How I Learned to Stop Worrying and Love Raw Events

Event Sourcing & CQRS with FastAPI and Celery

PyCon Athens 2024

Who Am I?

- **Staff Engineer** with 10+ years in Python
- Built systems handling **millions of events** daily
- **Event sourcing evangelist** (recovering from traditional architectures)
- Love for **immutable data** and **audit trails**

The Problem with Traditional Architectures

What we're used to:

```python
# Traditional approach
class UserService:
    def update_user(self, user_id: int, data: dict):
        user = self.db.get_user(user_id)
        user.name = data['name']  # Mutate state
        user.email = data['email']
        self.db.save(user)  # Overwrite history
```

The pain points:

- **Tight coupling** between read and write
- **Poor auditability** - who changed what when?
- **Mutable state** - data corruption risks
- **Scaling challenges** - read/write conflicts

What Are Raw Events?

Instead of storing current state...

```
# Event sourcing approach
class UserCreated:
    user_id: int
    name: str
    email: str
    timestamp: datetime

class UserNameChanged:
    user_id: int
    old_name: str
    new_name: str
    timestamp: datetime
    reason: str
```

We store every change as an immutable event

The Power of Raw Events

What happens when you store every change?

- **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

Core Concept: Event Sourcing

The fundamental idea:

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

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

Benefits:

- **Immutable history** - nothing is ever lost
- **Temporal queries** - "what was the state at 3pm?"
- **Event replay** - rebuild state from scratch
- **Audit by design** - every change is recorded

Core Concept: CQRS

Command Query Responsibility Segregation

Commands (Write Model)

```
class CreateUserCommand:
    name: str
    email: str

class ChangeUserNameCommand:
    user_id: int
    new_name: str
    reason: str
```

Queries (Read Model)

```
class UserQuery:
    def get_user_by_id(self, user_id: int) -> UserDTO:
        # Optimized for reading
        return self.read_db.get_user(user_id)
```

Why CQRS Matters

Separation of concerns:

- **Commands** - handle business logic, emit events
- **Queries** - optimized for fast, flexible reads
- **Independent scaling** - read/write workloads differ
- **Technology flexibility** - different DBs for different needs

You don't need Kafka to start!

Architecture Overview

High-level flow:

```
External Request → Command → Event Store → Event Bus → Read Model
```

Key components:

- **FastAPI** - API surface (commands + queries)
- **Celery** - async event processing
- **Event Store** - append-only event log
- **Read Model** - optimized for queries
- **Event Bus** - pub/sub communication

FastAPI: The Command Interface

```python
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI()

class CreateUserCommand(BaseModel):
    name: str
    email: str

@app.post("/users")
async def create_user(command: CreateUserCommand):
    # Validate command
    # Emit event
    # Return immediately (async)
    event = UserCreated(
        user_id=generate_id(),
        name=command.name,
        email=command.email
    )

    await event_store.append(event)
    await event_bus.publish(event)

    return {"user_id": event.user_id, "status": "processing"}
```

Celery: The Event Processing Engine

```python
from celery import Celery

celery_app = Celery('event_processor')

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

    # Update read model
    read_model.save_user(user)

    # Send welcome email
    email_service.send_welcome(event.email)

@celery_app.task
def process_user_name_changed(event: UserNameChanged):
    # Update read model
    read_model.update_user_name(event.user_id, event.new_name)

    # Notify other services
    notification_service.notify_name_change(event)
```

Event Store: The Source of Truth

```python
class EventStore:
    def __init__(self, db: Database):
        self.db = db

    async def append(self, event: Event):
        # Append-only operation
        await self.db.execute("""
            INSERT INTO events (stream_id, event_type, data, version)
            VALUES ($1, $2, $3, $4)
        """, event.stream_id, event.__class__.__name__,
             event.model_dump(), event.version)

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

        return [deserialize_event(row) for row in rows]
```

Replaying Events: Building State

```python
class UserAggregate:
    def __init__(self):
        self.id = None
        self.name = None
        self.email = None
        self.version = 0

    def apply(self, event: Event):
        if isinstance(event, UserCreated):
            self.id = event.user_id
            self.name = event.name
            self.email = event.email
        elif isinstance(event, UserNameChanged):
            self.name = event.new_name

        self.version += 1

def build_user_state(user_id: str) -> UserAggregate:
    events = event_store.get_stream(f"user-{user_id}")
    user = UserAggregate()

    for event in events:
        user.apply(event)

    return user
```

Read Model: Optimized for Queries

```python
class UserReadModel:
    def __init__(self, db: Database):
        self.db = db

    async def get_user_by_id(self, user_id: int) -> UserDTO:
        # Fast, direct query
        row = await self.db.fetchrow("""
            SELECT id, name, email, created_at, updated_at
            FROM users WHERE id = $1
        """, user_id)

        return UserDTO(**row) if row else None

    async def search_users(self, name_pattern: str) -> List[UserDTO]:
        # Optimized search
        rows = await self.db.fetch("""
            SELECT id, name, email, created_at
            FROM users
            WHERE name ILIKE $1
            ORDER BY created_at DESC
        """, f"%{name_pattern}%")

        return [UserDTO(**row) for row in rows]
```

Eventual Consistency: Feature, Not Bug

Why eventual consistency is powerful:

```python
# Command side - immediate response
@app.post("/users/{user_id}/name")
async def change_name(user_id: int, new_name: str):
    event = UserNameChanged(user_id, new_name)
    await event_store.append(event)
    await event_bus.publish(event)

    return {"status": "processing"}  # Immediate response

# Query side - eventually consistent
@app.get("/users/{user_id}")
async def get_user(user_id: int):
    # May not reflect latest changes yet
    return await read_model.get_user_by_id(user_id)
```

Benefits:

- **High availability** - system stays responsive

Performance: Snapshots

The replay problem:

```
# Without snapshots - slow for old aggregates
def build_user_state(user_id: str) -> UserAggregate:
    events = event_store.get_stream(f"user-{user_id}")  # Could be 1000s of events
    user = UserAggregate()

    for event in events:  # Expensive!
        user.apply(event)

    return user
```

With snapshots:

```
def build_user_state(user_id: str) -> UserAggregate:
    # Try to get latest snapshot
    snapshot = snapshot_store.get_latest(f"user-{user_id}")

    if snapshot:
        user = snapshot.state
```

Debugging in an Immutable World

The debugging superpowers:

```python
# See exactly what happened
async def debug_user_issue(user_id: int, timestamp: datetime):
    # Get all events for the user
    events = await event_store.get_stream(f"user-{user_id}")

    # Replay to any point in time
    user_state = UserAggregate()
    for event in events:
        if event.timestamp <= timestamp:
            user_state.apply(event)
        else:
            break

    return user_state

# Compare states at different times
state_3pm = await debug_user_issue(user_id, datetime(2024, 1, 1, 15, 0))
state_4pm = await debug_user_issue(user_id, datetime(2024, 1, 1, 16, 0))
```

Fixing Data: Reprocessing History

Instead of manual data patches:

```
# Traditional approach - scary!
UPDATE users SET name = 'correct_name' WHERE id = 123;

# Event sourcing approach - safe!
@celery_app.task
def fix_user_name(user_id: int, correct_name: str):
    # Emit correction event
    event = UserNameCorrected(
        user_id=user_id,
        old_name=get_current_name(user_id),
        new_name=correct_name,
        reason="Data correction"
    )

    await event_store.append(event)
    await event_bus.publish(event)

    # All projections will be updated automatically
```

Real-World Gotchas

Common challenges and solutions:

1. Event Schema Evolution

```
# Version your events
class UserCreatedV2:
    user_id: int
    name: str
    email: str
    phone: str  # New field
    version: int = 2
```

2. Event Ordering

```
# Use optimistic concurrency
async def append_event(event: Event):
    expected_version = event.version
    actual_version = await get_current_version(event.stream_id)
    if expected_version != actual_version:
```

Scaling Patterns

Horizontal scaling strategies:

1. Event Store Partitioning

```
# Partition by stream_id
stream_id = f"user-{user_id}"
partition = hash(stream_id) % num_partitions
```

2. Read Model Sharding

```
# Shard by user_id
shard_id = user_id % num_shards
read_db = get_shard_connection(shard_id)
```

3. Celery Worker Scaling

```
# Different worker pools for different event types
@celery_app.task(queue='user_events')
def process_user_event(event):
```

Testing Event-Sourced Systems

Testing strategies:

```python
class TestUserAggregate:
    def test_user_creation(self):
        # Given
        events = [
            UserCreated(user_id=1, name="John", email="john@example.com")
        ]

        # When
        user = UserAggregate()
        for event in events:
            user.apply(event)

        # Then
        assert user.name == "John"
        assert user.email == "john@example.com"

class TestEventStore:
    async def test_event_append_and_retrieve(self):
        # Given
        event = UserCreated(user_id=1, name="John", email="john@example.com")

        # When
        await event_store.append(event)
        retrieved_events = await event_store.get_stream("user-1")
```

Migration Strategy

How to introduce event sourcing:

Phase 1: Dual Write

```
# Write to both old and new systems
async def create_user(command: CreateUserCommand):
    # Old way
    await old_db.create_user(command)

    # New way
    event = UserCreated(user_id=generate_id(), **command.dict())
    await event_store.append(event)
    await event_bus.publish(event)
```

Phase 2: Read from New

```
# Switch reads to new system
async def get_user(user_id: int):
    return await read_model.get_user_by_id(user_id)
```

Key Takeaways

What you should remember:

1. **Raw events are powerful** - complete audit trail, time travel, debugging superpowers
2. **Python ecosystem is ready** - FastAPI + Celery + async/await = perfect combination
3. **Start simple** - you don't need Kafka or complex infrastructure to begin
4. **Event sourcing is a mindset** - think in terms of "what happened" not "what is"
5. **Your system should explain itself** - 6 months from now, you'll thank yourself

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 it mean to have complete audit trails?
- Could I separate my read and write concerns?
- What if my data was immutable?

Thank You!

Questions & Discussion

Slides & Code: [GitHub Link]

Twitter: @yourhandle

Email: your.email@example.com

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

Demo Time!

Live Event Sourcing Demo

Let's build a simple user management system with:

- FastAPI for the API
- Celery for event processing
- SQLite for event store
- Real-time event replay

Code: [Demo Repository Link]