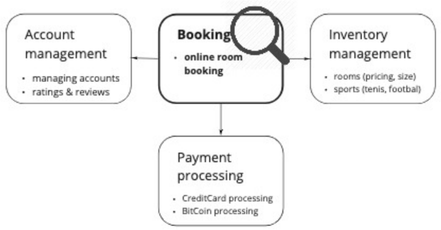
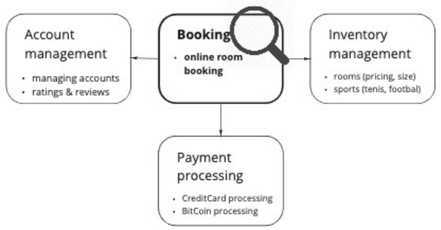


Demo

https://github.com/AxonIQ/hotel-demo

Event model **- 1. Brain storming**

We have someone explain the goals of the project and other information. The participants then envision what system would look and behave like. They put down all the events that they can conceive of having happened. Here we gently introduce the concept that only state-changing events are to be specified. Often, people will name "guest viewed calendar for room availability". We put those aside for now - they are not events.

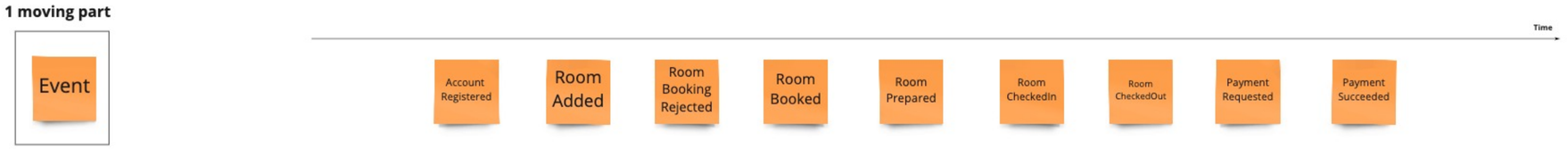


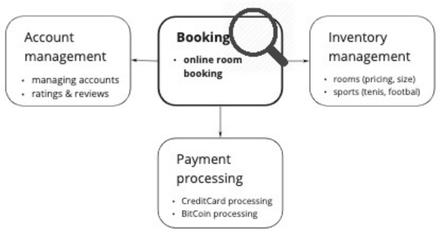
Demo

https://github.com/AxonIQ/hotel-demo

Event model **- 2. The plot**

The task is to create a plausible story made of these events. So they are arranged in a line and everyone reviews this time line to understand that this makes sense as events that happen in order



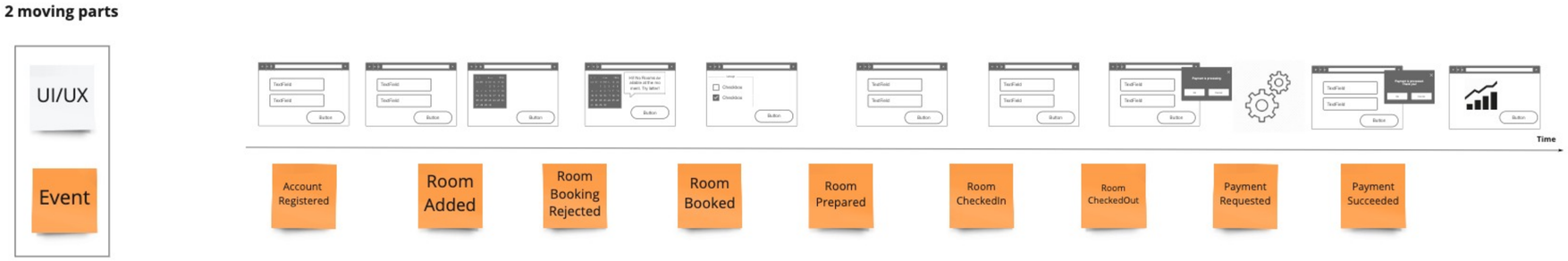


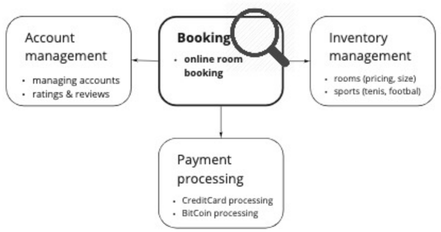
Demo

https://github.com/AxonIQ/hotel-demo

Event model **- 3. The story board**

Next, the wireframes or mockups of the story are needed to address those that are visual learners. More importantly, each field must be represented so that the blueprint for the system has the source of and destination of the information represented from the user's perspective. The wireframes are generally put at the top of the blueprint.



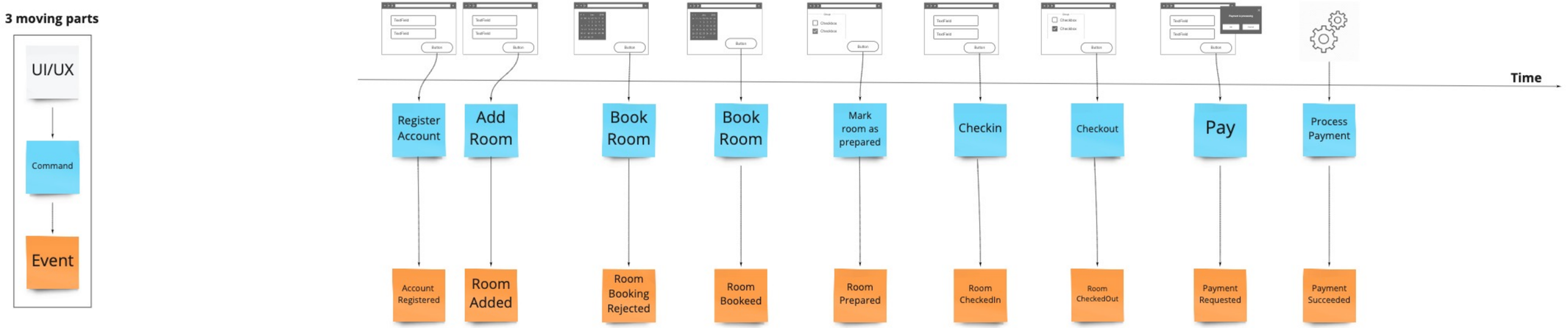


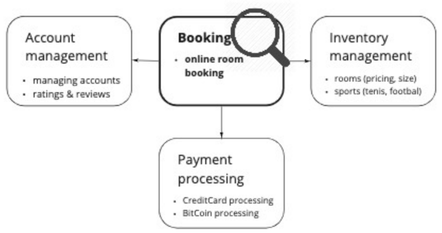
Demo

https://github.com/AxonIQ/hotel-demo

Event model **- 4. Identify inputs**

From the earlier section we saw that we need to show how we enable the user to change the state of the system. This is usually the step in which we do this introduction of these blue boxes. Each time an event is stored due to a users action, we link that to the UI by a command that shows what we are getting from the screen or implicitly from client state if it's a web application.



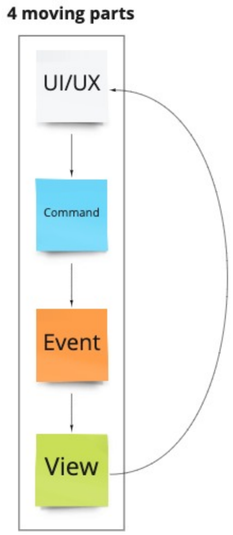


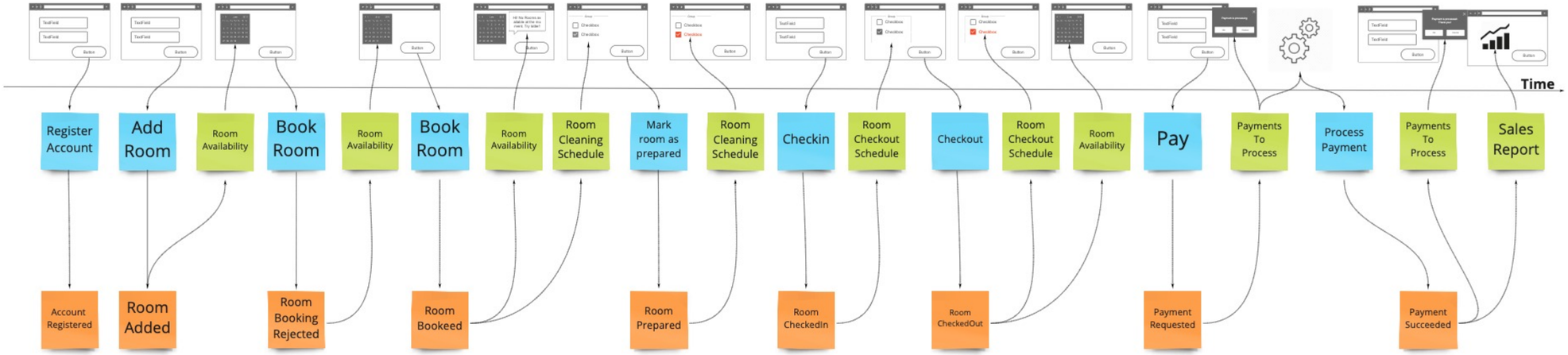
Demo

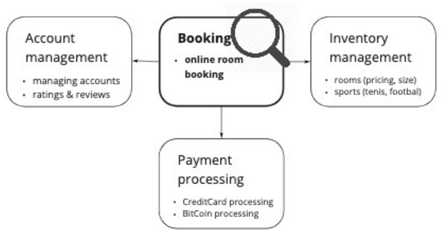
https://github.com/AxonIQ/hotel-demo

Event model **- 5. Identify outputs**

Again looking back at our goals for the blueprint, we now have to link information accumulated by storing events back into the UI via views (aka read-models).





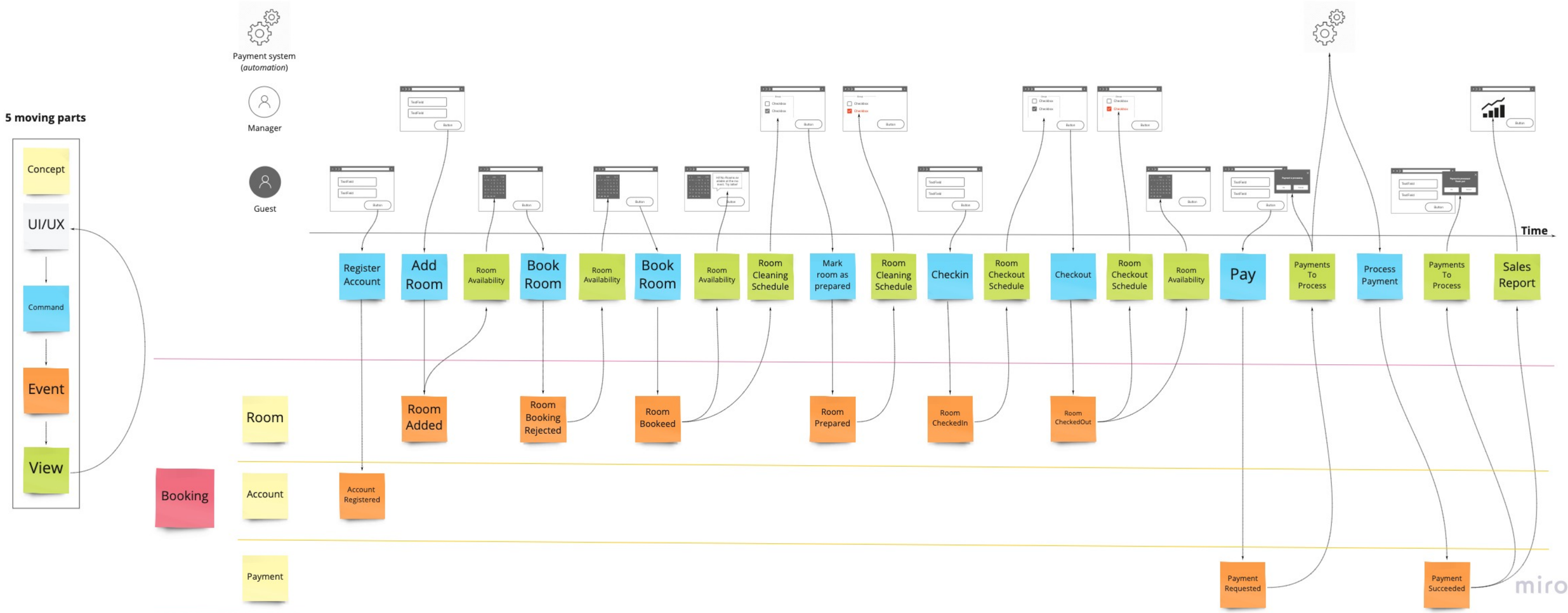


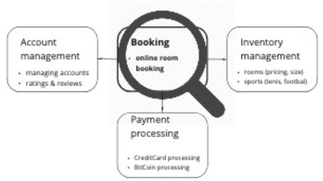
Demo

https://github.com/AxonIQ/hotel-demo

Event model **- 6. Identify roles and concepts**

Wireframes (UX/UI) are divided into separate swimlanes to show what each user can do or see. Additionally, we identify `concepts` and we group events by these concepts in independent swimlanes.





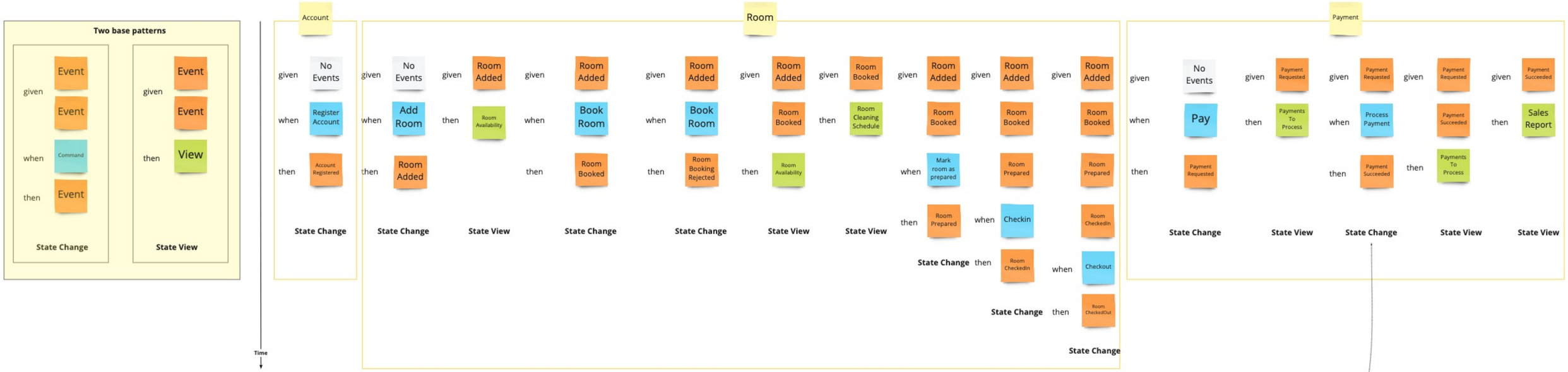
Demo

https://github.com/AxonIQ/hotel-demo

Event Model - Specification by example

Collaborative approach to defining requirements

Being more explicit about each State Change and State View we gain deeper understanding of the system requirements



Transition to the (Java) source code is immediate. We are able to reflect the presented white board in a series of `acceptance` tests very fast, without loosing any information.

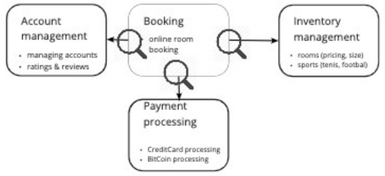
Axon Framework Test Fixture - Example

```
@Test
void processPaymentTest() {
   UUID accountId = UUID.randomUUID();
   UUID paymentId = UUID.randomUUID();
   PaymentRequested paymentRequested = new PaymentRequested(paymentId, accountId, BigDecimal.TEN);
   ProcessPaymentCommand processPaymentCommand = new ProcessPaymentCommand(paymentId);
   PaymentSucceeded paymentSucceeded = new PaymentSucceeded(paymentId);

   testFixture
         .given (paymentRequested)
         .when (processPaymentCommand)
         .expectEvents (paymentSucceeded);
}
```

Payment Requested

Process Payment

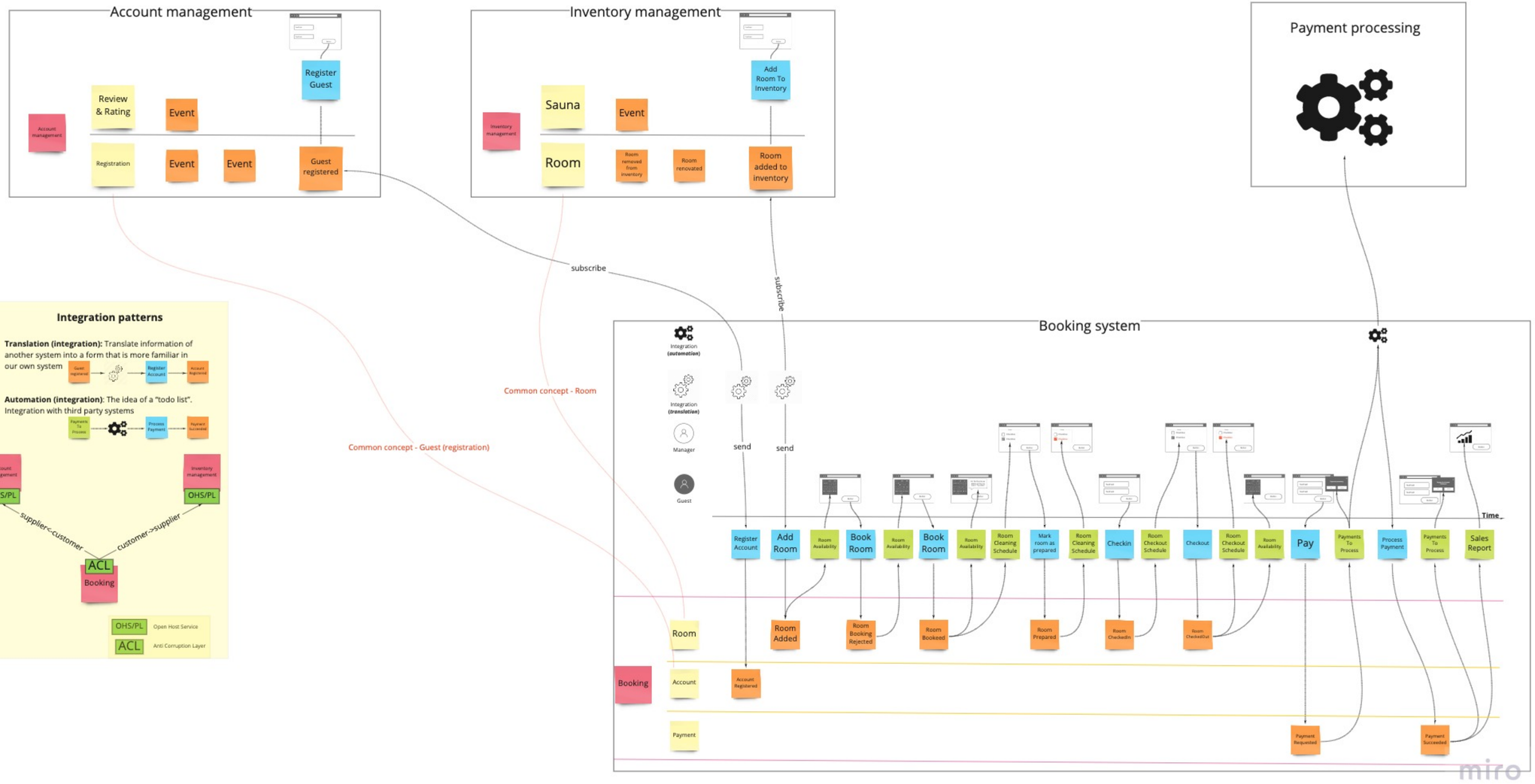


Demo

https://github.com/AxonIQ/hotel-demo

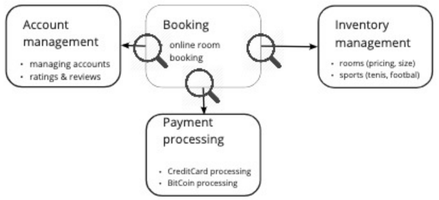
Systems Landscape - Integrations - Bounded contexts

It's often useful to understand how all of these software systems fit together within the bounds of an enterprise



DDD divides up a large system(s) into **Bounded Contexts,** each of which can have a unified model - essentially a way of structuring MultipleCanonicalModels. Bounded Contexts have both unrelated concepts (such as a RoomBooking only existing in a booking context) but also share concepts (such as Guest and Account). Different contexts may have completely different models of **common concepts** with mechanisms to map between these concepts for integration. Several strategic DDD patterns explore alternative relationships between contexts.

This enables independent evolution of your systems. For example, Booking is serving your customers for some time now, but as Accounting system is deployed, Booking subscribes to its event GuestRegistered rather than having the UI of its own. Accounting will grow without affecting Booking very much !

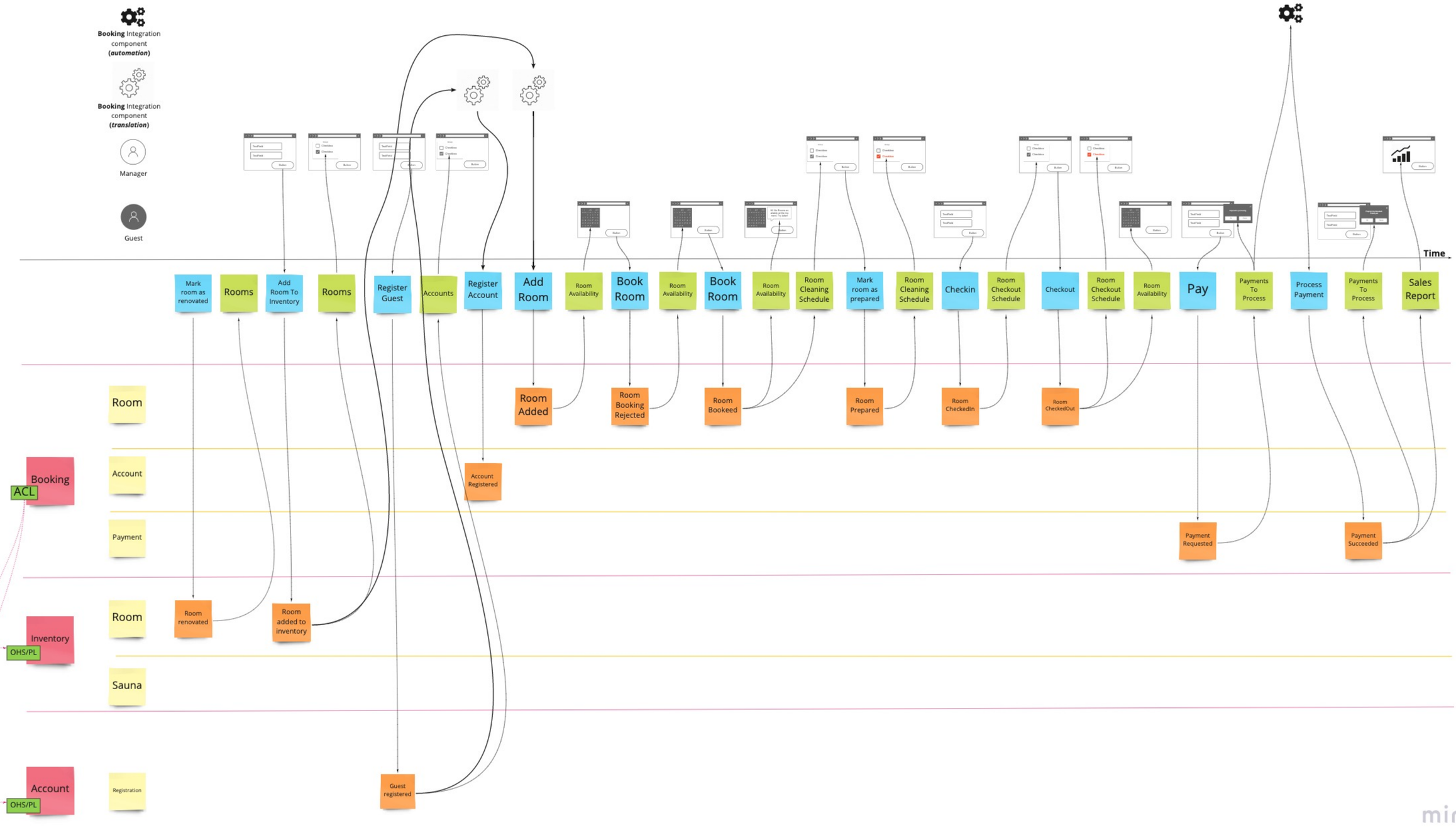


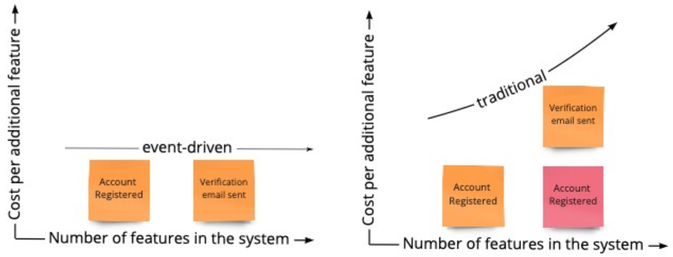
Demo

https://github.com/AxonIQ/hotel-demo

Systems Landscape - Integrations - Bounded contexts

One, final blueprint



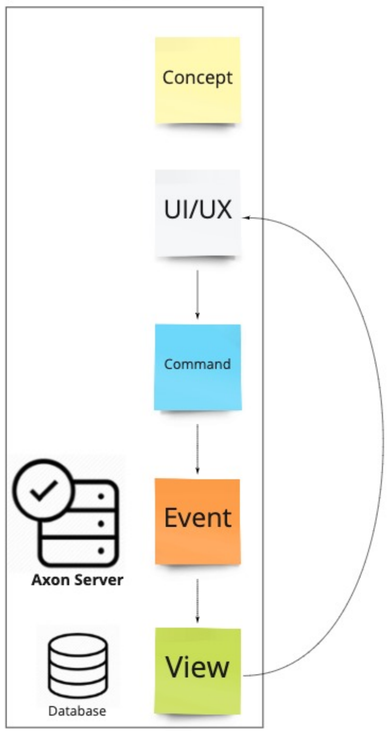


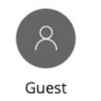
Demo

https://github.com/AxonIQ/hotel-demo

Event model **- Cost per additional feature**

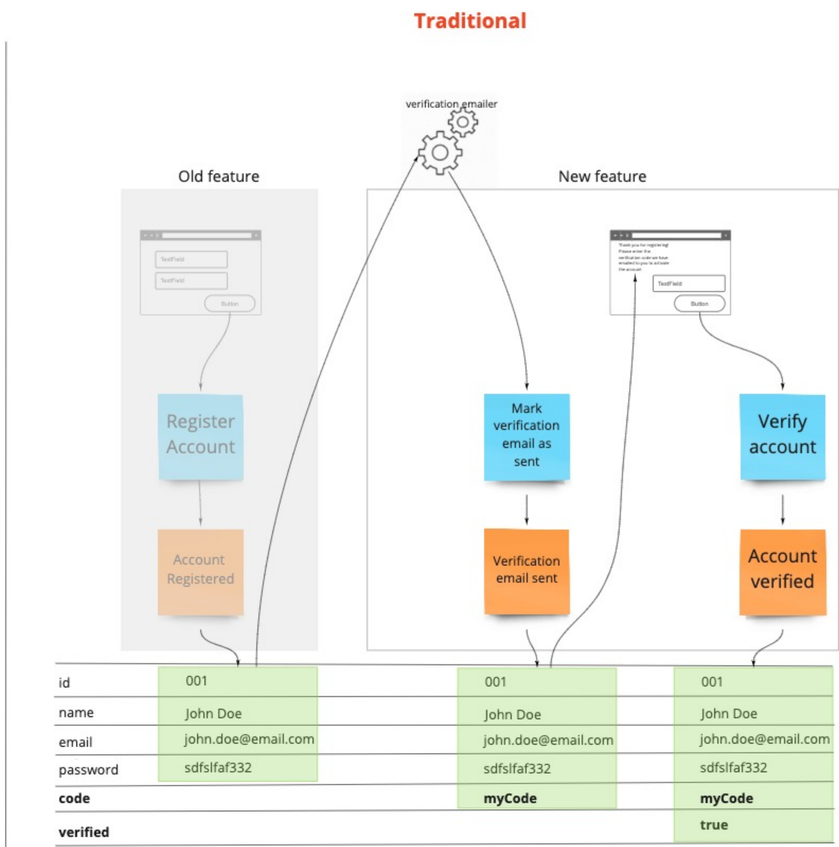
Event-Driven vs Traditional Systems





The query model is continuously updated to contain a certain representation of the current state (**state view**), based on the events. This way, every feature in the workflow has its own view (own table, own DB schema, ...), keeping features independent and making `cost per additional feature` flat. This is **CQRS.**

CQRS enables/unlocks Event Sourcing! Event Sourcing mandates that the state change of the application isn't explicitly stored in the database as the new state (overwriting the previous state) but as a series of events. This way you don't loose any data/information. Everything that happened in the system is stored. **Information is far more valuable then the price of the storage these days, Don't throw it away!**



Being 'efficient' with storage requires **re-opening the design of existing tables** as we add new features to our system. It is this rework that is responsible for features costing more and more as the size of the whole system grows.