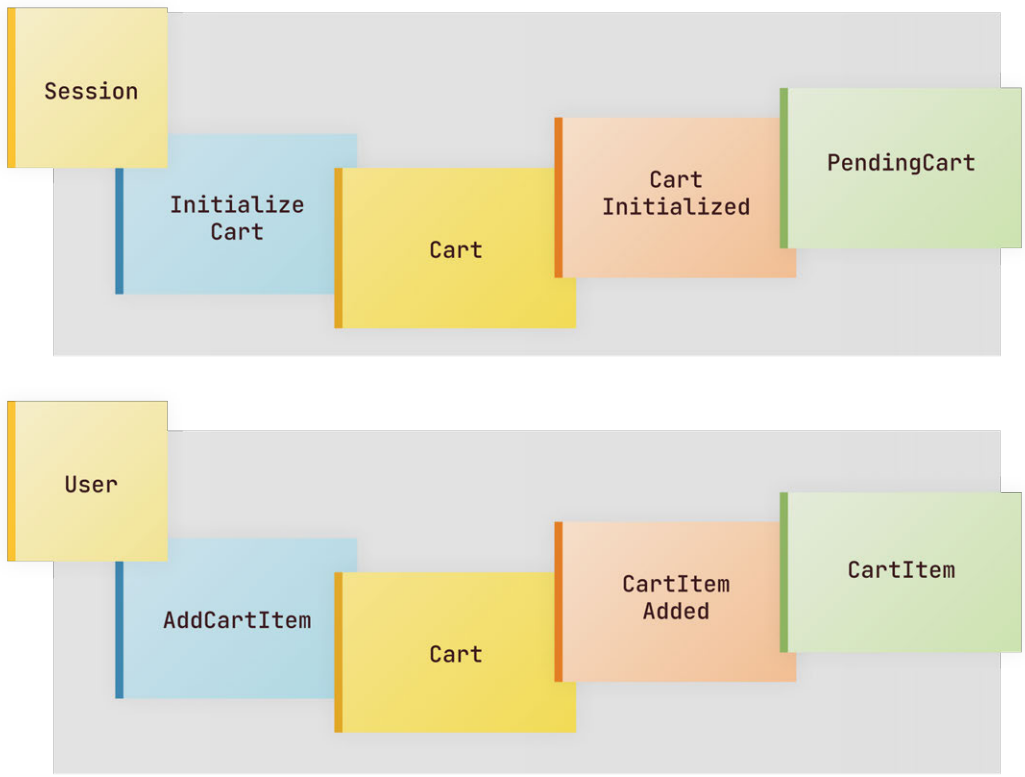
If you're doing a real event storming workshop with a client, it would be my recommendation to keep this goal in mind: what matters is discovering how a process works - not how to implement solutions technically.

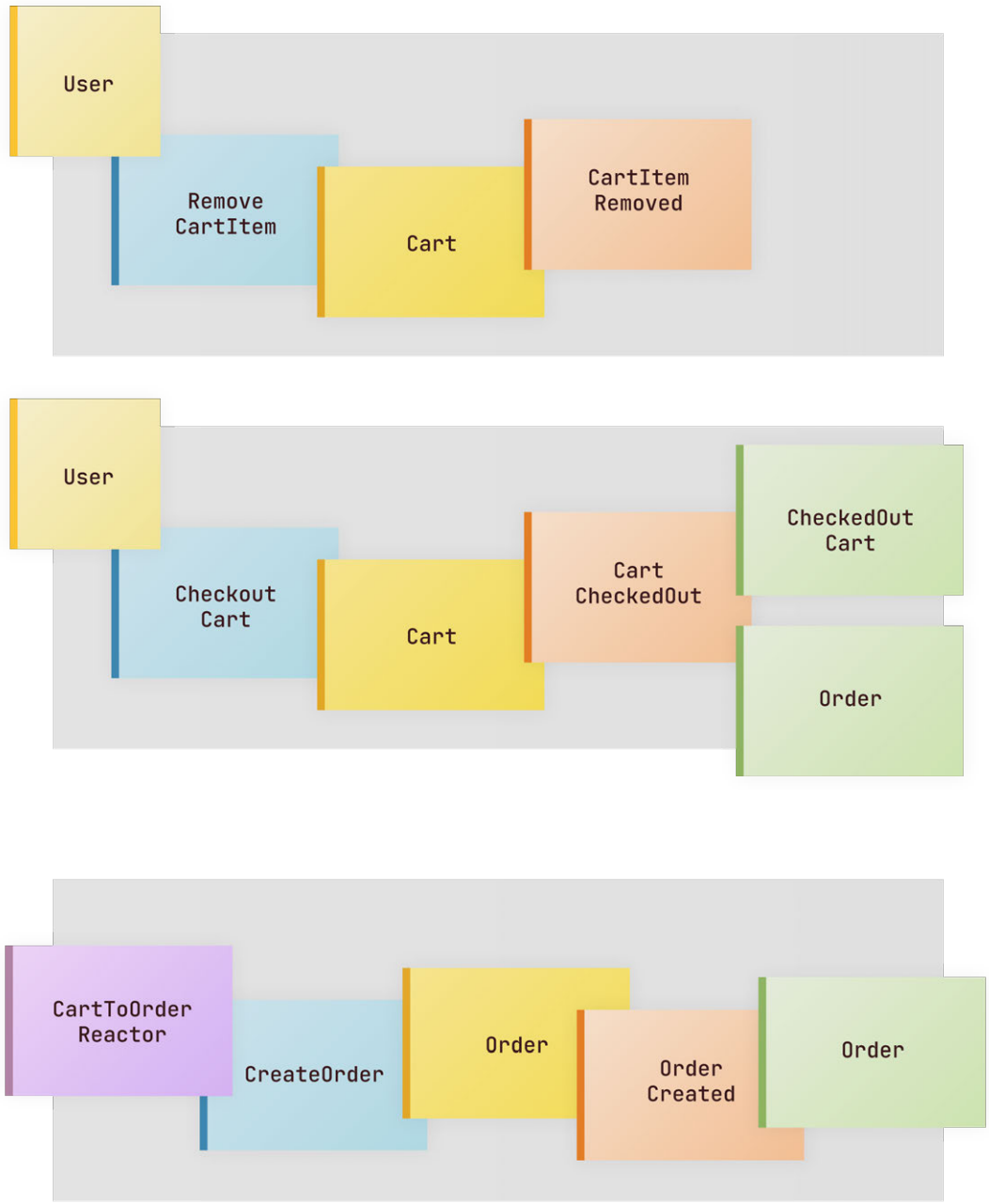
Throughout this book, we've used event storming in another way than Alberto meant it for. He said though, that there are very few rules to what he calls "the event storming game", so I'm confident in having used it in a way that worked for us.

In fact, I'd say both techniques have their value: mapping business processes with all stakeholders, as well as using sticky notes to model system architecture.

Another difference you'll note with the traditional event storming approach is the fact that our diagrams were much more vertically focused instead of horizontally. This might seem like a minor detail, but it signifies an important difference. "Big picture event storming" — the traditional way Alberto described it — has its focus on time moving forward. Our event flow diagram focused on detecting systems, boundaries, and relations between them - hence the difference.

For example, here's a diagram of what the flow of cart to order could look like with big picture event storming:





You can see it's much more linear and also repeats sticky notes like `Cart` or `Order`.

I figured it was important to make this side note at the end of the book. I didn't want this book to be "the definitive guide on event storming", so it's best to mention all caveats.

I would recommend looking into the original way of big picture event storming, because there's much to learn from, even as a software developer. I'll stick with a combination of both techniques: one for high level discovering and one for low-level architecture.

CHAPTER 21

DETAILS WORTH MENTIONING

There have been some small details throughout this book that might have made you think and wonder why they are done a specific way. They probably don't need a chapter on their own, so I want to address some of them in this chapter to make sure there are no loose ends.

UUIDs: objects or strings

In my examples, I always used strings to represent UUIDs. It's a common practice in the DDD community to actually model IDs with objects. By doing so, you can leverage the type checker and static analysis tools and prevent bugs. For example, a UUID belonging to a cart could be represented with a `CartUuid` instead of a string. On top of that, an object has more flexibility compared to a scalar type.

We're actually using UUID objects in our projects, but since I didn't find them to add value in the standalone examples in this book, I opted for strings.

Naming Aggregate Roots

You probably noticed that I always suffixed aggregate root classes throughout this book: `CartAggregateRoot`, `OrderAggregateRoot`, etc.

I usually suffix my classes when there's a likely chance of name collisions. Even though projections and aggregate roots live in separate namespaces, I find that sharing the same name (e.g. `Cart`) adds unnecessary cognitive overhead when programming, even when using an IDE and auto-imports. This problem is only amplified within the context of a book.

I deliberately chose to suffix aggregate roots instead of projections because of the expectations of the Laravel audience: most Laravel developers are used to naming their (read) models without suffix, and I wanted to mirror that.

If you're starting a new event sourcing project though, my advice would be to suffix projections and not the aggregate root. So `Cart` would be the aggregate root, and `CartProjection` could be a name for the projection.

Aggregate Partials

This was already briefly mentioned in the chapter on aggregate entities — a.k.a. partials — as well. In order to avoid confusion between the structural pattern of splitting aggregate roots into several classes and DDD entities, we've decided to rename `AggregateEntity` to `AggregatePartial` in the latest version of our event sourcing package.