FREE SAMPLE

Pragmatic DDD with Laravel

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 controllers. Not in Eloquent 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.

```
final class Booking extends Entity
{
    public function confirm(): void
    {
        if ($this->status !== BookingStatus::Pending) {
            throw new BookingCannotBeConfirmed($this->id);
        }

        $this->status = BookingStatus::Confirmed;
        $this->confirmedAt = new DateTimeImmutable();

        $this->recordEvent(new BookingConfirmed($this->id));
    }
}
```

Value Objects — Immutable objects defined by their attributes.

```php
final readonly class TimeSlot
{
    public function __construct(
        public DateTimeImmutable $date,
        public int $hour,
        public int $minute
    ) {
        if ($hour < 0 || $hour > 23) {
            throw new InvalidTimeSlot('Hour must be between 0 and 23');
        }
    }

    public function isBefore(TimeSlot $other): bool
    {
        return $this->toDateTime() < $other->toDateTime();
    }
}
```

Domain Events — Records of things that happened.

```php
final readonly class BookingConfirmed implements DomainEvent
{
    public function __construct(
        public BookingId $bookingId,
        public DateTimeImmutable $occurredAt
    ) {}
}
```

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

```php
final readonly class BookingAvailabilityChecker
{
    public function __construct(
        private BookingRepositoryInterface $bookingRepository
    ) {}

    public function isAvailable(
        RestaurantId $restaurantId,
        TimeSlot $timeSlot,
        PartySize $partySize
    ): bool {
        $existingBookings = $this->bookingRepository
            ->findActiveByRestaurantAndTimeSlot($restaurantId, $timeSlot);

        // Business logic for availability
        return $existingBookings->totalPartySize() + $partySize->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 Laravel.

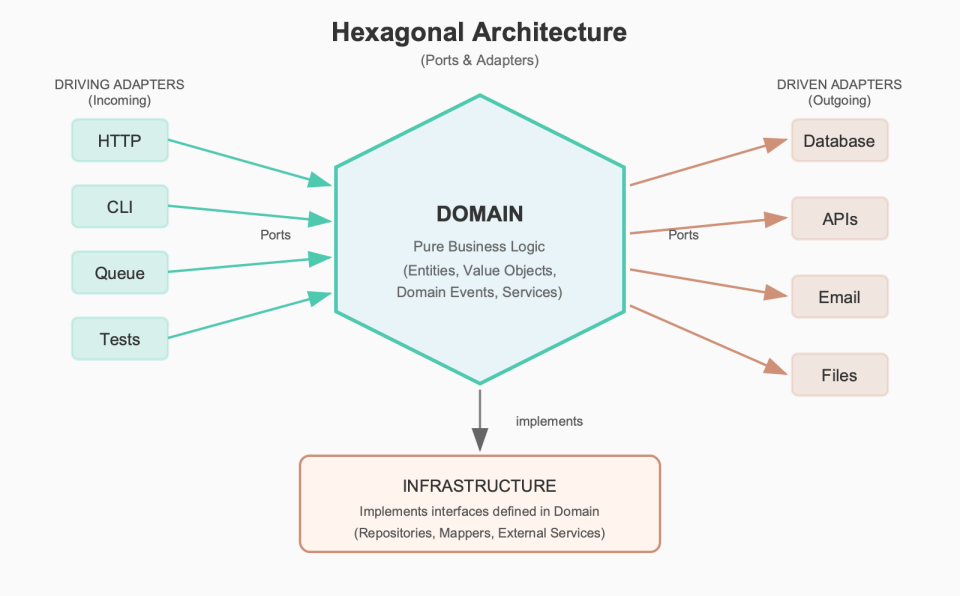
— — — — —

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 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:

```php
// This is a PORT — it's in the Domain layer
interface BookingRepositoryInterface
{
    public function save(Booking $booking): void;
    public function findById(BookingId $id): ?Booking;
    public function findActiveByRestaurant(RestaurantId $id): BookingCollection;
}
```

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:

```php
// This is an ADAPTER — it's in the Infrastructure layer
final class EloquentBookingRepository implements BookingRepositoryInterface
{
    public function save(Booking $booking): void
    {
        $model = BookingModel::findOrNew($booking->id->toString());
        $model->fill($this->mapper->toDatabase($booking));
        $model->save();
    }

    public function findById(BookingId $id): ?Booking
    {
        $model = BookingModel::find($id->toString());
        return $model ? $this->mapper->toDomain($model) : null;
    }
}
```

Why This Matters

You can swap adapters without changing business logic:

- Today: MySQL via Eloquent
- Tomorrow: PostgreSQL via Doctrine
- Testing: In-memory fake repository

```php
// Production
$repository = new EloquentBookingRepository();

// Testing
$repository = new InMemoryBookingRepository();

// The domain code is identical in both cases
$booking = $repository->findById($bookingId);
$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:

```
final readonly class CreateBookingCommand
{
    public function __construct(
        public BookingId $bookingId,
        public ClientId $clientId,
        public RestaurantId $restaurantId,
```

```
        public TimeSlot $timeSlot,
        public PartySize $partySize
    ) {}
}
```

Queries read state:

```
final readonly class GetBookingByIdQuery
{
    public function __construct(
        public BookingId $bookingId
    ) {}
}
```

Commands Never Return Domain Data

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

```
// Wrong thinking
$booking = $this->commandBus->dispatch(new CreateBookingCommand(...));
return response()->json($booking); // What to return?

// Right thinking
$bookingId = BookingId::generate(); // Generate ID first
$this->commandBus->dispatch(new CreateBookingCommand($bookingId, ...));
return response()->json(['id' => $bookingId->toString()]); // 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

```
final class GetBookingListHandler
{
    public function handle(GetBookingListQuery $query): BookingListDto
    {
        // Direct SQL, optimized for this specific use case
        // No need to go through domain entities
        $results = DB::table('bookings')
            ->join('clients', 'bookings.client_id', '=', 'clients.id')
            ->join('restaurants', 'bookings.restaurant_id', '=', 'restaurants.id')
            ->select([
```

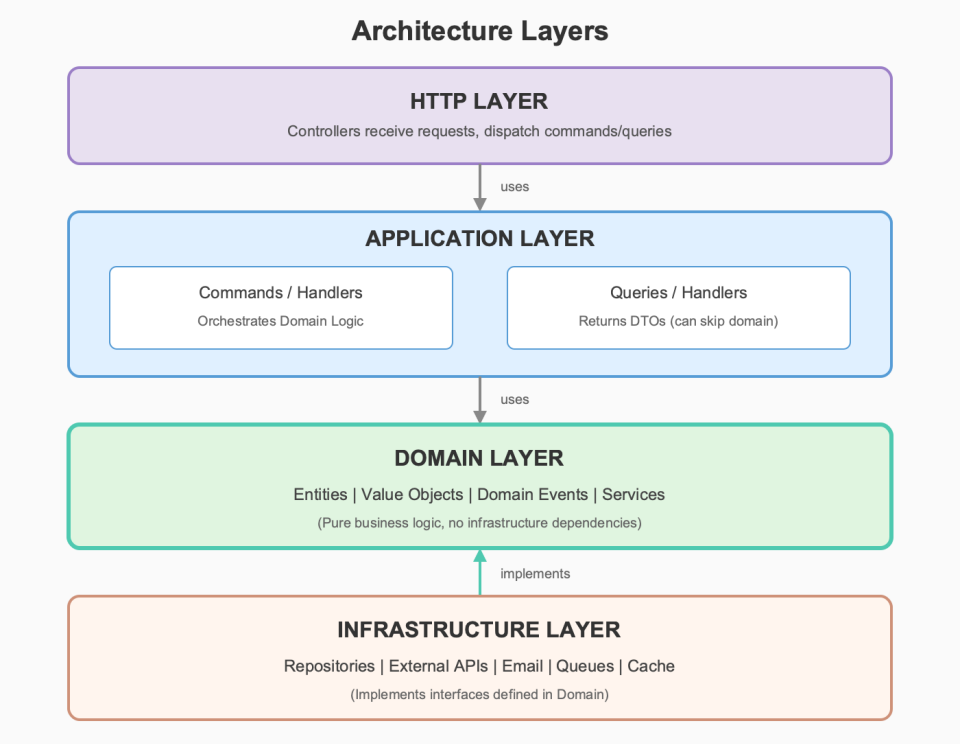
```
                'bookings.id',
                'bookings.date',
                'bookings.status',
                'clients.name as client_name',
                'restaurants.name as restaurant_name',
            ])
            ->where('bookings.restaurant_id', $query->restaurantId->toString())
            ->paginate(20);

        return BookingListDto::fromQueryResults($results);
    }
}
```

This isn't "cheating". This is the point of CQRS—different paths for different needs.

— — — — —

How The Three Pillars Fit Together



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** can go directly to the database for reads
5. **Domain** contains business logic, isolated from everything else
6. **Infrastructure** implements the technical details

— — — — —

A Concrete Example

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

HTTP Layer (Laravel)

```php
final class ConfirmBookingAction extends Controller
{
    public function __invoke(
        ConfirmBookingRequest $request,
        CommandBus $commandBus
    ): JsonResponse {
        $commandBus->dispatch(new ConfirmBookingCommand(
            bookingId: BookingId::fromString($request->route('id'))
        ));

        return response()->json(['status' => 'confirmed']);
    }
}
```

Application Layer (CQRS)

```php
final class ConfirmBookingHandler
{
    public function __construct(
        private BookingRepositoryInterface $repository,
        private EventBus $eventBus
    ) {}

    public function handle(ConfirmBookingCommand $command): void
    {
        $booking = $this->repository->findById($command->bookingId);

        if ($booking === null) {
            throw new BookingNotFound($command->bookingId);
        }

        $booking->confirm(); // Domain logic

        $this->repository->save($booking);
        $this->eventBus->publish($booking->pullEvents());
```

```
    }
}
```

Domain Layer (DDD)

```php
final class Booking extends Entity
{
    public function confirm(): void
    {
        if ($this->status !== BookingStatus::Pending) {
            throw new BookingCannotBeConfirmed($this->id);
        }

        $this->status = BookingStatus::Confirmed;
        $this->confirmedAt = new DateTimeImmutable();

        $this->recordEvent(new BookingConfirmed($this->id));
    }
}
```

Infrastructure Layer (Hexagonal)

```php
final class EloquentBookingRepository implements BookingRepositoryInterface
{
    public function save(Booking $booking): void
    {
        $model = BookingModel::find($booking->id->toString());
        $model->status = $booking->status->value;
        $model->confirmed_at = $booking->confirmedAt;
        $model->save();
    }
}
```

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

— — — — —

$29.99 - PDF + EPUB