Pragmatic DDD with Python

AI will love your code

by John Macias

— — — — —

Chapter 2: The Three Pillars

Chapter 2: The Three Pillars

This book combines three architectural concepts:

1. **Domain-Driven Design (DDD)** — Where to put business logic
2. **Hexagonal Architecture** — How to structure the code
3. **CQRS** — How to separate reads from writes

Each solves a different problem. Together, they create a coherent system.

— — — — —

Pillar 1: DDD — Business Logic in the Domain

DDD answers the question: *Where does business logic live?*

Answer: In the **domain layer**.

Not in views. Not in SQLAlchemy models. Not scattered across services. In a dedicated layer that knows nothing about HTTP, databases, or frameworks.

The Domain Layer Contains:

Entities — Objects with identity that persist over time.

```python
@dataclass
class Booking:
    id: BookingId
    client_id: ClientId
    restaurant_id: RestaurantId
    time_slot: TimeSlot
    party_size: PartySize
    status: BookingStatus
    confirmed_at: Optional[datetime] = None
    _events: list = field(default_factory=list)

    def confirm(self) -> None:
        if self.status != BookingStatus.PENDING:
            raise BookingCannotBeConfirmed(self.id)

        self.status = BookingStatus.CONFIRMED
        self.confirmed_at = datetime.utcnow()

        self._record_event(BookingConfirmed(self.id))
    def _record_event(self, event: DomainEvent) -> None:
        self._events.append(event)
```

Value Objects — Immutable objects defined by their attributes.

```python
@dataclass(frozen=True)
class TimeSlot:
    date: date
    hour: int
    minute: int

    def __post_init__(self):
        if self.hour < 0 or self.hour > 23:
            raise InvalidTimeSlot("Hour must be between 0 and 23")
        if self.minute < 0 or self.minute > 59:
            raise InvalidTimeSlot("Minute must be between 0 and 59")

    def is_before(self, other: "TimeSlot") -> bool:
        return self.to_datetime() < other.to_datetime()

    def to_datetime(self) -> datetime:
        return datetime.combine(self.date, time(self.hour, self.minute))
```

Domain Events — Records of things that happened.

```python
@dataclass(frozen=True)
class BookingConfirmed:
    booking_id: BookingId
    occurred_at: datetime = field(default_factory=datetime.utcnow)
```

Domain Services — Logic that doesn't belong to any single entity.

```python
class BookingAvailabilityChecker:
    def __init__(self, booking_repository: BookingRepositoryInterface):
        self._booking_repository = booking_repository

    def is_available(
        self,
        restaurant_id: RestaurantId,
        time_slot: TimeSlot,
        party_size: PartySize
    ) -> bool:
        existing_bookings = self._booking_repository.find_active_by_restaurant_and_time_slot(
            restaurant_id, time_slot
        )

        # Business logic for availability
        return existing_bookings.total_party_size() + party_size.value <= 50
```

What the Domain Layer Does NOT Contain:

- Database queries

- HTTP concerns
- Framework dependencies
- Email sending
- External API calls

The domain layer is pure business logic. It could run without FastAPI or SQLAlchemy.

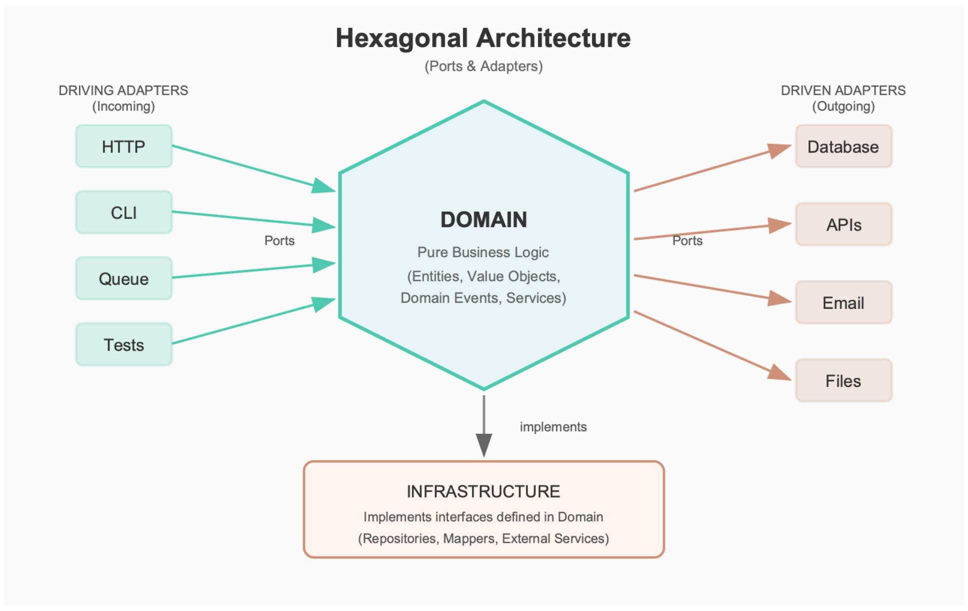
— — — — —

Pillar 2: Hexagonal Architecture — Ports & Adapters

Hexagonal Architecture (also called Ports & Adapters) answers: *How do I isolate my business logic?*

The Core Concept

Imagine your application as a hexagon:


Hexagonal Architecture
(Ports & Adapters)
DRIVING ADAPTERS (Incoming)
HTTP
CLI
Queue
Tests
Ports
DOMAIN
Pure Business Logic
(Entities, Value Objects, Domain Events, Services)
Ports
DRIVEN ADAPTERS (Outgoing)
Database
APIs
Email
Files
implements
INFRASTRUCTURE
Implements interfaces defined in Domain
(Repositories, Mappers, External Services)

Hexagonal Architecture diagram showing the domain at the center of a hexagon, with incoming adapters (HTTP, CLI, Queue, Tests) on the left side and outgoing adapters (Database, External APIs, File System) on the bottom

- **Inside the hexagon**: Your domain logic (pure business rules)
- **Ports**: Interfaces that define how the outside world interacts
- **Adapters**: Implementations that connect to specific technologies

Ports: The Interfaces

Ports are interfaces defined in your domain:

```python
# This is a PORT — it's in the Domain layer
from abc import ABC, abstractmethod
from typing import Optional

class BookingRepositoryInterface(ABC):
    @abstractmethod
    def save(self, booking: Booking) -> None:
        pass

    @abstractmethod
    def find_by_id(self, id: BookingId) -> Optional[Booking]:
        pass

    @abstractmethod
    def find_active_by_restaurant(self, id: RestaurantId) -> BookingCollection:
        pass
```

The domain knows it needs to save and retrieve bookings. It doesn't know how.

Adapters: The Implementations

Adapters live in the infrastructure layer and implement the ports:

```python
# This is an ADAPTER — it's in the Infrastructure layer
class SQLAlchemyBookingRepository(BookingRepositoryInterface):
    def __init__(self, session: Session):
        self._session = session
        self._mapper = BookingMapper()

    def save(self, booking: Booking) -> None:
        model = self._session.query(BookingModel).filter(
            BookingModel.id == str(booking.id)
        ).first()

        if model is None:
            model = BookingModel()

        self._mapper.to_model(booking, model)
        self._session.add(model)
        self._session.commit()

    def find_by_id(self, id: BookingId) -> Optional[Booking]:
        model = self._session.query(BookingModel).filter(
            BookingModel.id == str(id)
        ).first()

        return self._mapper.to_domain(model) if model else None
```

Why This Matters

You can swap adapters without changing business logic:

- Today: PostgreSQL via SQLAlchemy
- Tomorrow: MongoDB via Motor
- Testing: In-memory fake repository

```python
# Production
repository = SQLAlchemyBookingRepository(session)

# Testing
repository = InMemoryBookingRepository()

# The domain code is identical in both cases
booking = repository.find_by_id(booking_id)
booking.confirm()
repository.save(booking)
```

Your domain is protected from infrastructure changes.

— — — — —

Pillar 3: CQRS — Separating Reads from Writes

CQRS (Command Query Responsibility Segregation) answers: *How do I handle the different needs of reading and writing?*

The Problem

Reading and writing have different requirements:

Writing needs:

- Validation
- Business rules
- Transactions
- Event publishing
- Consistency

Reading needs:

- Speed

- Flexibility
- Joins across multiple tables
- Aggregations
- Pagination

Trying to use the same model for both creates compromises.

The Solution: Split Them

Commands change state:

```
@dataclass
class CreateBookingCommand:
    booking_id: BookingId
    client_id: ClientId
    restaurant_id: RestaurantId
    time_slot: TimeSlot
    party_size: PartySize
```

Queries read state:

```
@dataclass
class GetBookingByIdQuery:
    booking_id: BookingId
```

Commands Never Return Domain Data

This is a mindset shift. Commands don't return the created entity:

```
# Wrong thinking
booking = command_bus.dispatch(CreateBookingCommand(...))
return {"booking": booking}  # What to return?

# Right thinking
booking_id = BookingId.generate()  # Generate ID first
command_bus.dispatch(CreateBookingCommand(booking_id, ...))
return {"id": str(booking_id)}  # Return ID
```

The ID exists before the command. The command ensures persistence. You already have what you need.

Queries Can Be Optimized Independently

Since queries are separate, you can:

- Use raw SQL for complex reports

- Join tables from different domains
- Cache aggressively
- Use read replicas

The key is that database access still goes through a **Query Repository**—a specialized repository optimized for read operations:

```python
class GetBookingListHandler:
    def __init__(self, query_repository: BookingQueryRepositoryInterface):
        self._query_repository = query_repository

    def handle(self, query: GetBookingListQuery) -> BookingListDto:
        return self._query_repository.find_booking_list(
            restaurant_id=query.restaurant_id,
            page=query.page
        )
```

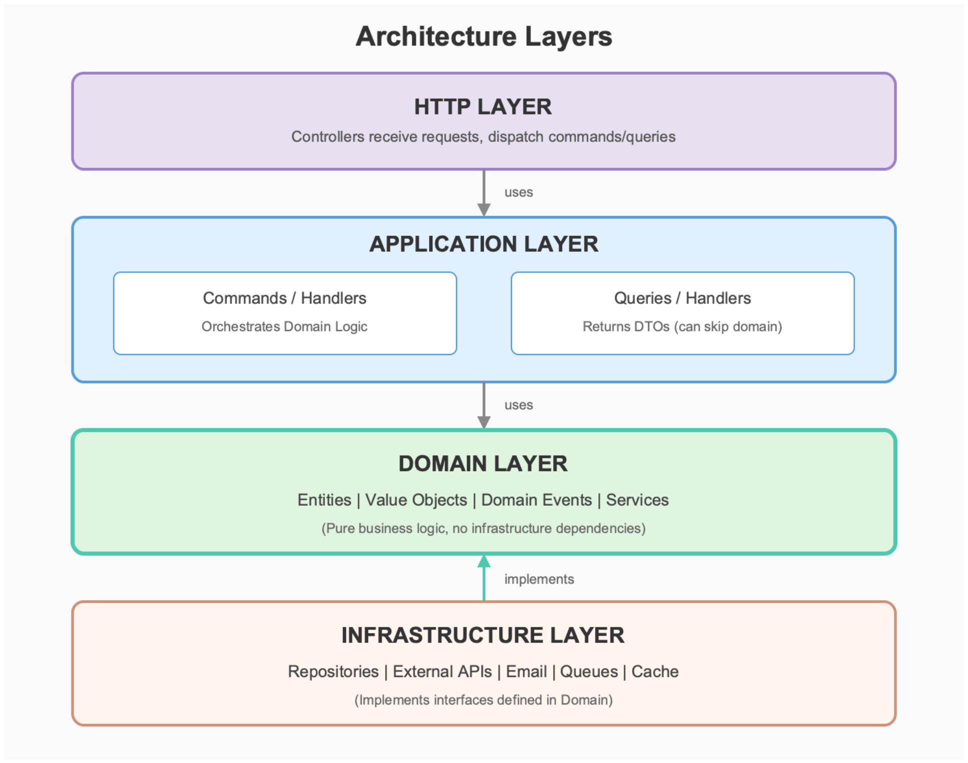
The repository implementation contains the optimized SQL:

```python
class SQLAlchemyBookingQueryRepository(BookingQueryRepositoryInterface):
    def __init__(self, session: Session):
        self._session = session

    def find_booking_list(
        self,
        restaurant_id: RestaurantId,
        page: int
    ) -> BookingListDto:
        query = text("""
            SELECT
                b.id,
                b.date,
                b.status,
                c.name as client_name,
                r.name as restaurant_name
            FROM bookings b
            JOIN clients c ON b.client_id = c.id
            JOIN restaurants r ON b.restaurant_id = r.id
            WHERE b.restaurant_id = :restaurant_id
            ORDER BY b.date DESC
            LIMIT 20 OFFSET :offset
        """)

        results = self._session.execute(
            query,
            {"restaurant_id": str(restaurant_id), "offset": (page - 1) * 20}
        ).fetchall()

        return BookingListDto.from_query_results(results)
```

This keeps handlers clean and testable while allowing optimized reads. The repository can use raw SQL, SQLAlchemy ORM, or any other approach—handlers don't care.

— — — — —

How The Three Pillars Fit Together


Architecture Layers
HTTP LAYER
Controllers receive requests, dispatch commands/queries
uses
APPLICATION LAYER
Commands / Handlers
Orchestrates Domain Logic
Queries / Handlers
Returns DTOs (can skip domain)
uses
DOMAIN LAYER
Entities | Value Objects | Domain Events | Services
(Pure business logic, no infrastructure dependencies)
implements
INFRASTRUCTURE LAYER
Repositories | External APIs | Email | Queues | Cache
(Implements interfaces defined in Domain)

Four-layer architecture diagram showing HTTP Layer at top, Application Layer with Commands and Queries in the middle, Domain Layer with Entities and Value Objects below, and Infrastructure Layer with Repositories and External Services at the bottom

1. **HTTP Layer** receives a request
2. **Application Layer** dispatches a command or query
3. **Command handlers** use **Domain** entities and **Infrastructure** repositories
4. **Query handlers** use **Query Repositories** for optimized reads
5. **Domain** contains business logic, isolated from everything else
6. **Infrastructure** implements the technical details (including repositories)

— — — — —

A Concrete Example

Let's trace a booking confirmation through all three pillars:

HTTP Layer (FastAPI)

```
@router.post("/bookings/{booking_id}/confirm")
async def confirm_booking(
    booking_id: str,
    command_bus: CommandBus = Depends(get_command_bus)
) -> dict:
    command_bus.dispatch(ConfirmBookingCommand(
        booking_id=BookingId.from_string(booking_id)
    ))

    return {"status": "confirmed"}
```

Application Layer (CQRS)

```
class ConfirmBookingHandler:
    def __init__(
        self,
        repository: BookingRepositoryInterface,
        event_bus: EventBus
    ):
        self._repository = repository
        self._event_bus = event_bus

    def handle(self, command: ConfirmBookingCommand) -> None:
        booking = self._repository.find_by_id(command.booking_id)

        if booking is None:
            raise BookingNotFound(command.booking_id)

        booking.confirm()  # Domain logic

        self._repository.save(booking)
        self._event_bus.publish(booking.pull_events())
```

Domain Layer (DDD)

```
@dataclass
class Booking:
    id: BookingId
    status: BookingStatus
    confirmed_at: Optional[datetime] = None
    _events: list = field(default_factory=list)

    def confirm(self) -> None:
        if self.status != BookingStatus.PENDING:
            raise BookingCannotBeConfirmed(self.id)

        self.status = BookingStatus.CONFIRMED
        self.confirmed_at = datetime.utcnow()

        self._record_event(BookingConfirmed(self.id))

    def _record_event(self, event: DomainEvent) -> None:
```

```python
        self._events.append(event)

    def pull_events(self) -> list:
        events = self._events.copy()
        self._events.clear()
        return events
```

Infrastructure Layer (Hexagonal)

```python
class SQLAlchemyBookingRepository(BookingRepositoryInterface):
    def __init__(self, session: Session):
        self._session = session
        self._mapper = BookingMapper()

    def save(self, booking: Booking) -> None:
        model = self._session.query(BookingModel).filter(
            BookingModel.id == str(booking.id)
        ).first()

        model.status = booking.status.value
        model.confirmed_at = booking.confirmed_at
        self._session.commit()
```

Each layer has one job. Each pillar contributes its strength.

— — — — —

Summary

Pillar	Question It Answers	Key Concept
DDD	Where does business logic live?	In the domain layer
Hexagonal	How do I isolate business logic?	Ports and adapters
CQRS	How do I handle reads vs writes?	Separate commands and queries

The next chapter explores why this architecture matters—not just for code quality, but for team productivity, AI assistance, and long-term maintainability.

Enjoyed this chapter?

The complete book includes 38 chapters covering DDD Building Blocks, CQRS, Hexagonal Architecture, Testing, Bounded Contexts, Event Sourcing, and AI-assisted development patterns.

Get the full book at:

www.pragmaticddd.com

— — — — —

Promotional Price - $9.99