How I Learned to Stop Worrying and Love Raw Events

Event Sourcing & CQRS with FastAPI and Celery

PyCon Athens 2025

My Journey: From Celestial Chaos to System Chaos

- **Senior Staff Engineer** @ Orfium
- **10+ years** Python experience
- **Background**: Physics → Computational Physics → Software Engineering
- **Now**: Debugging real-world system chaos
- **Philosophy**: Right architecture for each problem

The Nightmare: "Who Deleted My User?"

A real debugging story:

```python
def delete_user(user_id: int):
    db.delete_user(user_id)
```

The problem:

Monday 3:47 PM: "Sarah's account is missing!"
Tuesday 9:15 AM: "When was it deleted? Who did it? Why?"

What we can't answer:

- ❌ **When** was the user deleted?
- ❌ **Who** deleted the user?
- ❌ **Why** was it deleted?

The system has no memory of what happened

Enter Event Sourcing: The System That Remembers

We store every change as an immutable event:

```
UserDeleted(
    event_id=uuid4(),
    aggregate_id="user_123",
    revision=5,
    version=1,
    timestamp=datetime.now(),
    event_type="USER_DELETED",
    data={ "deleted_by": "admin_456", "reason": "Account closure request" }
)
```

Now we can answer everything:

- ✅ **When**: March 15, 3:47 PM
- ✅ **Who**: Admin ID 456
- ✅ **Why**: Account closure request

Every action becomes a permanent record

Core Concepts: Events

Immutable Facts

Events are immutable facts that represent state changes in the system.

Example:

User Created Event - John Doe, john@example.com, March 15

Key characteristics:

- **Immutable**: Once created, events never change
- **Facts**: They represent what actually happened
- **Complete**: Each event contains all necessary data
- **Revisioned**: Events have sequence numbers for ordering
- **Versioned**: Events have schema versions for serialization

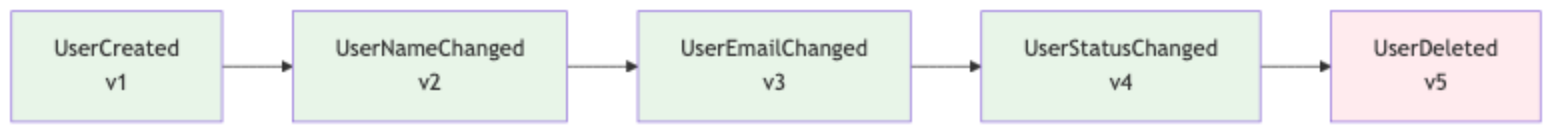
Key principle: **Events are immutable facts** - they never change

Core Concepts: Event Streams

Ordered Sequences

Event streams are ordered sequences of events for a specific aggregate.

Example:



Key characteristics:

- **Ordered**: Events have strict chronological ordering
- **Complete**: Contains the full history of an aggregate
- **Replayable**: Can rebuild any point in time
- **Source of truth**: The definitive record of what happened

The stream is the source of truth - rebuild any point in time

Core Concepts: Commands

Intent to Change

Commands represent the intent to change the system state.

Example:

"Create a new user account"

Key characteristics:

- **Intent**: They express what we want to happen
- **Validation**: Can be validated before execution
- **Idempotent**: Safe to retry if needed
- **Entry point**: The starting point for all changes

Commands are the entry point - they represent what we want to do

Core Concepts: Queries

Intent to Read (CQRS Separation)

Queries represent the intent to read data from the system.

Example:

"Show me user John Doe's profile"

Key characteristics:

- **Read-only**: They never change system state
- **Optimized**: Designed for specific read patterns
- **Separate models**: Different from command models (CQRS)
- **Fast**: Optimized for quick data retrieval

Queries are separate from commands - different models for different purposes

Core Concepts: Aggregates

Domain Logic

Aggregates contain domain logic and apply business rules to create events.

Example:

- User email must be unique
- Cannot delete already deleted user

Key characteristics:

- **Business rules**: Enforce domain-specific validation
- **State management**: Maintain current state from events
- **Event creation**: Generate new events based on commands
- **Consistency**: Ensure business invariants are maintained

Aggregates apply business logic and create events

Core Concepts: Event Store

Source of Truth

Event Store is the append-only storage for all events in the system.

Example:

User John Doe's Event Stream

- **Event 1**: User Created (March 15, 2:30 PM)
- **Event 2**: Email Changed (March 16, 10:15 AM)

Key characteristics:

- **Append-only**: Events are never modified or deleted
- **Immutable**: Once written, events are permanent
- **Stream management**: Organizes events by aggregate
- **Optimistic concurrency**: Prevents conflicting writes

Event Store is append-only - events never change or delete

Core Concepts: Projections

Building Read Models

Projections build optimized read models from events for fast querying.

Example:

- Event: User Created → Action: Create user record
- Event: Email Changed → Action: Update email field

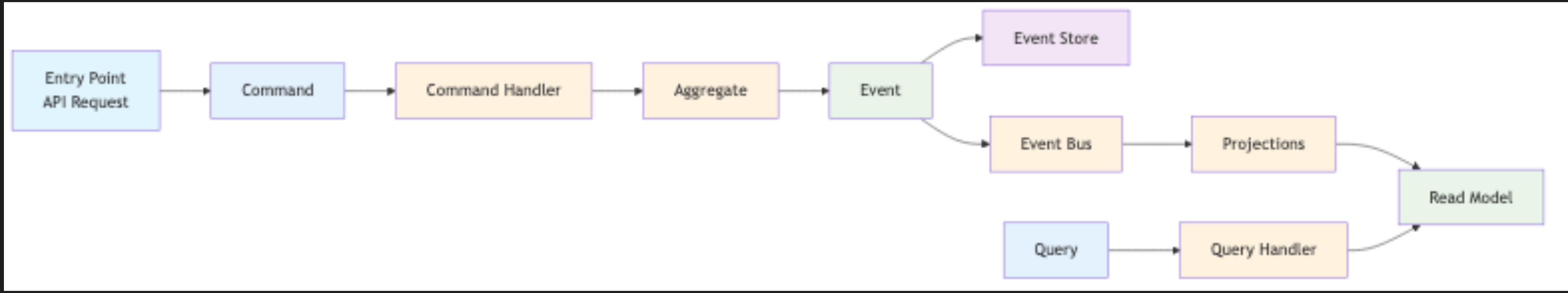
Key characteristics:

- **Event-driven**: Triggered by new events
- **Read-optimized**: Designed for specific query patterns
- **Denormalized**: Optimized for performance, not normalization
- **Eventually consistent**: Updated asynchronously

Projections update read models from events

How Everything Works Together

The complete flow:



Each interaction follows this pattern - from command to projection

FastAPI: The Command Interface

Real implementation with Pydantic:

```python
@router.post("/users")
async def create_user(
    user_data: dict,
    handler: CreateUserCommandHandler = Depends(InfraFactory.create_user_command_handler)
):
    # Create command with validation
    command = CreateUserCommand(
        name=user_data["name"],
        email=user_data["email"]
    )

    # Process command
    await handler.handle(command)

    # Return immediately (event stored successfully) or catch exceptions via middleware
    return {"user_id": command.user_id}
```

FastAPI commands accept requests and return immediately after event storage

Command Handlers: Business Logic

How we structure command processing:

```python
class CreateUserCommandHandler:
    async def handle(self, command: CreateUserCommand) -> None:
        # Retrieve all events for this aggregate
        events = await self.event_store.get_stream(command.user_id)

        # Create empty aggregate and replay events
        user = UserAggregate(command.user_id)
        for event in events:
            user.apply(event)

        # Call domain method and get new events
        new_events = user.create_user(command.name, command.email)

        # Persist and dispatch events using unit of work
        async with self.uow:
            await self.event_store.append_to_stream(command.user_id, new_events)
            await self.event_handler.dispatch(new_events)
```

Command Handler orchestrates: Event Store + Event Handler with Unit of Work

Event Handler: Celery Integration

How events are dispatched to Celery tasks:

```
class CeleryEventHandler:
    def __init__(self):
        # Map event types to Celery tasks
        self.event_handlers = {
            "USER_CREATED": [
                "process_user_created_task",
                "send_welcome_email_task"
            ],
            # ... other event types
        }

    async def dispatch(self, events: List[Event]) -> None:
        for event in events:
            for task_name in self.event_handlers[event.event_type]:
                # All tasks receive the same event payload structure
                celery_app.send_task(task_name, kwargs={"event": event.model_dump()})
```

Event Handler dispatches to message queues, Celery tasks handle messages and call projections

Celery Tasks: Event Processing

How Celery tasks process events and call projections:

```python
@app.task(name="process_user_created_task")
def process_user_created_task(event: Dict[str, Any]) -> None:
    # Convert async function to sync for Celery
    process_user_created_async_sync = async_to_sync(process_user_created_async)

    # Execute the async projection
    process_user_created_async_sync(event=event)

async def process_user_created_async(event: EventDTO) -> None:
    # Get projection and call it
    projection = InfraFactory.get_user_created_projection()
    await projection.handle(event)
```

Celery tasks are wrappers that call the appropriate projection handlers

Projections: Event-Driven Read Models

How projections build read models:

```python
class UserCreatedProjection:
    async def handle(self, event: Event) -> None:
        # Build read model from event
        user_data = {
            "aggregate_id": event.aggregate_id,
            "name": event.data.get("name"),
            "email": event.data.get("email"),
            "status": event.data.get("status"),
            "created_at": event.timestamp,
        }

        # Save to read model
        await self.db.save(user_data)
```

Projections update read models from events

FastAPI: Query Interface

How we expose read models:

```python
@users_router.get("/{user_id}/")
async def get_user(
    user_id: str,
    query_handler: GetUserQueryHandler = Depends(InfraFactory.create_get_user_query_handler)
) -> Dict[str, Any]:
    return await query_handler.handle(GetUserQuery(user_id=user_id)).model_dump()

@users_router.get("/{user_id}/{timestamp}/")
async def get_user_at_timestamp(
    user_id: str,
    timestamp: datetime = Query(..., description="ISO 8601 format: YYYY-MM-DDTHH:MM:SSZ"),
    query_handler: GetUserAtTimestampQueryHandler = Depends(InfraFactory.create_get_user_at_timestamp_query_handler)
):
    return await query_handler.handle(GetUserAtTimestampQuery(user_id=user_id, timestamp=timestamp)).model_dump()
```

FastAPI queries expose read models

Eventual Consistency: The Real Challenge

The story: "Update user's first name"

```
# User updates first name
PUT /users/123/ {"first_name": "John"}
# API returns success immediately
# But read model might not be updated yet
```

Two approaches to handle this:

1. Optimistic Updates (Naive)

2. Outbox Pattern (Advanced)

Eventual consistency requires thoughtful UI design

Performance with Snapshots

The performance challenge:

```
async def handle(self, command: CreateUserCommand) -> None:
    events = await self.event_store.get_stream(command.user_id)  # 10,000 events!
    user = UserAggregate(command.user_id)
    for event in events:
        user.apply(event)  # Takes 5 seconds 😱
```

The solution: Snapshots in Command Handler

```
async def handle(self, command: CreateUserCommand) -> None:
    try:
        snapshot = await self.snapshot_store.get_latest_snapshot(command.user_id)
        recent_events = await self.event_store.get_events_since_snapshot(command.user_id, snapshot.revision)
        # Rebuild aggregate from snapshot
        user = UserAggregate.from_snapshot(snapshot)
        for event in recent_events:
            user.apply(event)
    except SnapshotNotFound:
```

Snapshots require aggregate changes - rebuild state efficiently

Error Handling & Retries: Two Different Worlds

Commands (Synchronous) - API Failures:

```
# Unit of Work ensures atomicity
async with self.uow:
    await self.event_store.append_to_stream(user_id, new_events)
    await self.event_handler.dispatch(new_events)
# Either succeeds or fails - API gets 500
```

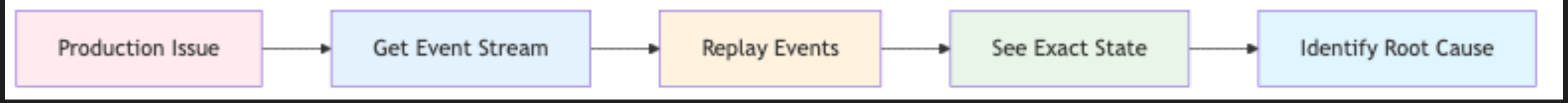
Projections (Asynchronous) - Celery Retries:

```
# Celery handles retries with late acknowledgment
@app.task(bind=True, max_retries=3, acks_late=True)
def process_user_created_task(self, event: Dict[str, Any]) -> None:
    projection.handle_user_created(event)
```

Different strategies for different failure modes

Debugging Superpowers: Testing Business Logic

The story: "What was the user's state at 3:47 PM?"




```python
# Rebuild state at specific point in time
def debug_incident(user_id: str, incident_time: datetime):
    events = event_store.get_events_before(user_id, incident_time)
    user = UserAggregate(user_id)

    # Rebuild exact state at incident time
    for event in events:
        user.apply(event)

    # Apply the problematic event that caused the issue
    user.apply(UserSuspendedEvent(user_id, reason="fraud_detected"))
```

Test business logic with real production data

Summary: Key Takeaways

Start Simple - You don't need fancy tech from day one:

- **Event Store**: PostgreSQL is sufficient for most cases
- **Event Bus**: SQS works great for most applications
- **Read Models**: PostgreSQL can handle most query patterns
- **No need for**: Kurrent, Elasticsearch, MongoDB, Kafka initially

When NOT to use Event Sourcing:

- **Simple CRUD with basic audit needs** - traditional logging suffices
- **High-frequency trading systems** - immediate consistency required
- **Teams without distributed systems experience** - steep learning curve

What you gain:

- **Complete audit trail** and time travel capabilities
- **Debugging superpowers** with real production data
- **Scalability** and eventual consistency patterns

Thank You!

Q & A

Let's Connect!

GitHub: github.com/anmarkoulis
LinkedIn: linkedin.com/in/anmarkoulis
Dev.to: dev.to/markoulis