Event-Driven Architecture Project Structure

Asynchronous event-based architecture with Python. Message queues, event handlers, and optional event sourcing.

#event-driven #python #architecture #async #cqrs #messaging

PNG PDF Copy Prompt

Project Directory

```
myproject/
    pyproject.toml
    docker-compose.yml  RabbitMQ, Redis…
    README.md
    .gitignore
    .env.example
  src/
    myproject/
      __init__.py
      domain/  Core domain log…
        __init__.py
        events/  Domain event de…
          __init__.py
          base.py  Base event class
          user_events.py
          order_events.py
        commands/  Command definit…
          __init__.py
          user_commands.py
          order_commands.py
        models/
          __init__.py
          user.py
          order.py
      application/  Application ser…
        __init__.py
        command_handlers/  Handle commands…
          __init__.py
          user_handlers.py
          order_handlers.py
        event_handlers/  React to domain…
          __init__.py
          notification_handler.py
          projection_handler.py
          integration_handler.py
        queries/  Read-side queri…
          __init__.py
          user_queries.py
          order_queries.py
      infrastructure/  Technical imple…
        __init__.py
        messaging/  Message broker …
          __init__.py
          event_bus.py  Event publisher
          rabbitmq.py  RabbitMQ adapter
          consumer.py  Message consumer
        persistence/
          __init__.py
          event_store.py  Event storage
          read_models.py  Query projectio…
          database.py
        api/  HTTP interface
          __init__.py
          main.py  FastAPI app
          routes/
            __init__.py
            commands.py  Command endpoin…
            queries.py  Query endpoints
      workers/  Background proc…
        __init__.py
        event_processor.py  Main worker ent…
        retry_handler.py  Failed event re…
      config/
        __init__.py
        settings.py
        container.py  DI wiring
  tests/
    __init__.py
    conftest.py
    unit/
      __init__.py
      test_command_handlers.py
      test_event_handlers.py
    integration/
      __init__.py
      test_event_flow.py
```

Why This Structure?

Event-driven architecture decouples producers from consumers. Commands trigger state changes that emit domain events. Event handlers react asynchronously—sending notifications, updating read models, or triggering integrations. This enables loose coupling and horizontal scaling.

Key Directories

`domain/events/` - Immutable event classes representing what happened

`application/command_handlers/` - Process commands, persist state, emit events

`application/event_handlers/` - React to events: notifications, projections, integrations

`workers/` - Background processes consuming from message queues

Event Definition and Handler

```python
# domain/events/order_events.py
@dataclass(frozen=True)
class OrderCreated(DomainEvent):
    order_id: str
    user_id: str
    total: Decimal

# application/event_handlers/notification_handler.py
async def handle_order_created(event: OrderCreated):
    await email_service.send_confirmation(event.user_id)
    await slack.notify_sales_channel(event.order_id)
```

When To Use This

- Operations can happen asynchronously
- Multiple systems need to react to the same event
- Need audit trail of all state changes
- Read and write workloads scale differently
- Building reactive, loosely coupled systems

CQRS Pattern

`Commands` - Write operations that change state and emit events

`Queries` - Read from optimized projections, not event store

`Projections` - Event handlers that build read-optimized views

Trade-offs

`Eventual consistency` - Read models may lag behind writes

`Complexity` - More moving parts: queues, workers, projections

`Debugging harder` - Async flows harder to trace than sync calls

Testing Strategy

`Unit tests` - Test handlers with in-memory event bus

`Integration` - Test full event flow with real message broker

`Event replay` - Rebuild projections from stored events

Best Practices

- Events are immutable—never modify published events
- Include all necessary data in events (avoid lookups)
- Make event handlers idempotent (may receive duplicates)
- Use correlation IDs to trace event chains
- Version your events for schema evolution

Naming Conventions

`Events` - Past tense: `OrderCreated`, `PaymentProcessed`

`Commands` - Imperative: `CreateOrder`, `ProcessPayment`

`Handlers` - `handle_{event_name}` or `{Event}Handler`