How I Learned to Stop Worrying and Love Raw Events

Event Sourcing & CQRS with FastAPI and Celery

PyCon Athens 2025

1. Intro & Motivation

- Who am I?
- What are raw events? And why would anyone love them?
- The pain points of traditional architectures
- Quick teaser: what does an "audited-by-design" system look like?

Who Am I?

- **Senior Staff Engineer** at Orfium
- **10+ years** in Python and software engineering
- Built systems handling **millions of events** daily
- **Event sourcing evangelist** (recovering from traditional architectures)
- Love for **immutable data** and **audit trails**
- **Real implementation**: Built complete event sourcing systems

What Are Raw Events?

Instead of storing current state...

```
# Traditional approach - we overwrite history
user.name = "John Doe"
user.email = "john@example.com"
db.save(user)  # Previous state is lost forever
```

We store **every change** as an immutable event

```
# Event sourcing approach - we remember everything
UserCreated(user_id="123", name="John", email="john@example.com")
UserNameChanged(user_id="123", old_name="John", new_name="Johnny")
UserEmailChanged(user_id="123", old_email="john@example.com", new_email="johnny@example.com")
```

Why Would Anyone Love Raw Events?

The superpowers you get:

- **Complete audit trail** - every action is recorded
- **Time travel** - replay any point in history
- **Debugging superpowers** - see exactly what happened
- **Data integrity** - no more "lost" changes
- **Scalability** - separate read and write concerns
- **External system integration** - handle external events seamlessly

Pain Points of Traditional Architectures

What we're used to:

- **Tight coupling** between read and write
- **Poor auditability** - who changed what when?
- **Mutable state** - data corruption risks
- **Scaling challenges** - read/write conflicts
- **External system sync** - how do you handle external changes?

The result: systems that can't explain themselves

Quick Teaser: Audited-by-Design System

What does it look like?

```
Every action becomes an event:
UserCreated → UserNameChanged → UserEmailChanged → UserStatusChanged
```

Benefits:

- **Complete history** - nothing is ever lost
- **Temporal queries** - "what was the user state at 3pm?"
- **Event replay** - rebuild state from scratch
- **Audit by design** - every change is recorded
- **External system integration** - seamless external sync

2. Core Concepts

- Event Sourcing: Store every change as an immutable event
- System state = the result of replaying events
- CQRS: Separate write model (commands/events) from read model (queries)
- Benefits: auditability, modularity, scalability
- Misconception bust: You don't need Kafka to do this

Event Sourcing: The Fundamental Idea

The core equation:

> **System state = The result of replaying all events**

```
Instead of: current_state = database.get()
We have: current_state = replay_all_events()
```

Key principles:

- **Store every change** as an immutable event
- **Never update or delete** events
- **Replay events** to build current state
- **Events are the source of truth**

Event Sourcing in Practice

How it works:

1. **Events are stored** in an append-only log
2. **Aggregates apply events** to build state
3. **State is reconstructed** by replaying events
4. **Current state** = result of all applied events

Example:

```
UserCreated → UserNameChanged → UserEmailChanged
     ↓              ↓                  ↓
   State 1      State 2          Current State
```

CQRS: Command Query Responsibility Segregation

Separate concerns:

Commands (Write Model):

- Handle business logic
- Emit events
- Change system state

Queries (Read Model):

- Optimized for fast reads
- Denormalized for performance
- No business logic

Benefits:

- **Independent scaling** of read/write

CQRS Benefits

What you get:

- **Auditability** - commands emit events, queries are optimized
- **Modularity** - different models for different concerns
- **Scalability** - read/write workloads differ
- **Technology flexibility** - different DBs for different needs

Key point: You don't need Kafka to start with CQRS

Start simple, evolve as needed!

3. Architecture Walkthrough

- High-level flow: External event → queue → processing → publish → read model
- Tools & layers: Celery, FastAPI, Event Bus
- Key components: Event ingestion, event store, replay, read DB
- Design flexibility: Services + repositories, async + decoupling

High-Level Architecture Flow

The complete picture:

```
External Request → FastAPI → Command → Event Store
                                                ↓
Read Model ← Event Bus ← Celery Workers ← Event
```

Key components:

- **FastAPI**: API surface for commands/queries
- **Celery**: Async task runner, scalable workers
- **Event Store**: Append-only log (source of truth)
- **Read Model**: Optimized for queries
- **Event Bus**: Pub/sub communication

FastAPI: The Command Interface

Real implementation:

```python
@router.post("/users")
async def create_user(command: CreateUserCommand):
    # Validate and create event
    event = UserCreated(
        user_id=command.user_id,
        name=command.name,
        email=command.email
    )

    # Store event (append-only)
    await event_store.append(event)

    # Publish to event bus
    await event_bus.publish(event)

    # Return immediately (async processing)
    return {"status": "accepted", "user_id": command.user_id}
```

Celery: Async Task Runner & Scalable Workers

Event processing tasks:

```
@celery_app.task
def process_user_created(event: UserCreated):
    # Business logic
    user = User(
        id=event.user_id,
        name=event.name,
        email=event.email,
        created_at=event.timestamp
    )

    # Update read model
    read_model.save_user(user)

    # Side effects
    send_welcome_email(user.email)
    notify_analytics(user)
```

Benefits:

Event Store: The Source of Truth

Append-only operations:

```python
class EventStore:
    async def append(self, event: Event):
        # Never update or delete
        await self.db.execute("""
            INSERT INTO events (stream_id, event_type, data, version)
            VALUES ($1, $2, $3, $4)
        """, event.stream_id, event.type, event.data, event.version)

    async def get_stream(self, stream_id: str, from_version: int = 0):
        # Get all events for a stream
        return await self.db.fetch("""
            SELECT * FROM events
            WHERE stream_id = $1 AND version >= $2
            ORDER BY version
        """, stream_id, from_version)
```

Read Model: Search-Optimized Database

Optimized for fast queries:

```
class UserReadModel:
    async def save_user(self, user: User):
        # Optimized for fast reads
        await self.db.execute("""
            INSERT INTO users_view (id, name, email, status, created_at)
            VALUES ($1, $2, $3, $4, $5)
            ON CONFLICT (id) DO UPDATE SET
                name = EXCLUDED.name,
                email = EXCLUDED.email,
                status = EXCLUDED.status
        """, user.id, user.name, user.email, user.status, user.created_at)

    async def get_user(self, user_id: str) -> User:
        # Fast, simple query
        return await self.db.fetchrow(
            "SELECT * FROM users_view WHERE id = $1", user_id
        )

    async def search_users(self, query: str) -> List[User]:
        # Complex search queries
        return await self.db.fetch("""
            SELECT * FROM users_view
```

Design Flexibility: Services + Repositories

Clean separation of concerns:

```python
# Command side service
class UserCommandService:
    def __init__(self, event_store: EventStore, event_bus: EventBus):
        self.event_store = event_store
        self.event_bus = event_bus

    async def create_user(self, command: CreateUserCommand):
        # Business logic
        event = UserCreated(
            user_id=command.user_id,
            name=command.name,
            email=command.email
        )

        # Store and publish
        await self.event_store.append(event)
        await self.event_bus.publish(event)

# Query side service
class UserQueryService:
    def __init__(self, read_model: UserReadModel):
        self.read_model = read_model
```

Async + Decoupling for Scale

Benefits of this architecture:

- **High availability** - commands return immediately
- **Independent scaling** - read/write workloads differ
- **Resilience** - failures don't cascade
- **Technology flexibility** - different DBs for different needs
- **Eventual consistency** - a feature, not a bug

Real performance:

Systems process external events with proper choreography and maintain complete audit trails.

4. Real-World Patterns & Gotchas

- Eventual consistency: why it's a feature, not a bug
- Snapshots for performance on replay
- Initial backfill: bootstrapping from source APIs
- Fixes by reprocessing history — no manual data patching
- Debugging & testing in an immutable world

Eventual Consistency: Feature, Not Bug

Why it's powerful:

```
User creates account → Event stored → API returns success
                            ↓
              Event processing (async)
                            ↓
              Read model updated (eventually)
```

Benefits:

- **High availability** - API responds immediately
- **Scalability** - processing can be distributed
- **Fault tolerance** - retry on failure
- **Performance** - no blocking operations

Snapshots for Performance

The replay problem:

```
# Without snapshots - slow for long histories
def get_user_state(user_id: str, at_time: datetime):
    events = event_store.get_events(user_id, until=at_time)
    return replay_events(events)  # Could be thousands of events
```

With snapshots:

```
# With snapshots - fast state reconstruction
def get_user_state(user_id: str, at_time: datetime):
    snapshot = get_latest_snapshot(user_id, before=at_time)
    events = event_store.get_events(user_id, from_version=snapshot.version, until=at_time)
    return replay_events_from_snapshot(snapshot, events)  # Much faster
```

Initial Backfill: Bootstrapping

From external systems:

```python
@celery_app.task
def backfill_from_salesforce():
    # Get all users from Salesforce
    sf_users = salesforce_client.get_all_users()

    for sf_user in sf_users:
        # Create events for existing data
        event = UserCreated(
            user_id=sf_user.id,
            name=sf_user.name,
            email=sf_user.email,
            source="salesforce_backfill"
        )

        # Store and process
        event_store.append(event)
        process_user_created.delay(event.dict())
```

Fixes by Reprocessing History

No manual data patching:

```python
# Instead of: UPDATE users SET name = 'John' WHERE id = '123'

# We do: Reprocess all events with the fix
@celery_app.task
def reprocess_user_events(user_id: str):
    events = event_store.get_stream(user_id)

    # Clear read model
    read_model.delete_user(user_id)

    # Replay with fix
    for event in events:
        if event.type == "UserCreated":
            process_user_created.delay(event.dict())
        elif event.type == "UserNameChanged":
            process_user_name_changed.delay(event.dict())
```

Debugging in an Immutable World

What debugging looks like:

```python
# See exactly what happened
def debug_user_issue(user_id: str, timestamp: datetime):
    events = event_store.get_events(user_id, around=timestamp)

    print(f"User {user_id} events around {timestamp}:")
    for event in events:
        print(f"  {event.timestamp}: {event.type} - {event.data}")

    # Replay to see state
    state = replay_events(events)
    print(f"Final state: {state}")
```

Benefits:

- **Complete visibility** - every change is recorded
- **Time travel** - see state at any point
- **No lost data** - nothing is ever overwritten

Testing in an Immutable World

Testing strategies:

```python
# Test aggregates by applying events
def test_user_aggregate():
    user = UserAggregate()

    # Apply events
    user.apply(UserCreated(user_id="123", name="John", email="john@example.com"))
    user.apply(UserNameChanged(user_id="123", old_name="John", new_name="Johnny"))

    # Assert final state
    assert user.name == "Johnny"
    assert user.email == "john@example.com"

# Test event store
def test_event_store():
    event = UserCreated(user_id="123", name="John", email="john@example.com")

    # Store and retrieve
    event_store.append(event)
    events = event_store.get_stream("123")
```

5. Key Takeaways & Reflections

- Raw events are scary — until you realize how powerful they are
- Python + Celery + FastAPI are more than capable for serious architecture
- Event sourcing is a mindset shift, not a silver bullet — but it's fun
- The system you build today should be able to explain itself 6 months from now

Key Takeaways

The mindset shift:

- **Raw events are scary** — until you realize how powerful they are
- **Python + Celery + FastAPI** are more than capable for serious architecture
- **Event sourcing is a mindset shift**, not a silver bullet — but it's fun
- **The system you build today** should be able to explain itself 6 months from now

Start simple, evolve as needed:

1. **Begin with basic event sourcing** - store events, replay for state
2. **Add CQRS gradually** - separate read/write concerns
3. **Scale with async processing** - Celery for background work
4. **Embrace eventual consistency** - it's a feature, not a bug

Questions to Challenge Your Architecture

Before your next project, ask:

- What if I stored every change instead of just current state?
- How would I debug this if I could replay every action?
- What would complete audit trails mean for my business?
- Could I separate read and write concerns?
- What if my data was immutable?
- How would I handle external system changes?

These questions will help you think differently about your architecture.

Questions & Discussion

Let's talk about:

- **Your experiences** with event sourcing
- **Challenges** you've faced with traditional architectures
- **Questions** about implementation details
- **Next steps** for your projects

Resources:

- **Code examples**: Available on GitHub
- **Architecture docs**: Detailed implementation guide
- **Community**: Event sourcing meetups and conferences

Resources

Further Reading:

- **Event Sourcing** by Martin Fowler
- **CQRS** by Greg Young
- **Domain-Driven Design** by Eric Evans
- **Building Event-Driven Microservices** by Adam Bellemare

Tools & Libraries:

- **FastAPI** - Modern Python web framework
- **Celery** - Distributed task queue
- **Pydantic** - Data validation
- **SQLAlchemy** - Database ORM
- **PostgreSQL** - Event store & read model

Thank You!

How I Learned to Stop Worrying and Love Raw Events

Event Sourcing & CQRS with FastAPI and Celery

PyCon Athens 2025

Questions? Let's discuss!