Axon Training

Module 1 – Fundamentals of DDD and CQRS

AxonIQ

Tips and tricks for attendees

How this works...

Delivery of the training

- Online Sessions
  - 2-hour sessions
  - Previous day recap and Q&A – 30 minutes
  - Topic of the day – 60 minutes
  - Q&A and Labs – 30 minutes

- Labs
  - Exercises for each module
  - During appx. last 30 minutes of online session,
    or at your own leisure

- Slack channel
  - For discussions and Q&A outside of online session times
  - “Best effort” availability of trainer

Tips & Tricks

- Please mute yourself to avoid background noise
- Press and hold “space-bar” to talk.
  Releasing automatically puts you on mute again
- Press space once to mute/unmute

- If you have a question, ask it!
  (stupid questions are the ones that were never asked)

Please introduce yourself

Who's who

Agenda

Week 1

1. **DDD and CQRS Fundamentals**
2. Command Model
3. Event Handling & Projections
4. Sagas and Deadlines

Week 2

1. Snapshotting and Event Processors
2. Preparing for Production
3. CQRS and Distributed Systems
4. Monitoring, Tracing, Advanced Tuning

Tackling complexity in the heart of software

Domain Driven Design

AxonIQ

Domain-Driven Design

- The Domain Model is the “heart” of an application.
- An increase of complexity is often caused by a poorly designed model.

Domain

> A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.

Model

> A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.

The domain

Model

AxonIQ

Model

AxonIQ

Entity

> Objects that are not fundamentally defined by their attributes, but rather by a thread of continuity and identity.

Value Object

> Value objects have no conceptual identity, but are fundamentally defined by their attributes.
> They describe some characteristic of a thing.
>
> Value Objects are Immutable.

Entity or value object?

Human Being

It depends

Entity or value object?

10 Euro bill

It depends

Mixing paint – Entities

```
Bucket literOfRed = new Bucket(1L, Color.RED);
Bucket literOfWhite = new Bucket(1L, Color.WHITE);

literOfRed.add(literOfWhite);
// now literOfRed contains 2 liters…. Of PINK!
// and literOfWhite… Empty! Or is
it?
```

Mixing paint – Value Objects

```
Bucket literOfRed = new Bucket(1L, Color.RED);
Bucket literOfWhite = new Bucket(1L, Color.WHITE);

Bucket lotsOfPink = literOfRed.mixedWith(literOfWhite);
// “magically” a new bucket appears
😊
Bucket evenMorePaint = newBucket.mixedWith(literOfWhite);
```

Bounded context

> Explicitly define the context within which a model applies. Explicitly set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas. Keep the model strictly consistent within these bounds, but don't be distracted or confused by issues outside.

Repository

> A mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects.

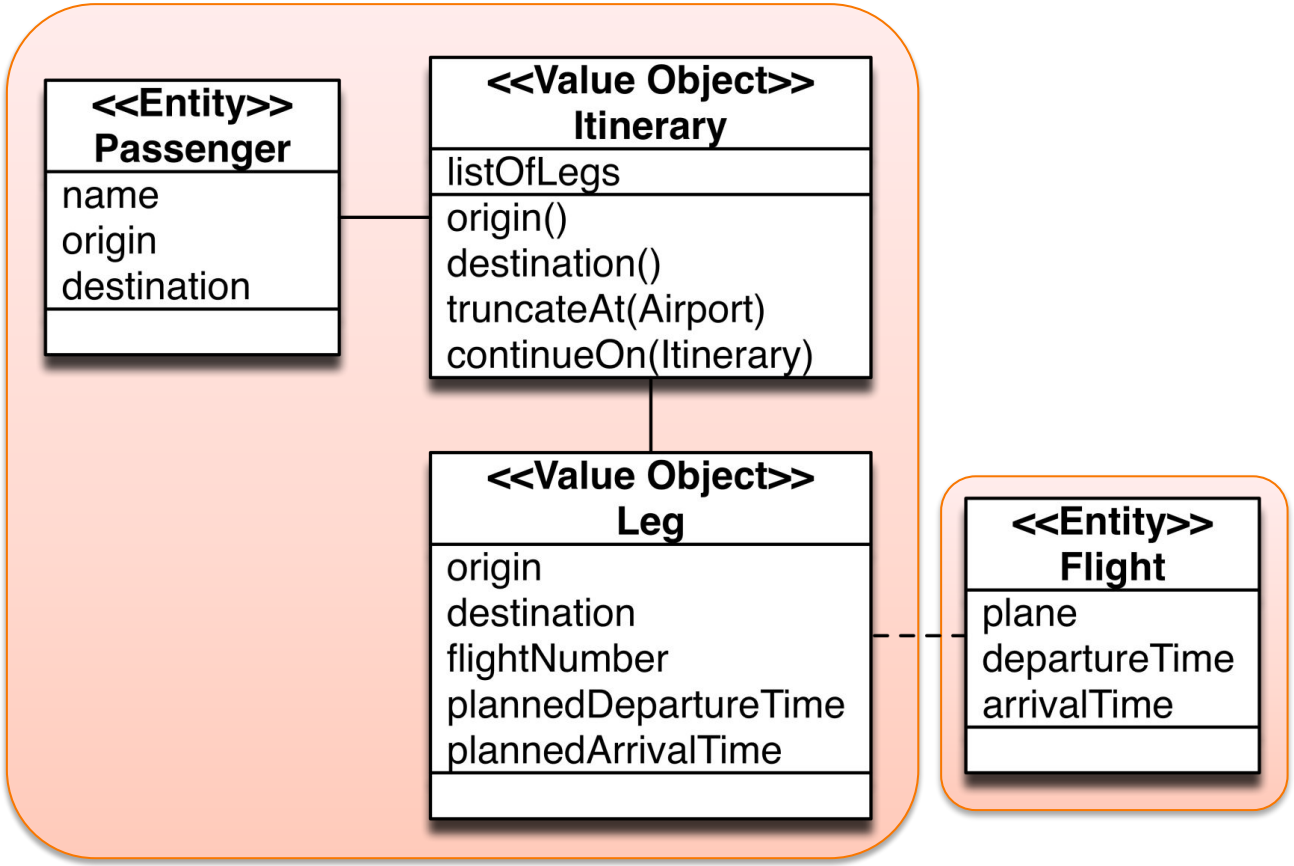
Aggregate

> A group of associated objects which are considered as one unit with regard to data changes…

Aggregate

> External references are restricted to one member of the aggregate, designated as the Root. A set of consistency rules applies within the Aggregate's boundaries.

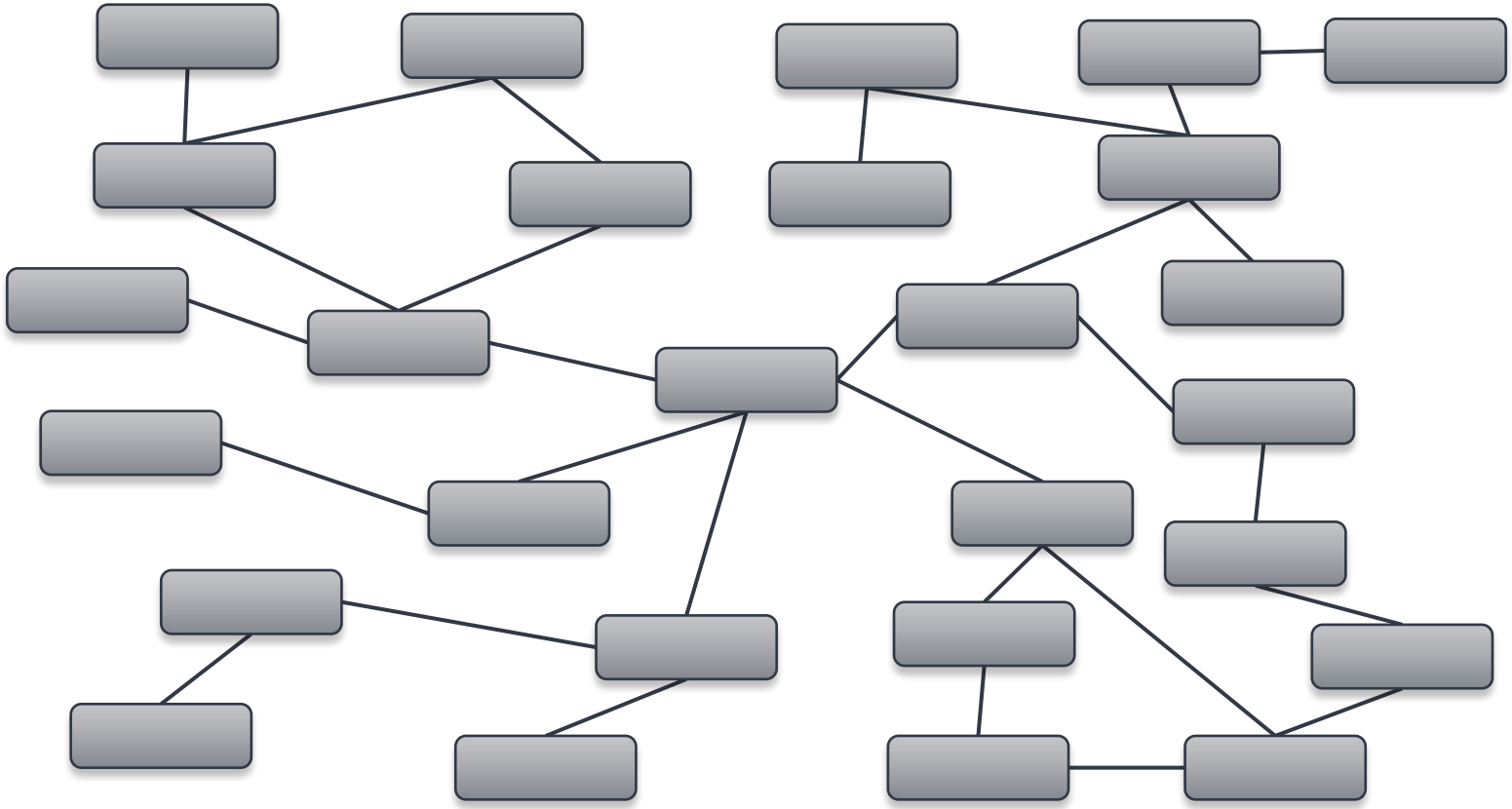
Where is the boundary?



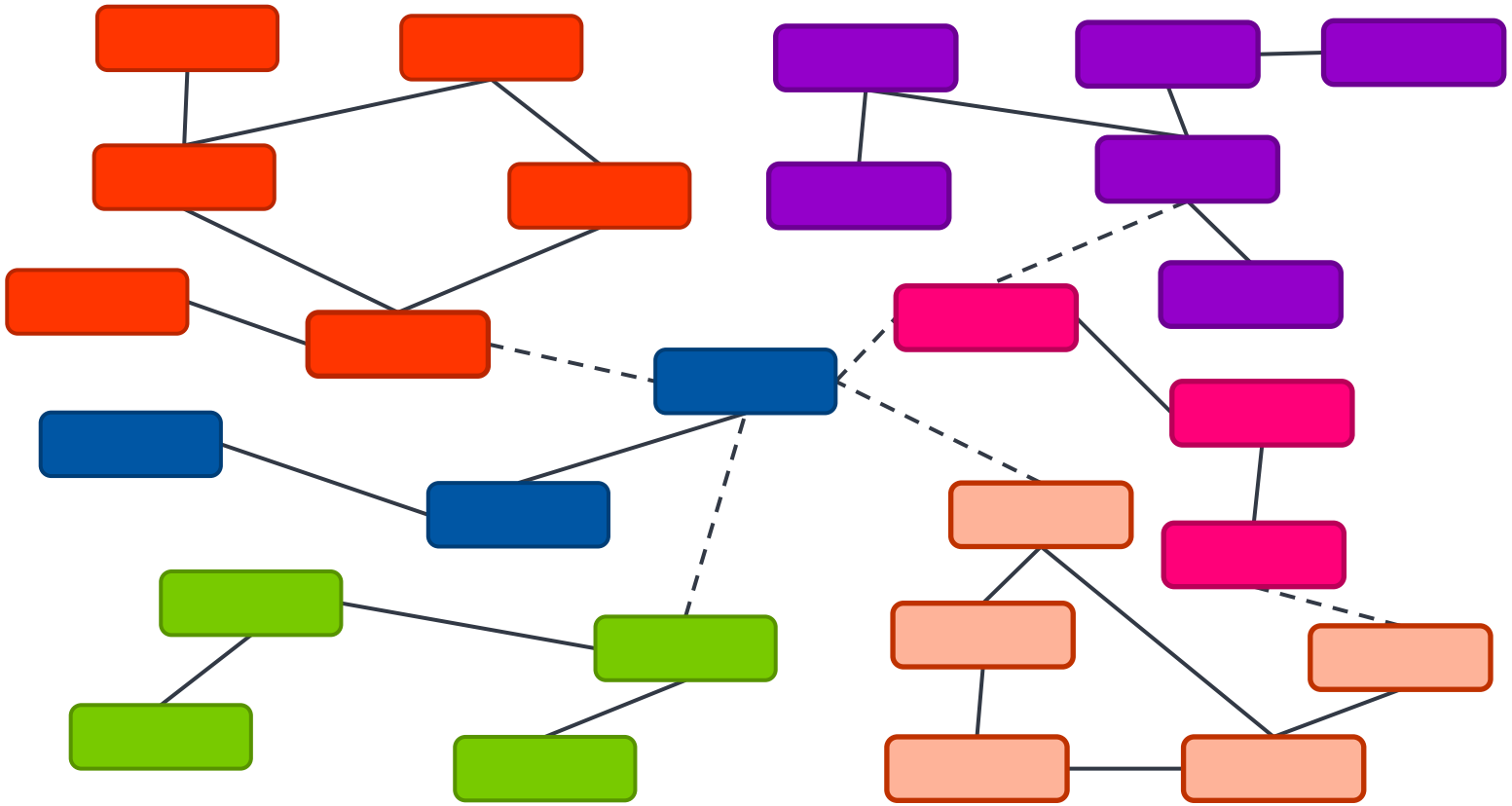
Event

> A notification that something relevant has happened inside the domain.

Adding structure to the model



Adding structure to the model



But still...

- Scope of Model too large
- What is the model actually optimized for?
- How do we deal with different (non-functional) concerns?
- What if a service needs to interact with more than one aggregate?

Divide and conquer…

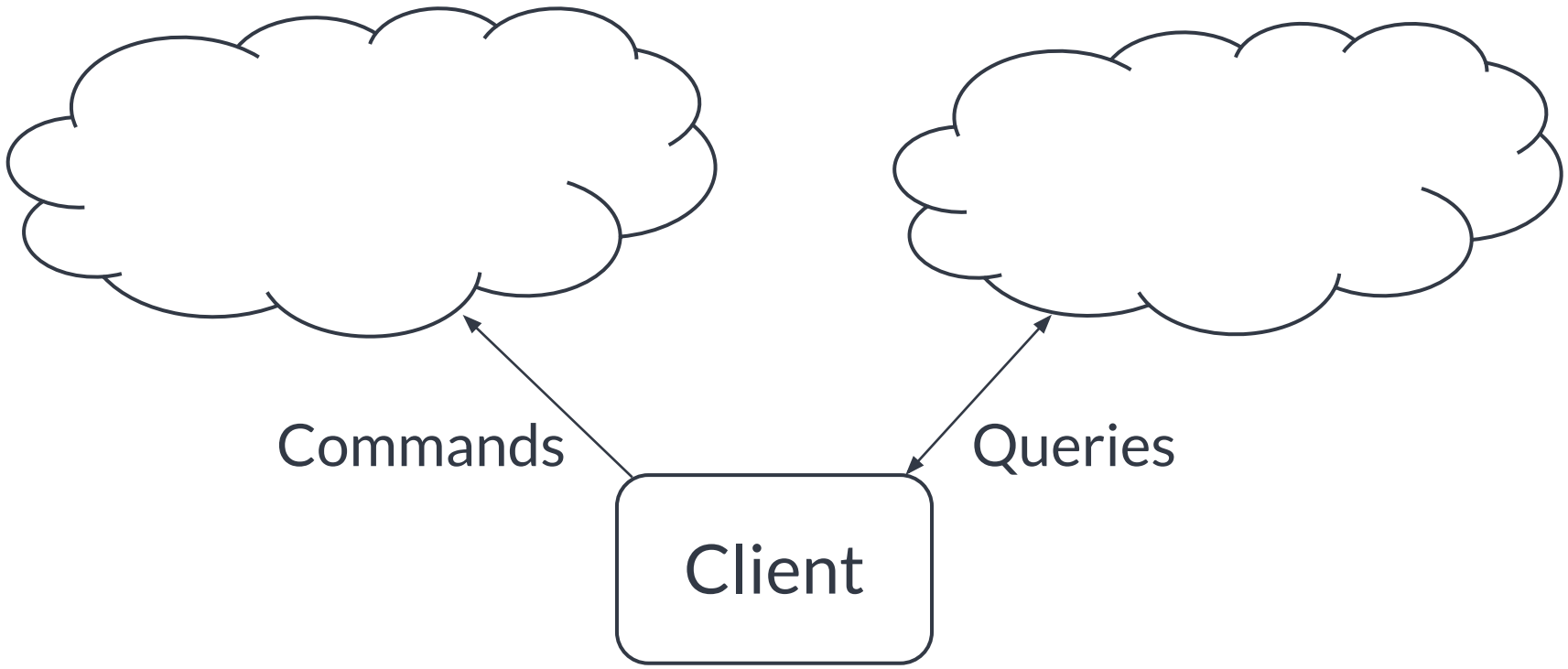
Command Query Responsibility Segregation

CQRS - Definition

Command-Query Responsibility Segregation is an architectural pattern that distinguishes between two parts of an application:

- one with the responsibility to process commands,
- another that provides information (queries).

Command Query Responsibility Segregation



Two Models

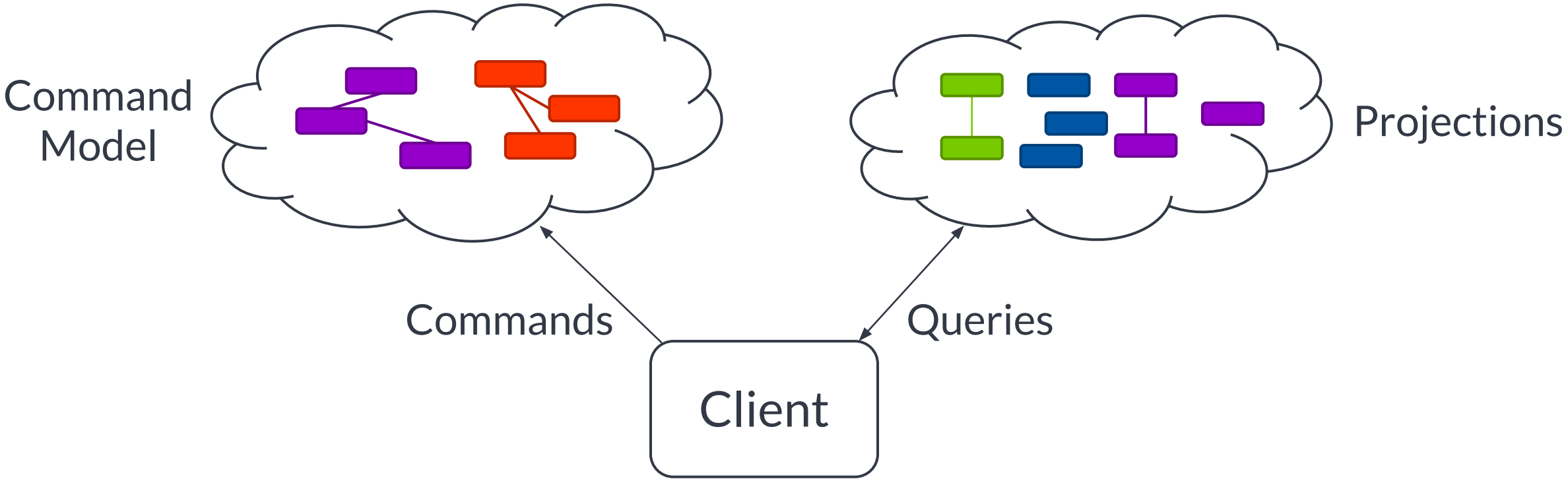
Command Model

- Focused on executing tasks.
- Primarily expressed in operations.
- Only contains data necessary for task execution and decision making.

Query Model / Projections

- Focused on delivering information.
- Data is stored the way it is used.
- Denormalized / “table-per-view”

Command Query Responsibility Segregation



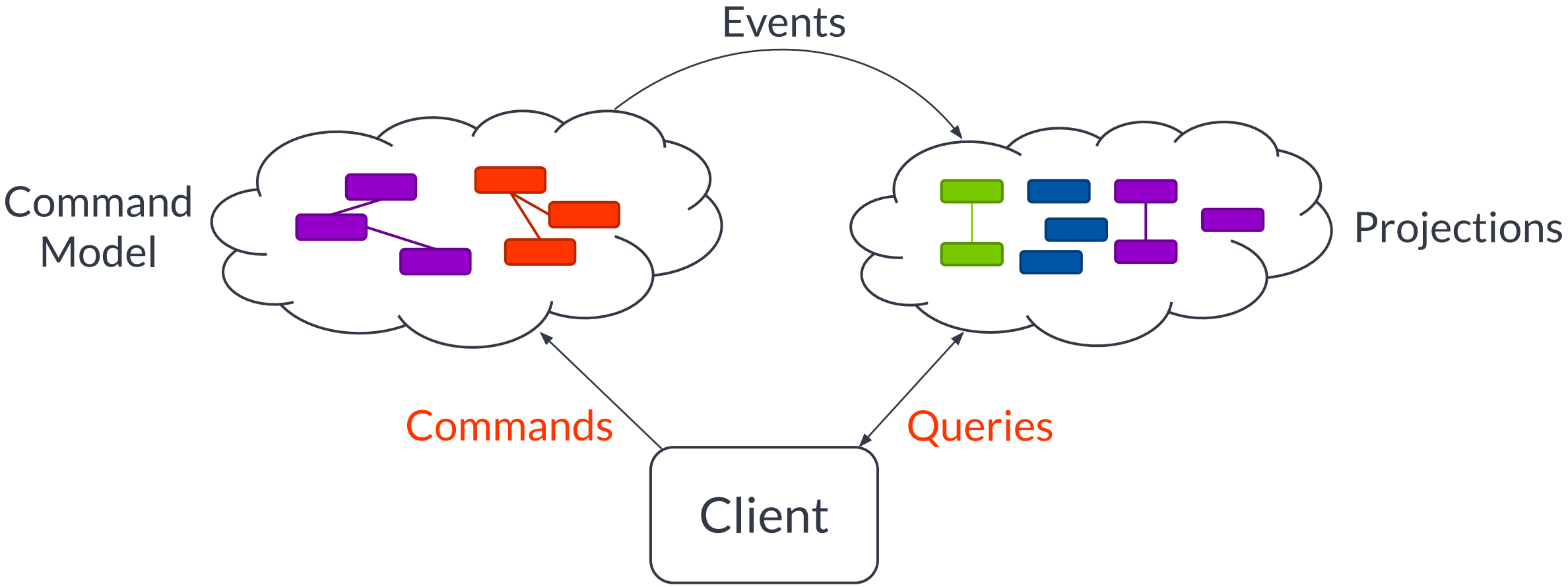
Synchronization of models

Changes in the Command Model should (eventually) be visible in the Query Model.

Options:

- Shared data source
- Stored procedures
- Event-Driven Architecture

Command Query Responsibility Segregation



CQRS Building Blocks

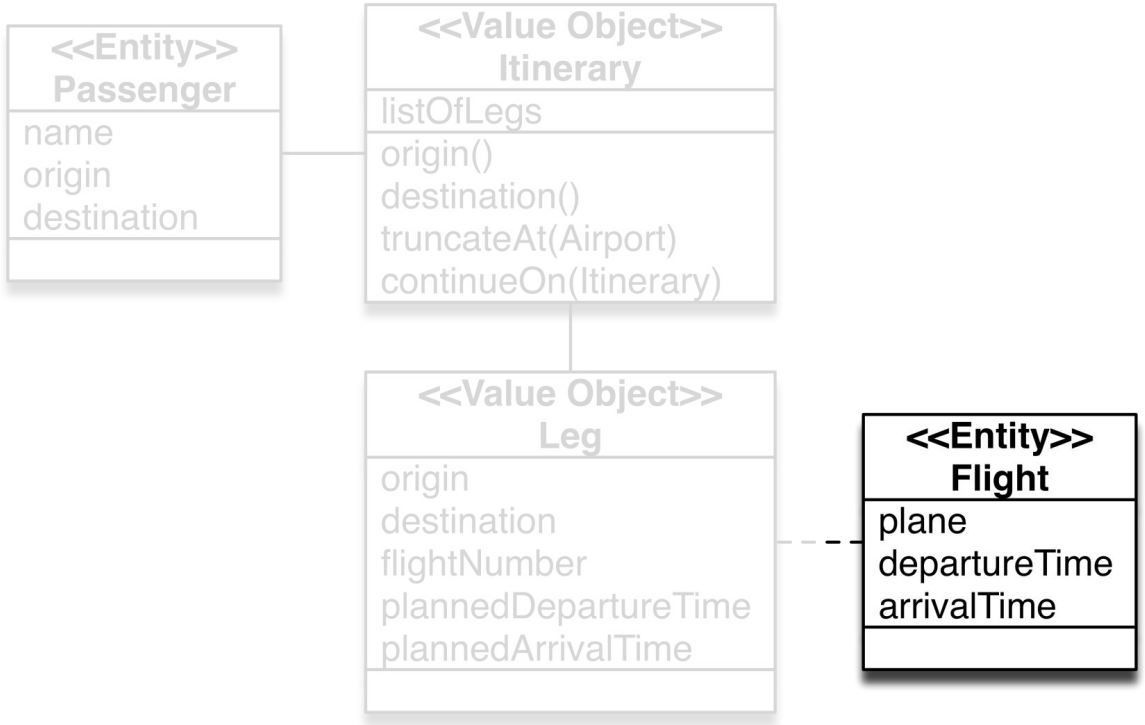
Command

An expression of intent to trigger an action in the domain.

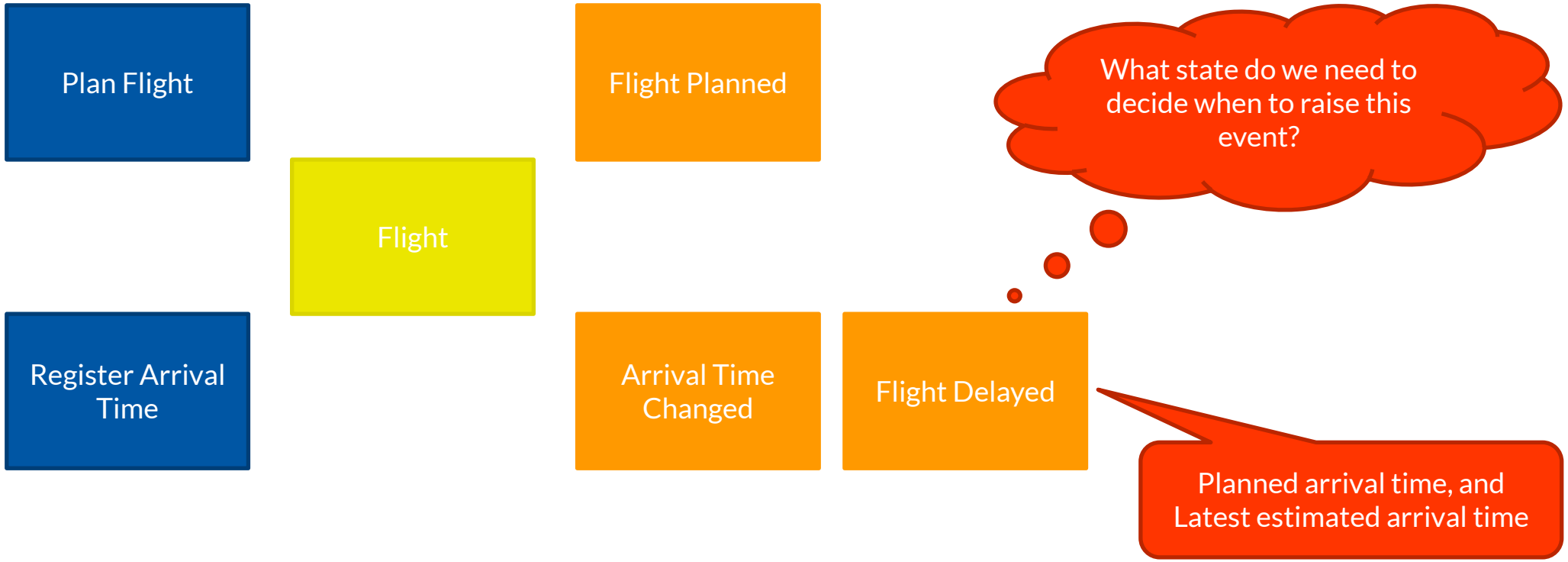
Query

A request for information or state.

Example - Command Model



Example - Command Model



Example - Query Model

- Relevant information for Passengers

FlightTimes Table

PNR	Origin	Dest.	Departure	Arrival	Layover
BGTR4	AMS	SLC	8:12	10:50	2:48
BGTR4	SLC	LAX	10:30	12:50	<null>
HGYT2	JKF	GRU	8:12	15:50	<null>

Example - Query Model

- Relevant information for Pilots

FlightTimes Table

Origin	Dest.	Departure	Arrival	Captain	Aircraft
AMS	SLC	10:50	8:12	C. Lindbergh	B747
SLC	LAX	10:30	12:50	J. Yeager	ES80
JKF	GRU	8:12	15:50	T. Cruise	A380

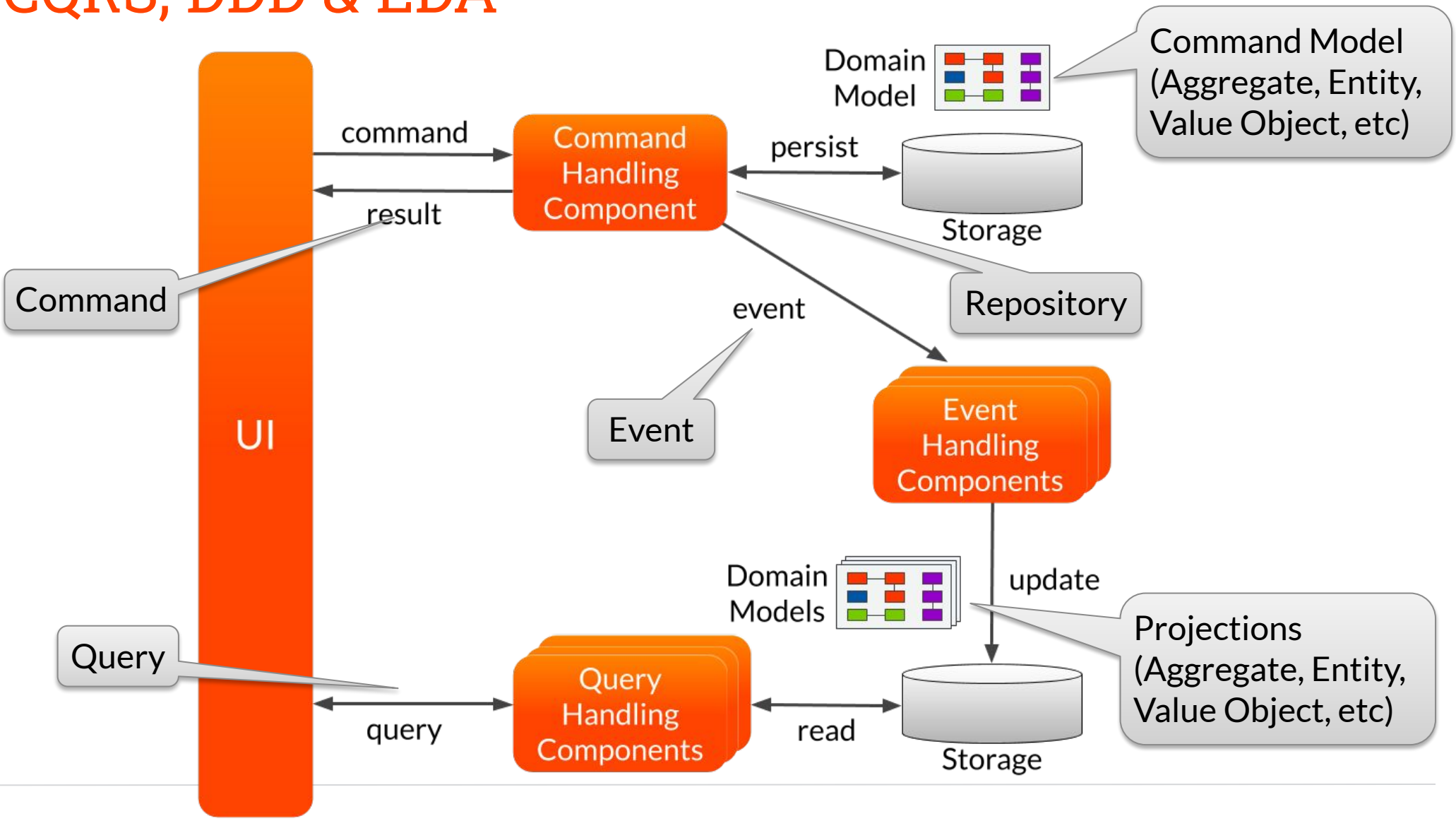
Example - Query Model

- Optimized for full-data retrieval based on ID
  - Give full Itinerary overview for PNR BGTR4

Itinerary Table

PNR	ItineraryData
BGTR4	{“passengerName”:”John Doe”, “legs” : [{“origin”: “AMS”, “dest…
YTFE4	{“passengerName”:”Mary Joe”, “legs” : [{“origin”: “SLC”, “dest…
POGH2	{“passengerName”:”Steven May”, “legs” : [{“origin”: “GRU”, “d…

CQRS, DDD & EDA

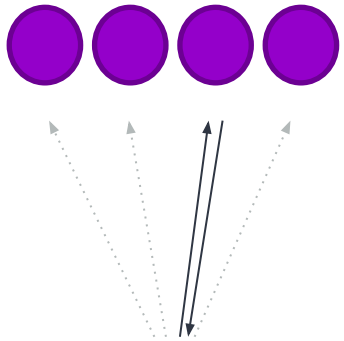

Domain Model
Command Model (Aggregate, Entity, Value Object, etc)
command
result
Command Handling Component
persist
Storage
Command
Repository
event
Event
UI
Event Handling Components
Domain Models
update
Projections (Aggregate, Entity, Value Object, etc)
Query
query
Query Handling Components
read
Storage

Interaction between components

Communication

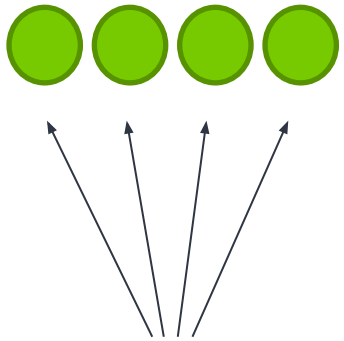
Messaging

Commands



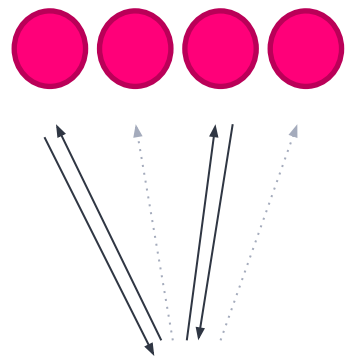
Route to single handler
Use consistent hashing
Provides confirmation/result

Events



Distribute to all logical handlers
Consumers express ordering req's
No results

Queries



Route with load balancing
Sometimes scatter/gather
Provides result

"Event" and "Message" is **not** the same thing.

Location Transparency

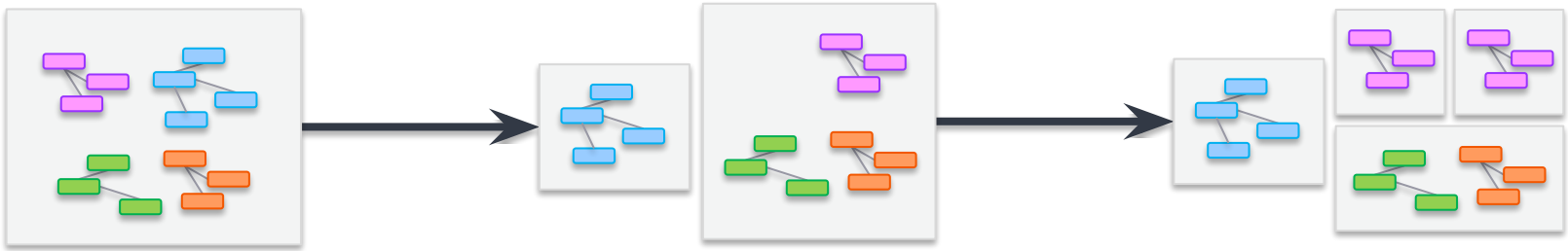
- A Component should not be aware, nor make any assumptions, of the physical location of Components it interacts with.
- Beware of APIs & method signatures:
  - Not location transparent:

```
public Result doSomething(Request request) {...}
```

  - Location transparent alternatives:

```
public void doSomething(Request request, Callback<Response> callback) {...}
public CompletableFuture<Result> doSomething(Request request) {...}
```

Location Transparency

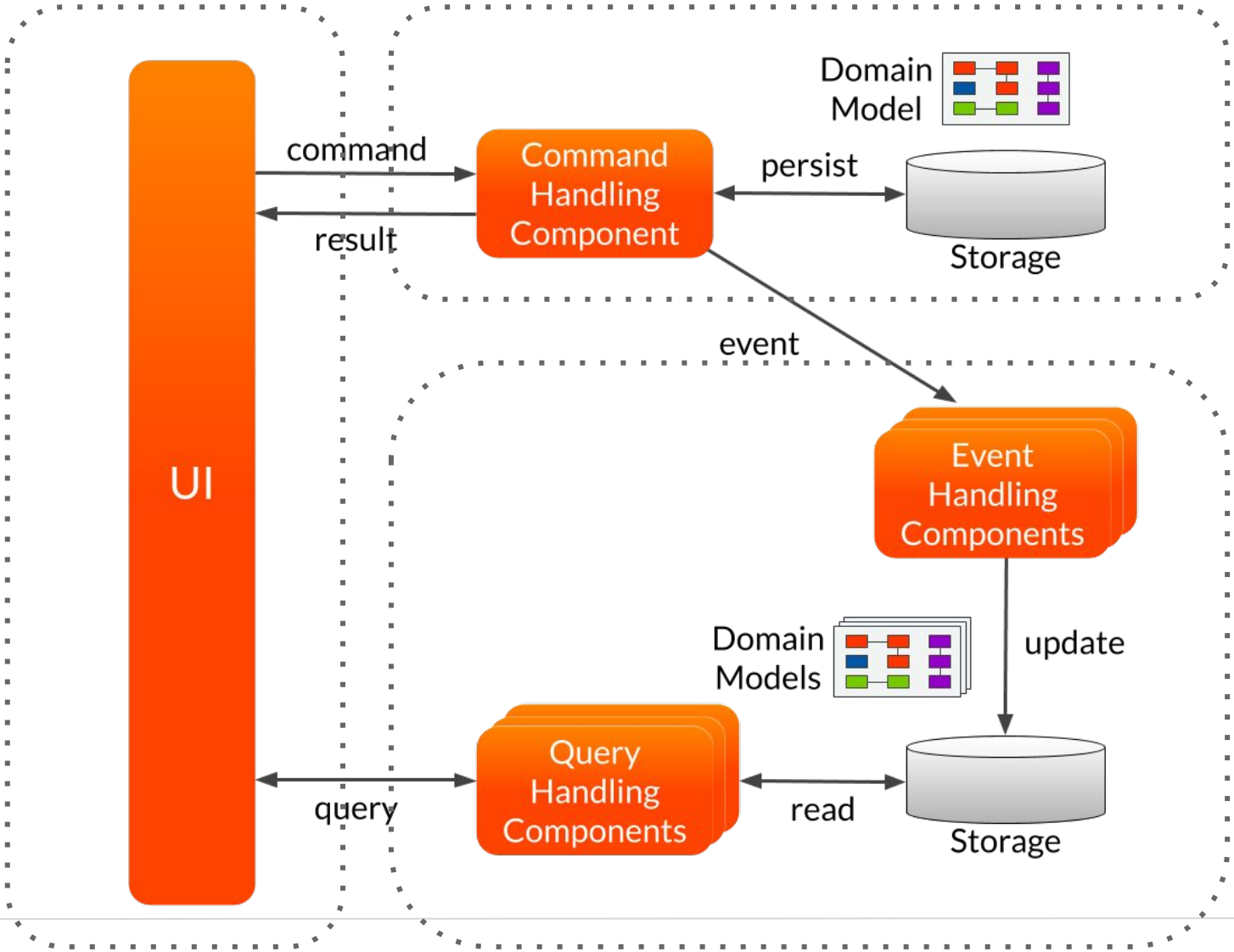


1 – Build a monolith

2 – Extract components

3 – Extract more
4 – Scale as needed

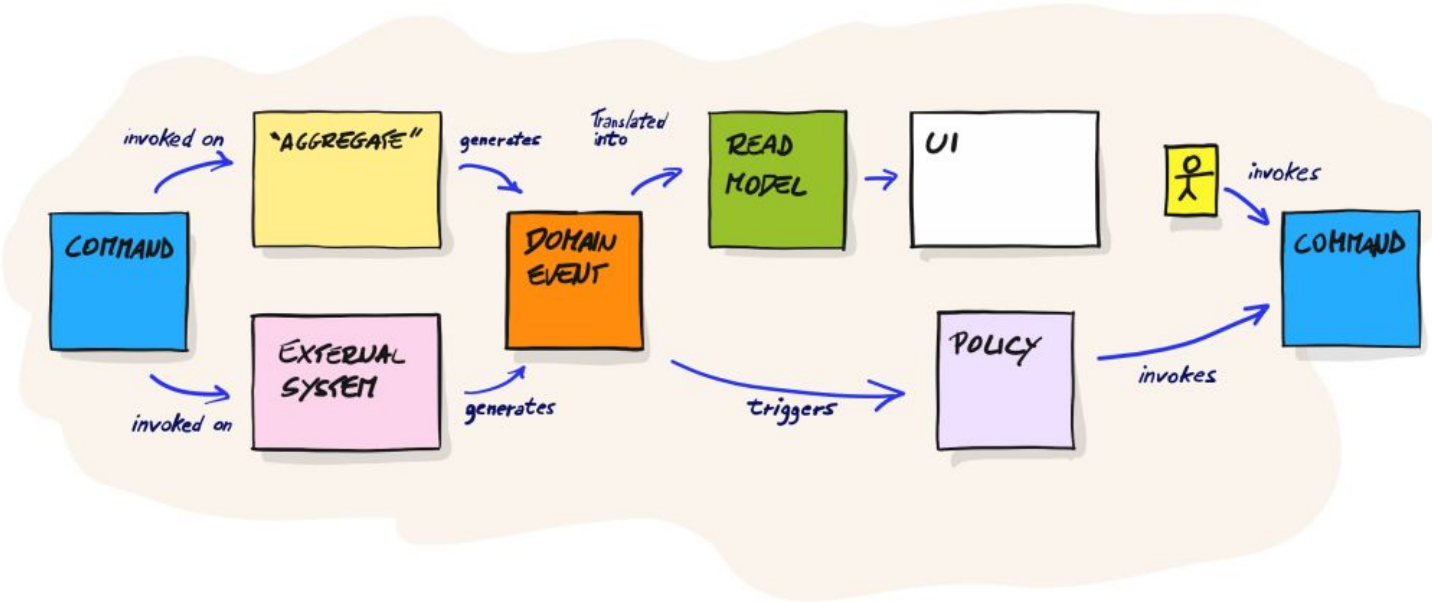
Location Transparency boundaries



Other things that are interesting to have a look at

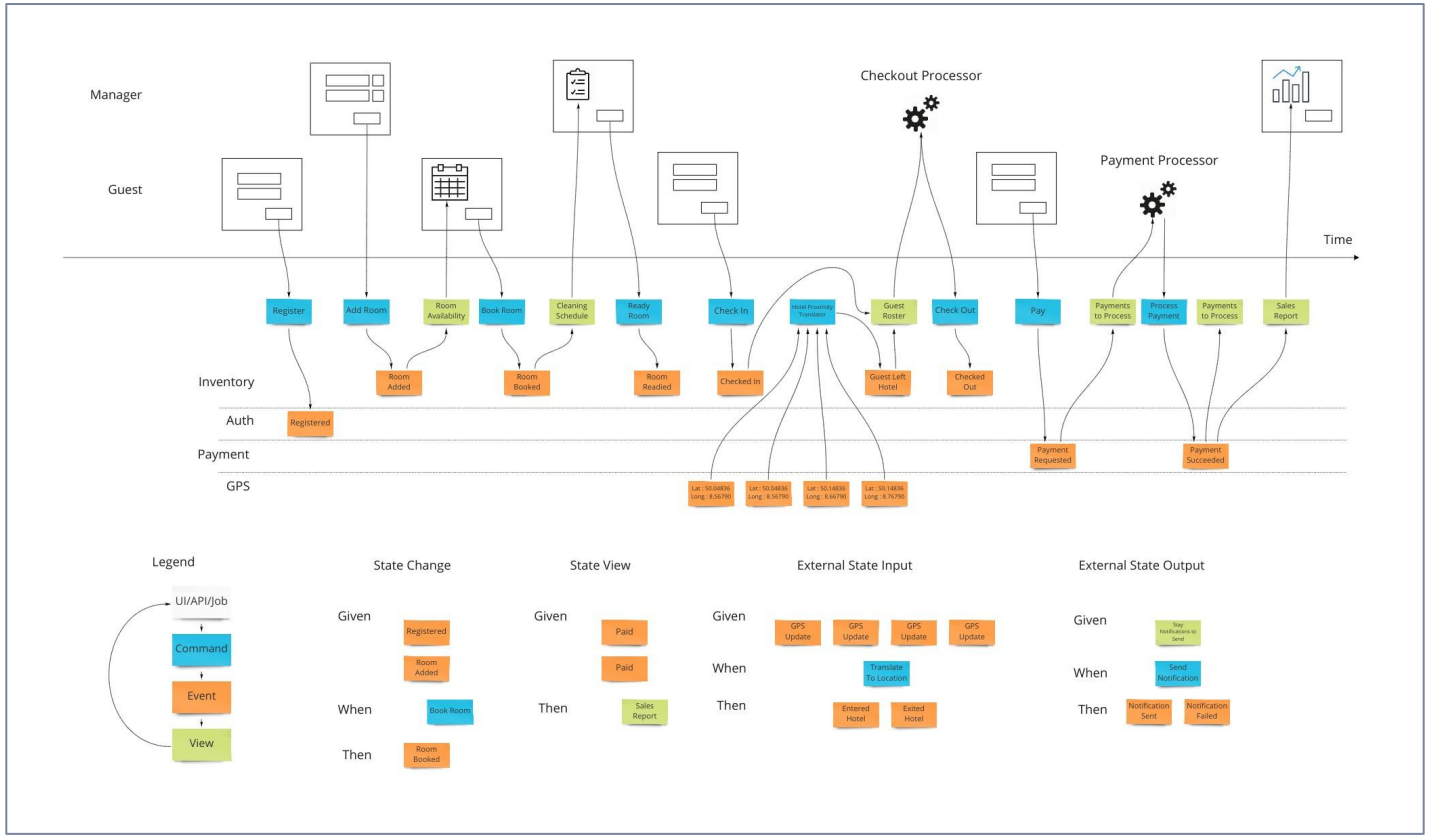
Related topics

Event Storming



AxonIQ

Event Modeling



AxonIQ

Whatever else you wanted to know...

Questions