How I Learned to Stop Worrying and Love Raw Events

Event Sourcing & CQRS with FastAPI and Celery

Antonis Markoulis | Senior Staff Engineer @ Orfium
PyCon Athens 2025

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

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

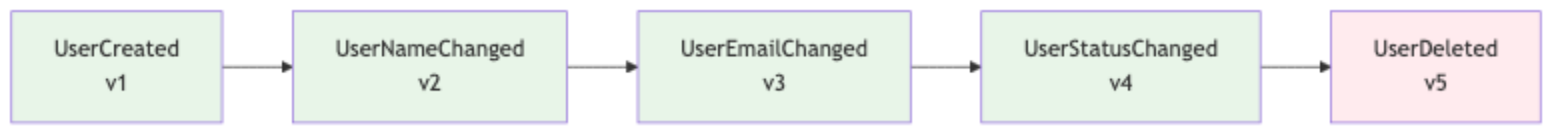
Key principle: Events are immutable facts - they never change

Core Concepts: Event Store & Streams

Source of Truth

Event Store is append-only storage where events are organized in **streams per aggregate**.

Example:



Key characteristics:

- **Append-only**: Events are never modified or deleted
- **Streams per aggregate**: Each user has their own ordered event stream
- **Immutable**: Once written, events are permanent
- **Replayable**: Can rebuild any point in time from the stream

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)

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

Aggregates apply business logic and create events

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

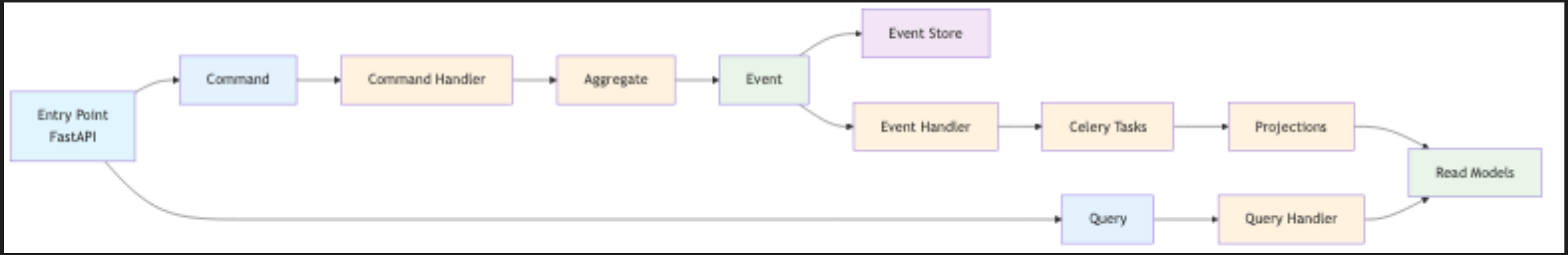
Key characteristics:

- **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:



Two paths: Commands (Write) and Queries (Read)

FastAPI: Command & Query Interface

Commands (Write) - Entry Point

```python
@router.post("/users/{user_id}/change-password/")
async def change_password(
    user_id: UUID,
    password_data: ChangePasswordDTO,
    handler: ChangePasswordCommandHandler = Depends(InfraFactory.create_change_password_command_handler)
) -> ChangePasswordResponseDTO:
    command = ChangePasswordCommand(user_id=user_id, password_data=password_data)
    await handler.handle(command)
    return ChangePasswordResponseDTO(success=True, message="Password updated successfully")
```

Queries (Read) - CQRS Separation

```python
@users_router.get("/{user_id}/")
async def get_user(
    user_id: UUID,
    query_handler: GetUserQueryHandler = Depends(InfraFactory.create_get_user_query_handler)
) -> UserReadDTO:
    return await query_handler.handle(GetUserQuery(user_id=user_id))
```


CQRS separation in action - different endpoints for different purposes

Command Handlers: Business Logic

How we structure command processing:

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

        # Create 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.change_password(command.password_data.current_password, command.password_data.new_password)

        # 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:

```python
class CeleryEventHandler(EventHandler):
    def __init__(self):
        # Map event types to Celery tasks
        self.event_handlers: Dict[EventType, List[str]] = {
            EventType.USER_CREATED: [
                "process_user_created_task",
                "send_welcome_email_task"
            ],
            EventType.USER_PASSWORD_CHANGED: [
                "process_password_changed_task",
                "send_security_notification_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

Celery Tasks: Event Processing

How Celery tasks process events and call projections:

```
@app.task(
    name="process_user_created_task",
    bind=True,
    max_retries=3,
    acks_late=True,
    autoretry_for=(Exception,),
    retry_backoff=True,
    retry_jitter=True
)
def process_user_created_task(self, event: Dict[str, Any]) -> None:
    """Celery task for processing USER_CREATED events"""
    # Get infrastructure factory
    factory = get_infrastructure_factory()

    # Get projection
    projection = factory.create_user_created_projection()

    # Process the event using async_to_sync
    async_to_sync(projection.handle)(EventDTO(**event))
    logger.info(f"Successfully processed USER_CREATED event for user {EventDTO(**event).aggregate_id}")
```

Celery tasks use async_to_sync to bridge async projections with sync Celery

Projections: Event-Driven Read Models

How projections build read models:

```
class UserCreatedProjection(Projection[UserCreatedEvent]):
    async def handle(self, event: UserCreatedEvent) -> 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

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:

```python
async def handle(self, command: ChangePasswordCommand) -> 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

```python
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)
        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:

```python
# 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
```

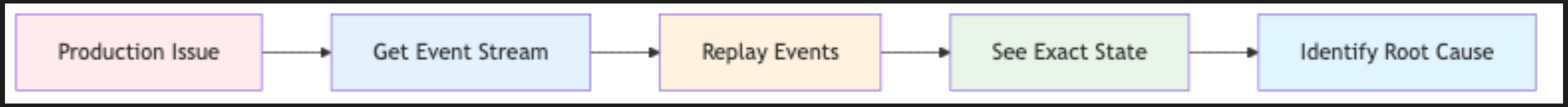
Projections (Asynchronous) - Celery Retries:

```python
# 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
# Test business logic with real production data
def test_incident_scenario(user_id: str, incident_time: datetime):
    """Test what would happen if we applied a specific event at a point in time"""
    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)

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

    # Assert the expected behavior
    assert result.is_success, f"User should be suspended: {result.error}"
    assert user.status == "suspended"
    print(f"✅ Test passed: User {user_id} would be suspended at {incident_time}")
```


Test business logic with real production data

Summary: Key Takeaways

🚀 Start Simple

- No fancy event stores needed initially
- Familiar tools, powerful results

⚠️ When NOT to use Event Sourcing

- **Simple CRUD** or **high-frequency systems** (immediate consistency needed)
- **Teams without distributed systems experience**

🎯 What you gain

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

Event sourcing + Python ecosystem = distributed systems superpowers

Thank You!

Q & A

Let's Connect!

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