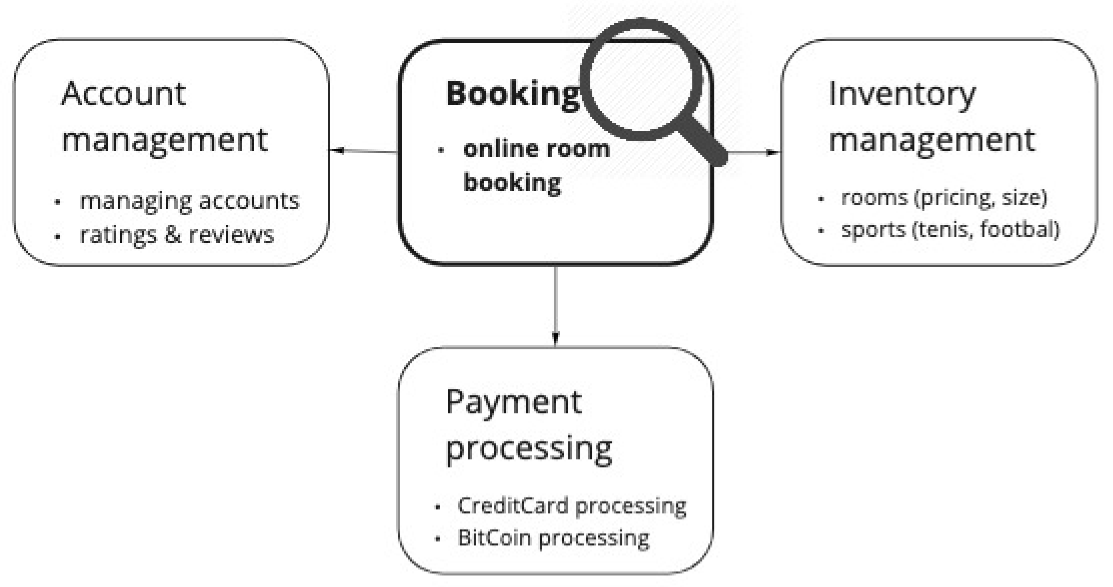


Demo

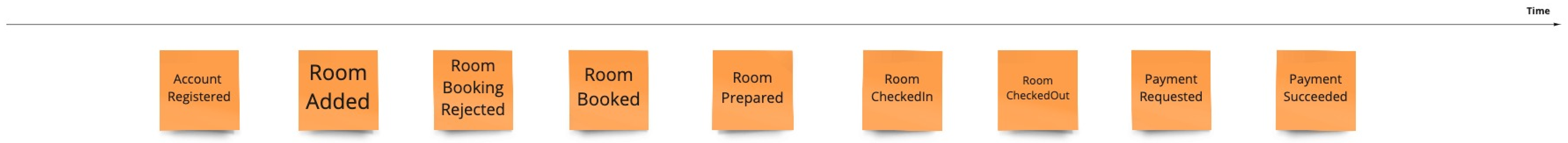
https://github.com/AxonIQ/hotel-demo

Event model

"What a system is supposed to do from start to finish, on a time line and with no branching"

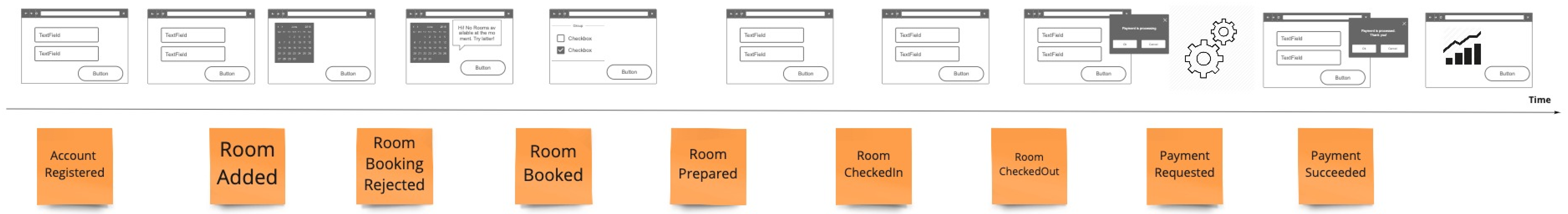
1 moving part

Step 1: "A plausible story made of events"

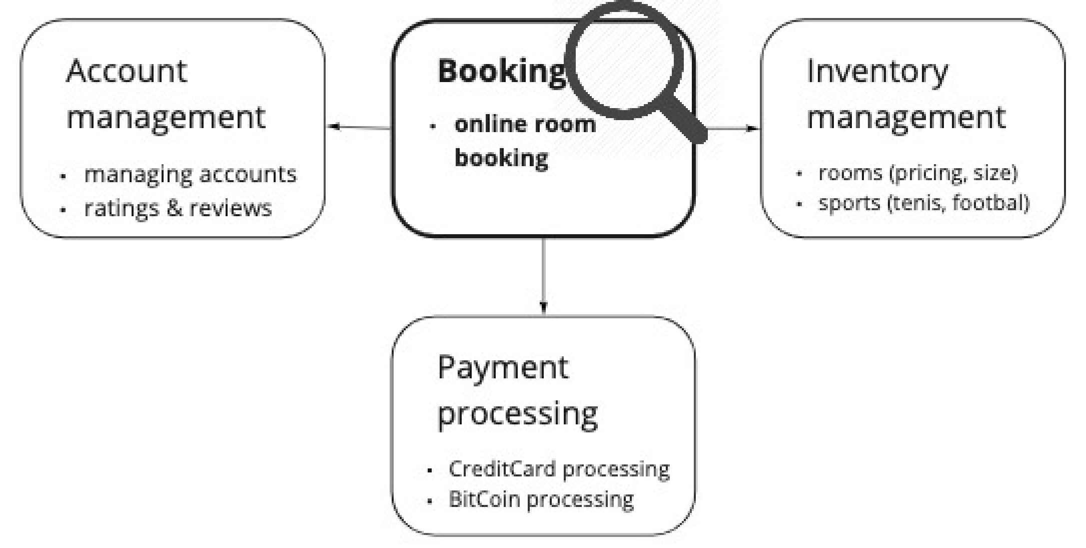


2 moving parts

Step 2: "Adding wireframes to address visual learners."



miro



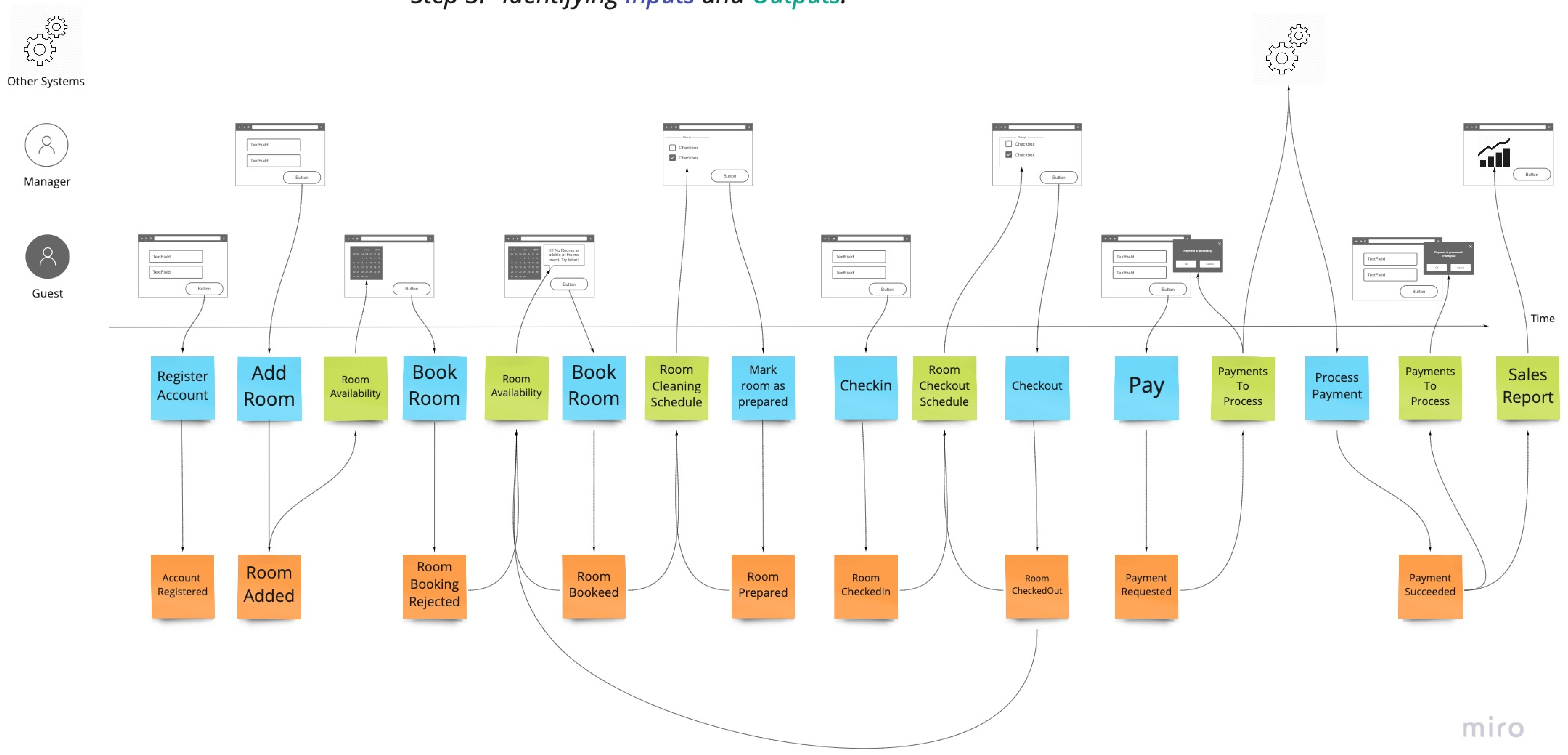
Demo

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

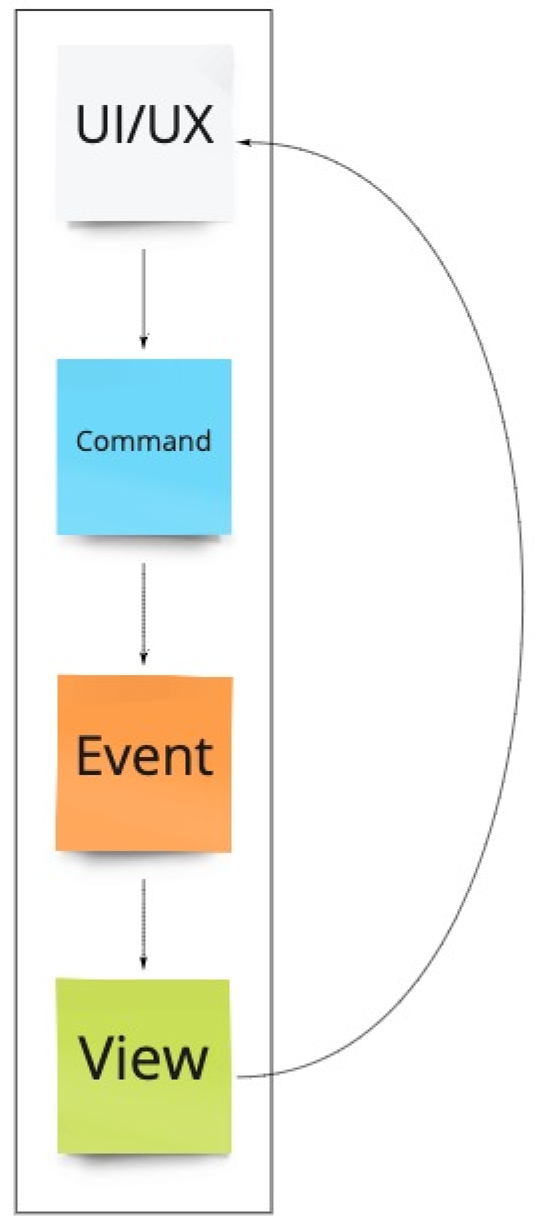
Event model

"What a system is supposed to do from start to finish, on a time line and with no branching"

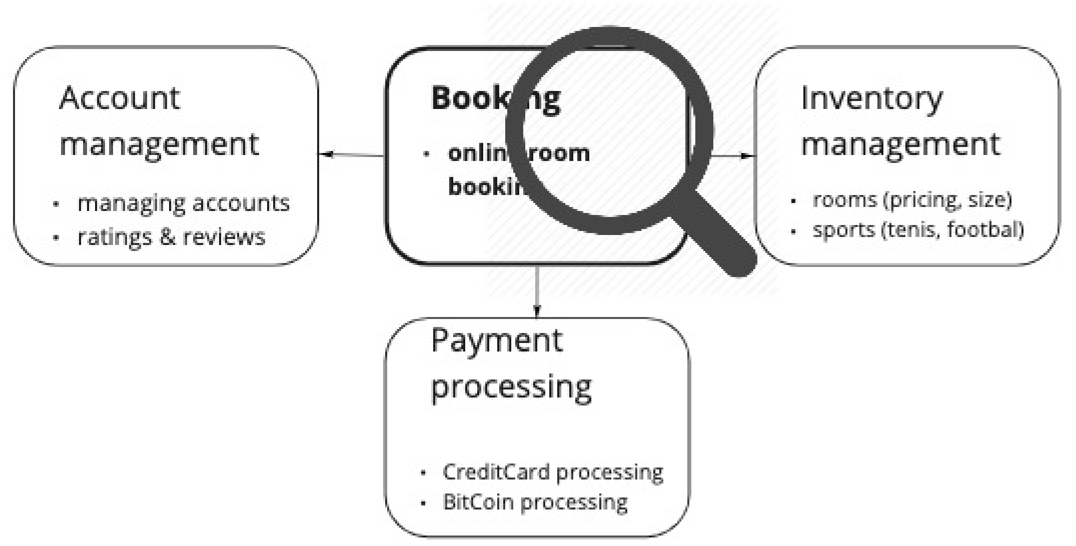
Step 3: "Identifying Inputs and Outputs."



4 moving parts



miro



Demo

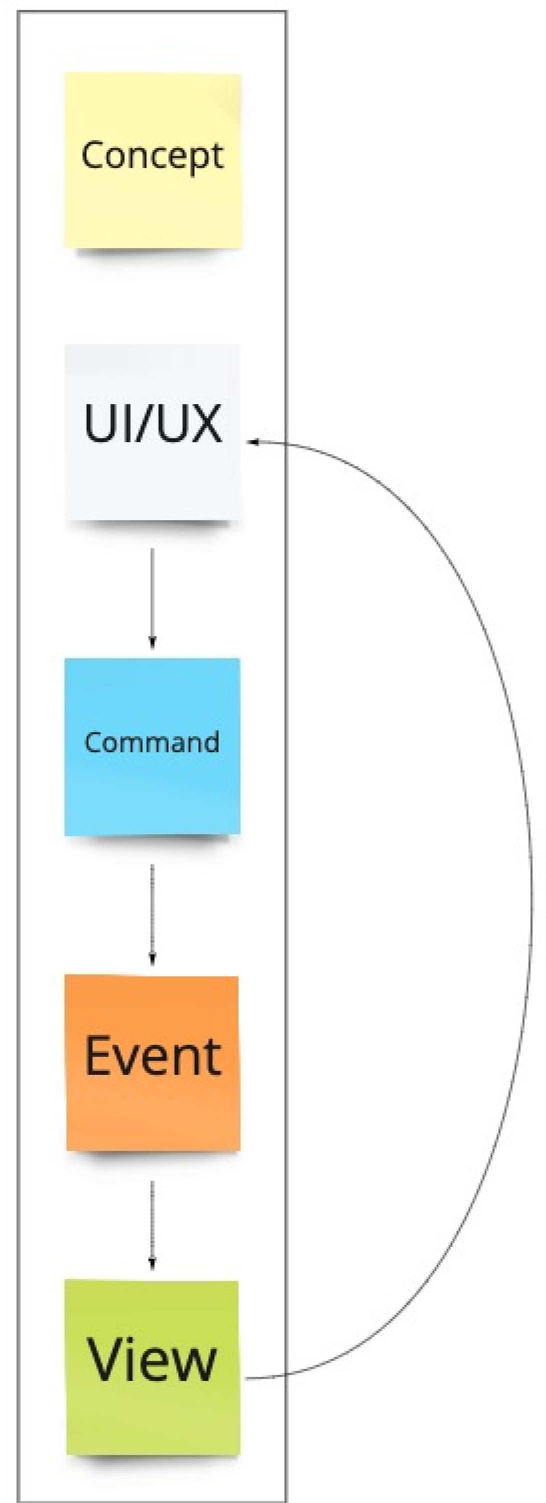
https://github.com/AxonIQ/hotel-demo

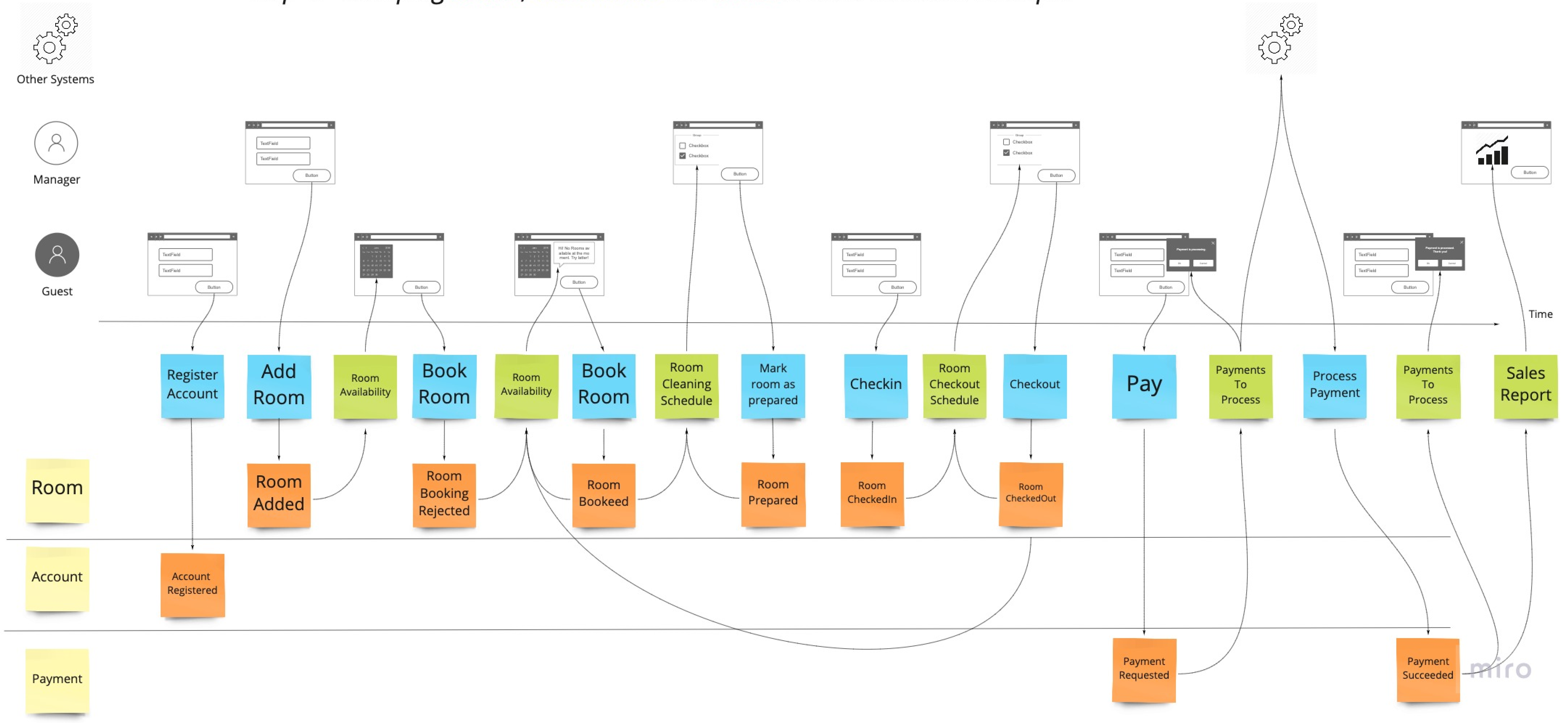
Event model

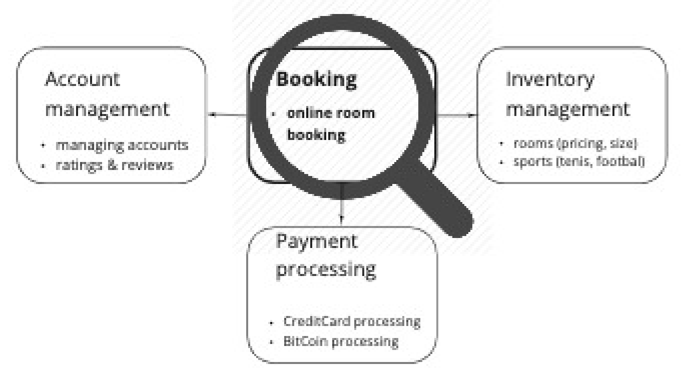
"What a system is supposed to do from start to finish, on a time line and with no branching"

Step 4: "Grouping events, commands and views around business concepts"

5 moving parts







Demo

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

Event Model - Specification by example

"Collaborative approach to defining requirements"

Step 5: "Being more explicit about each State Change and State View we gain deeper understanding of the system requirements"

Two base patterns
given
Event
Event
when
Command
then
Event
State Change
given
Event
Event
then
View
State View
Time
Account
given
No Events
when
Register Account
then
Account Registered
State Change
Room
given
No Events
when
Add Room
then
Room Added
State Change
given
Room Added
then
Room Availability
State View
given
Room Added
when
Book Room
then
Room Booked
State Change
given
Room Added
when
Book Room
then
Room Booking Rejected
State Change
given
Room Added
Room Booked
then
Room Availability
State View
given
Room Booked
then
Room Cleaning Schedule
State View
given
Room Added
Room Booked
when
Mark room as prepared
then
Room Prepared
State Change
given
Room Added
Room Booked
Room Prepared
when
Checkin
then
Room CheckedIn
State Change
given
Room Added
Room Booked
Room Prepared
Room CheckedIn
when
Checkout
then
Room CheckedOut
State Change
Payment
given
No Events
when
Pay
then
Payment Requested
State Change
given
Payment Requested
then
Payments To Process
State View
given
Payment Requested
when
Process Payment
then
Payment Succeeded
State Change
given
Payment Requested
Payment Succeeded
then
Payments To Process
State View
given
Payment Succeeded
then
Sales Report
State View

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.

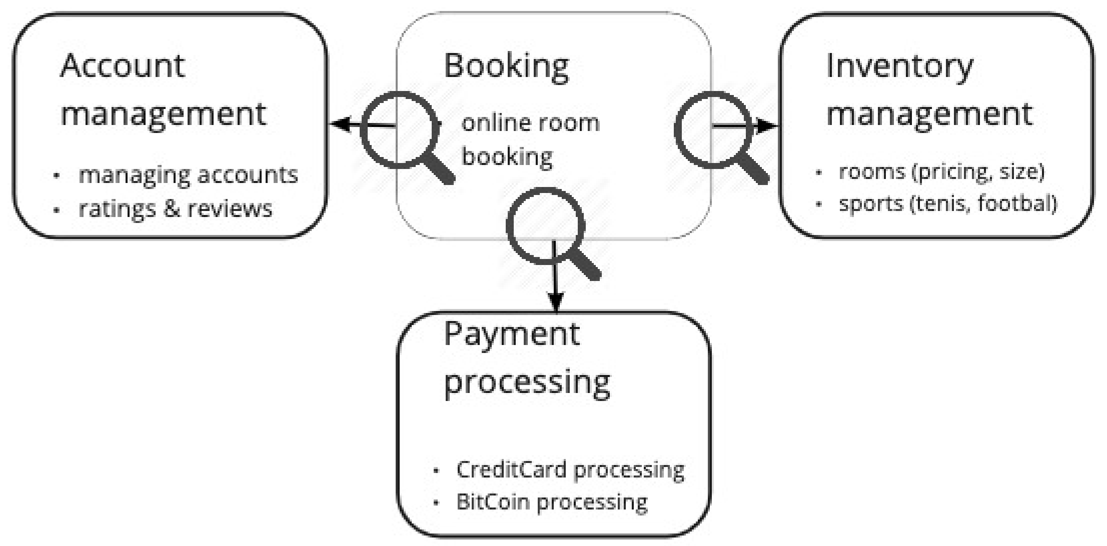
Axon Framework Test Fixture - Example

```
@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
Payment Succeeded

miro

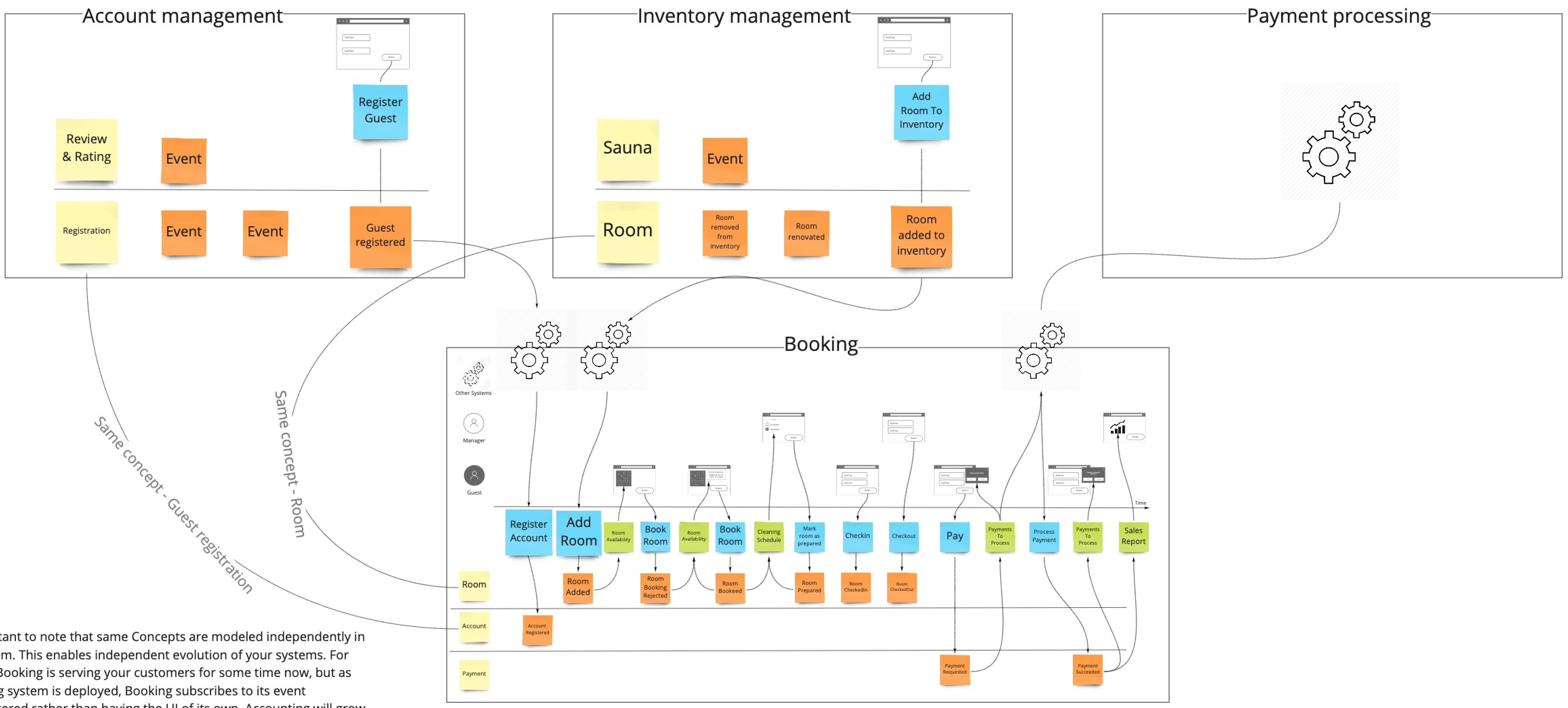


Demo

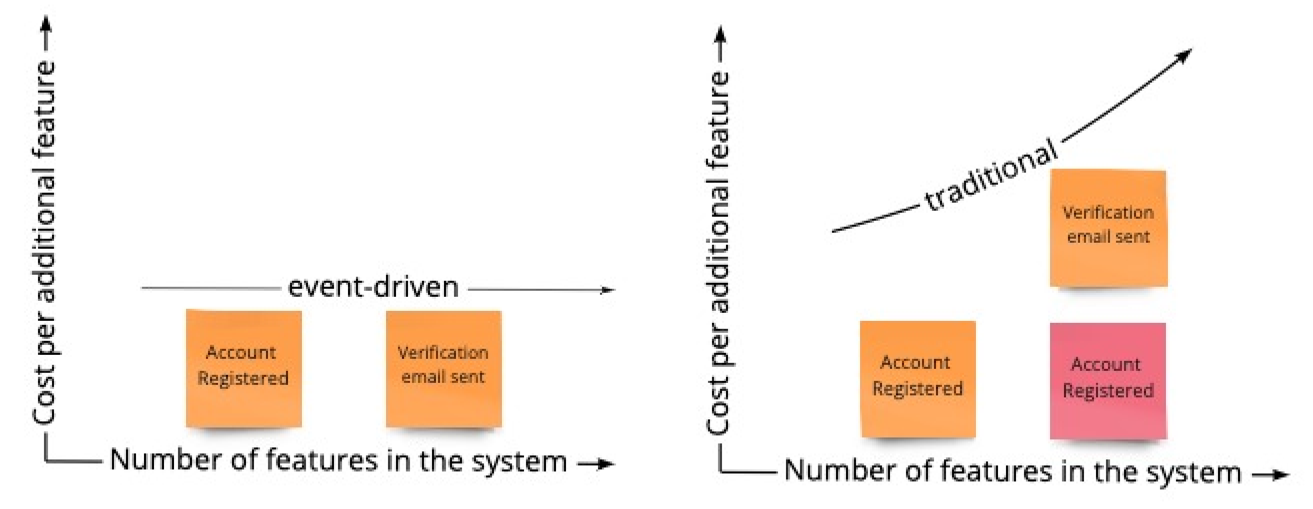
https://github.com/AxonIQ/hotel-demo

Systems Landscape - Integrations

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



It is important to note that same Concepts are modeled independently in each System. 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 GuestRgistered rather than having the UI of its own. Accounting will grow without affecting Booking very much !



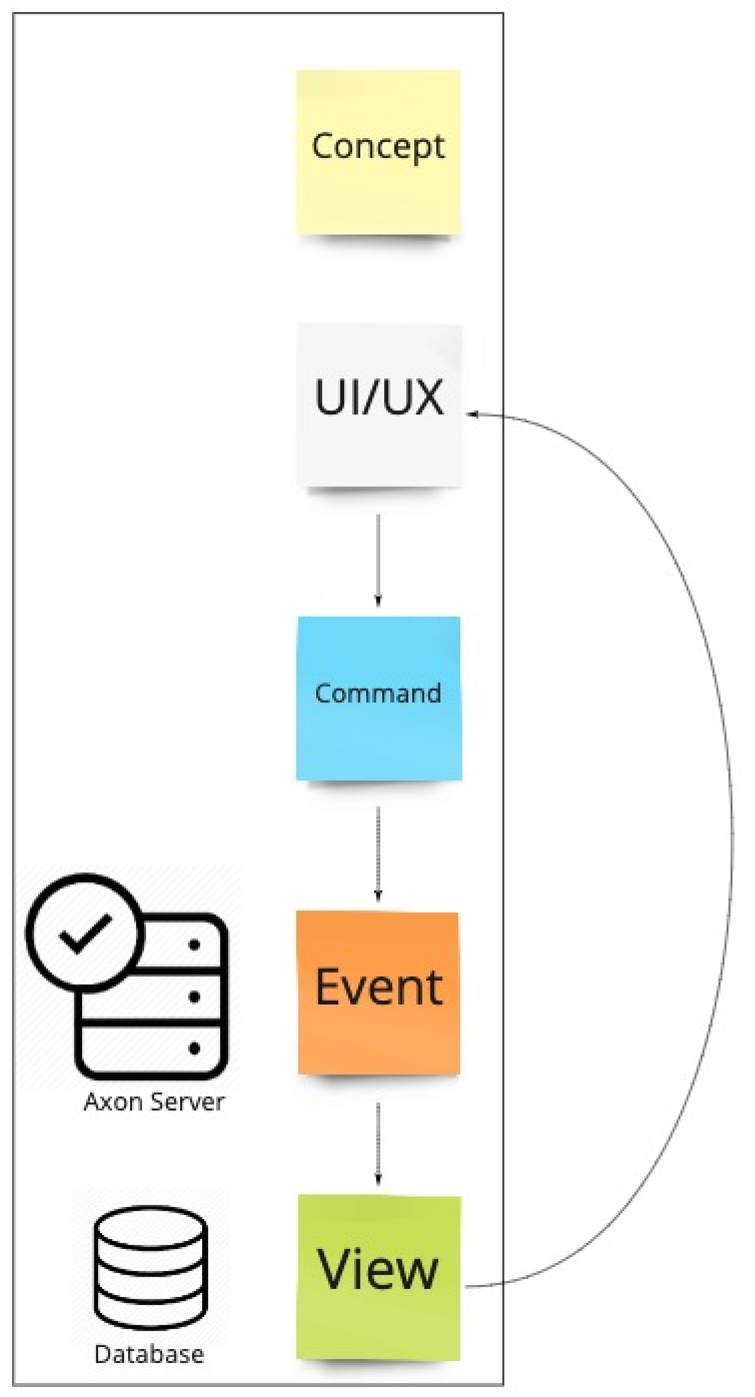
Demo

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

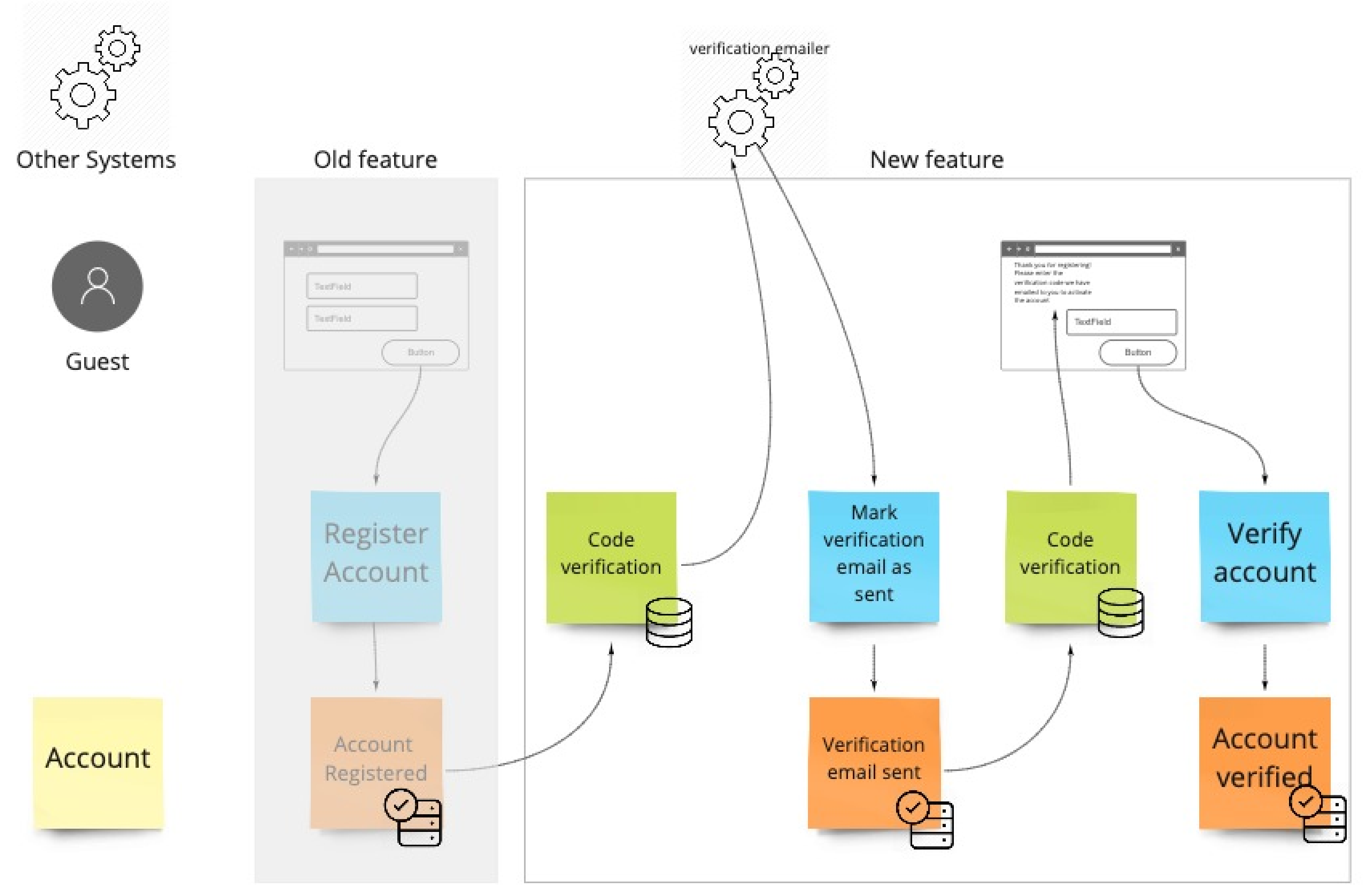
Event model

"Cost per additional feature - Event-Driven vs Traditional Systems "

5 moving parts



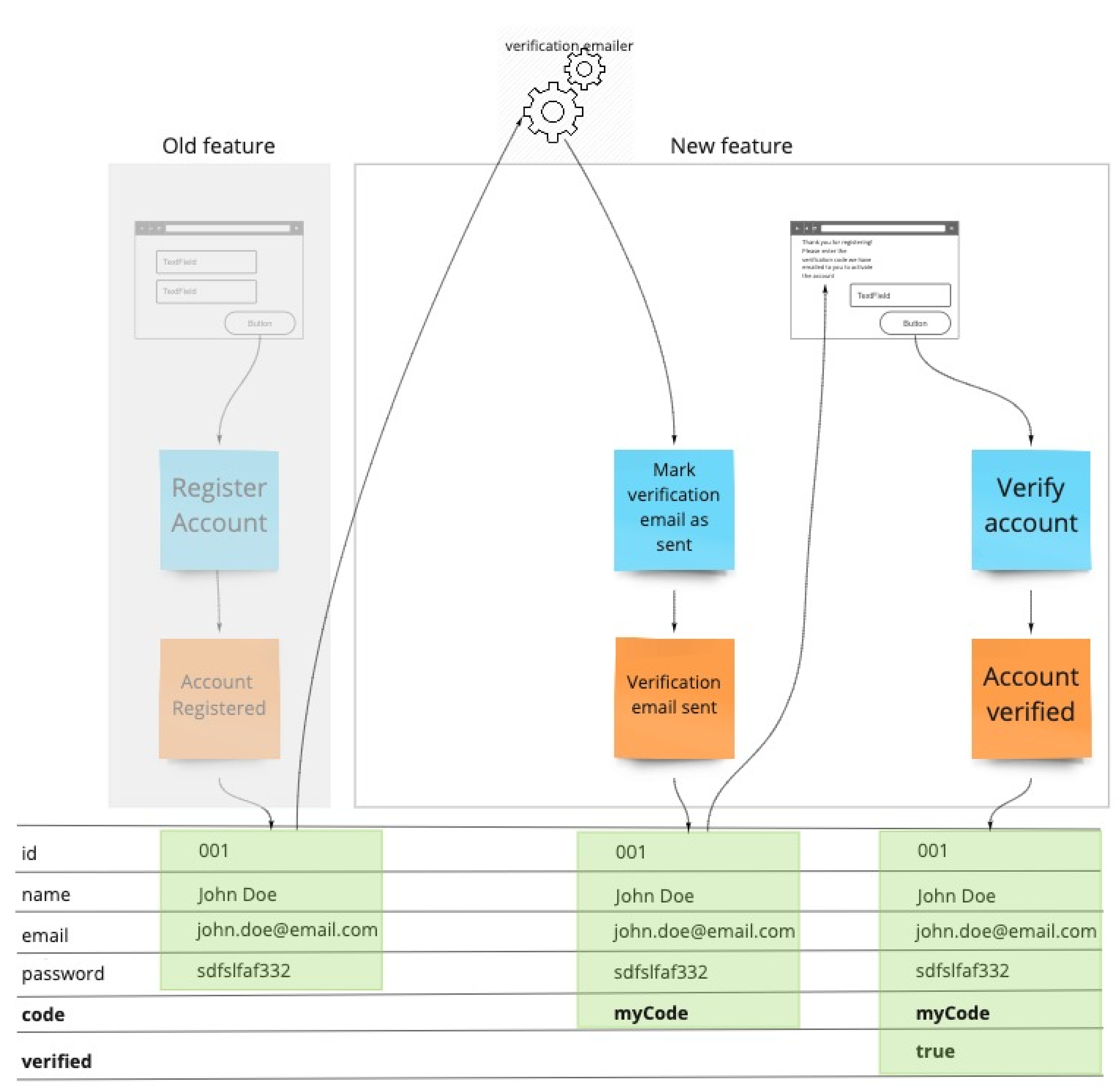
Event-Driven



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 <u>making `cost per additional feature` flat.</u> 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!**

Traditional



id	001	001	001
name	John Doe	John Doe	John Doe
email	john.doe@email.com	john.doe@email.com	john.doe@email.com
password	sdfslfaf332	sdfslfaf332	sdfslfaf332
code		**myCode**	**myCode**
verified			**true**

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.