From domain to code: a practical approach

phpday 2023

👋 Hello there!

- I’m Davide (he/him)
- PHP Developer for 15+ years
- Technical Leader at Reverse
- I love building stuff

About Domain-Driven Design

> Domain-Driven Design is an approach to software development that centers the development on programming a domain model that has a rich understanding of the processes and rules of a domain (Martin Fowler)

Cheat sheet

- **Ubiquitous language**: a common language used to describe the domain, spoken by developers, stakeholders, and users, and so on.
- **Bounded context**: the scope within which the ubiquitous language is applied
- **Aggregate**: a cluster of associated objects that are treated as a unit (for instance an order and its line items). One of its components is known as the *aggregate root*, which is the "entry point" for the whole aggregate.
- **Invariant**: an assertion about a business rule that must be true at all times.
- **Repository pattern**: an abstraction over storage of aggregates.
- **Event**: something in your domain that has happened in the past.

What problem are we trying to solve?

Well, this is much more complex to get started than usual!

Architecture

Command-Query Responsibility Segregation

- A *command* is an instruction to perform a specific task
- A *query* is a request for information

They both work with the concept of *handlers*, which are those elements (classes) that actually process a command/query and do something with them

Layered architecture

Also *kind of known* as

- Hexagonal architecture
- Clean architecture
- Onion architecture
- Ports and adapters

Layered architecture: Domain layer

- The base layer: it encapsulates the business rules that must be true at all times
- It contains the subjects of your business (aggregates) and their behavior
- It behaves the same across all platforms and systems: it must contain “placeholders” (as interfaces) where it can’t be completely independent and agnostic

```
// Entities
class Book { /** ... */ }

// Value objects
class BookId { /** ... */ }
class Isbn { /** ... */ }

// Interfaces for repositories
interface BookRepositoryInterface { /** ... */ }

// Exceptions use by elements in the domain layer
class BookWithGivenIdCouldNotBeFoundException extends Exception { /** ... */ }
class IsbnIsNotValidException extends Exception { /** ... */ }
```

Layered architecture: Application layer

- This layer encapsulates application-specific business rules, it's what your application can actually do
- It has knowledge of the domain, and works with it to provide use cases for the users
- Just like the domain, it's still quite abstract so it can't interact with external actors

```php
class AddBookCommand {
    public function __construct(
        public readonly BookId $id,
        public readonly Isbn $isbn,
        public readonly $title,
    ) {}
}

class AddBookHandler {
    public function __construct(
        private readonly BookRepository $bookRepository,
    ) {}

    public function __invoke(AddBookCommand $command): void {
        $book = new Book($command->id, $command->isbn, $command->title);

        $this->bookRepository->save($book);
    }
}
```

Layered architecture: Infrastructure layer

This is where you'll find the concrete implementations of the interfaces defined in the domain and application layer, and anything strictly related

```php
class DoctrineBookRepository implements BookRepositoryInterface {
    public function save(Book $book): void {
        try {
            $this->entityManager->persist($book);
            $this->entityManager->flush();
        } catch (Exception $exception) {
            throw BookCouldNotBeSavedException::create($book, $exception);
        }
    }
}
```

Layered architecture: User Interface

- This where you'll have the code that is specific to any platform where you application runs
- It contains controllers, CLI commands, forms, API response definitions, etc

```php
class AddBookController {
    public function __invoke(
        CommandBusInterface $commandBus,
        QueryBusInterface $queryBus,
        Request $request
    ) {
        // Map your request to $bookId, $isbn and $title values

        $command = new AddBookCommand($bookId, $isbn, $title);
        $commandBus->dispatch($command);

        $query = new GetBookQuery($bookId);
        $result = $queryBus->dispatch($query);

        // Do something with $result->book
    }
}
```

Code

How to organize your code

Do not organize by type (Controller, Entity, etc): organize by something that has meaning in your domain

```
$ tree src/
src
└── Library  # This is your main bounded context
    └── Book  # The aggregate root
        ├── Application
        │   └── Command  # The same structure applies to Query
        │       └── AddBook  #  Commands live alongside their handlers
        │           ├── AddBookCommand.php
        │           └── AddBookHandler.php
        ├── Domain
        │   ├── Book.php  # Keep aggregate roots and other entities at the base level
        │   ├── Exception
        │   │   ├── BookWithGivenIdCouldNotBeFound.php
        │   ├── Model
        │   │   └── Isbn.php
        │   └── Repository
        │       └── BookRepository.php
        └── Infrastructure
            └── Doctrine  # Try to group infrastructure elements by a shared trait, like with "Doctrine" here
                ├── Entity
                │   └── Book.orm.xml
                ├── Repository
                │   └── DoctrineBookRepository.php
                └── Type
                    └── IsbnDoctrineType.php
```

IDs

Aggregates must be valid at all times: this means IDs must be handled by your application and not by the database

UUIDs

`df34b7ea-744e-448e-99fa-f47a8e3929a1`

- The easy way of generating in your application an identifier which is *virtually* unique at all times
- They contain some randomness, so their creation must be abstracted from the domain layer: use a library such as `ramsey/uuid` or `symfony/uid`

```php
// Having the ID defined in the constructor allows the entity to be immediately valid
// without having to wait for the database to have an identity value
class Book {
    public function __construct(
        private readonly BookId $bookId,
    ) {}
}
```

Doctrine and DDD

Attributes VS XML

XML metadata allows you to keep Doctrine outside your entities…

…but not completely, because of collections 😢

```
class Order {
    private \Doctrine\Common\Collections\Collection $lineItems;
    public function __construct() {
        $this->lineItems = new \Doctrine\Common\Collections\ArrayCollection();
    }
}
```

Entity behavior can still be unit-tested regardless of how you define metadata!

Aggregates with Doctrine

The overall idea is to make Doctrine work well with clusters of objects (aggregates), treating them *as a single unit* and handling their consistency boundaries correctly

Cascades

```php
class Order {
    private ShippingAddress $shippingAddress;
}
```

```xml
<one-to-one field="shippingAddress" target-entity="App\ShippingAddress" inversed-by="order" orphan-removal="true">
    <!--
    In 1:1 relations, keep the identifier inside the entity closest to the aggregate root,
    this way the owner of the relation is the "parent" and not the "child"
    -->
    <join-column name="shipping_address_id" referenced-column-name="id" nullable="false" />
    <cascade>
        <cascade-all />
    </cascade>
</one-to-one>
```

Change tracking policy

The default option is called `DEFERRED_IMPLICIT`, which means Doctrine will flush all changes to every entity to the database

```
// ...
$book = $entityManager->find(Book::class, $bookId);
$book->rename('...');

// This will save the new name to the database
$entityManager->flush();
```

Using `DEFERRED_EXPLICIT`, Doctrine will flush only changes to those entities that were explicitly set to be persisted or removed, along with their cascades

```
<entity name="App\Book" table="book" change-tracking-policy="DEFERRED_EXPLICIT">
<!-- ... -->
</entity>
```

```
$book = $entityManager->find(Book::class, $bookId);
$book->rename('...');

// This * will not * trigger a save
$entityManager->flush();

// This * will *
$entityManager->persist($book);
$entityManager->flush();
```

Custom types

- Custom Doctrine types allow you to map a column to a value object, and vice versa
- Great for persisting value objects inside your aggregates
- Almost no perceivable performance impact
- Better option than embeddables, which have issues with how null values are treated

Repository pattern

As repositories in the domain layer are just interfaces, they can’t define actual behavior

`Persister` services do this, and they should be the entry points for all persistence logic. Repositories should not be used directly anywhere else.

```
class BookPersister {
    /**
     * @throws BookCouldNotBeSavedException
     * @throws BookWithGivenIsbnCouldNotBeSavedBecauseOneAlreadyExistsException
     */
    public function save(Book $book): void {
        if ($this->bookRepository->isIsbnAlreadyTaken($book)) {
            throw BookWithGivenIsbnCouldNotBeSavedBecauseOneAlreadyExistsException::create($book);
        }

        $this->bookRepository->save($book);
    }
}
```

Events, commands and queries

They are value objects that must be forwarded to the handler is responsible for processing them.

A *bus* is the service that does this.

Bus

- A bus is something that matches an object expressing an intention to a service that does something with it
- Commands and queries must match exactly with one handler
- Events can have zero, one or many handlers.

```php
interface CommandBusInterface {
    /**
     * @throws Exception
     */
    public function dispatch(CommandInterface $command): void;
}

class SymfonyMessengerCommandBus implements CommandBusInterface {
    public function __construct(private MessageBusInterface $symfonyMessengerMessageBus) {
        $this->symfonyMessengerMessageBus = $symfonyMessengerMessageBus;
    }

    public function dispatch(CommandInterface $command): void {
        try {
            $this->symfonyMessengerMessageBus->dispatch($command);
        } catch (HandlerFailedException $exception) {
            throw $exception->getPrevious() ?? $exception;
        }
    }
}
```

```php
class AddBookController {
    public function __invoke(
        CommandBusInterface $commandBus,
        QueryBusInterface $queryBus,
        Request $request
    ) {
        // Map your request to $bookId, $isbn and $title values

        $command = new AddBookCommand($bookId, $isbn, $title);
        $commandBus->dispatch($command);

        $query = new GetBookQuery($bookId);
        $result = $queryBus->dispatch($query);

        // Do something with $result->book
    }
}
```

Recording and dispatching events

As they are part of the domain layer, the natural place to record them is the aggregate root, especially if you use it as entry point for all your actions

An easy way of doing this is to create a specific trait and including it in the aggregate root.

```php
trait EventRecorderTrait {
    /** @var list<EventInterface> */
    private array $recordedEvents = [];

    /** @return list<EventInterface> */
    public function releaseEvents(): array {
        $events = $this->recordedEvents;
        $this->recordedEvents = [];

        return $events;
    }

    protected function recordThat(EventInterface $event): void {
        $this->recordedEvents[] = $event;
    }
}

class Book {
    use EventRecorderTrait;
    public function __construct() {
        // ...
        $this->recordThat(new BookWasAdded($this->id));
    }
}
```

Dispatching events

```php
class BookPersister {
    public function __construct(
        private readonly EventBusInterface $eventBus,
    ) {}

    public function save(Book $book): void {
        // ...

        foreach ($book->releaseEvents() as $event) {
            $this->eventBus->dispatch($event);
        }

        // Not really necessary, but often useful
        $this->eventBus->dispatch(new BookWasSaved($book->id));
    }
}
```

What should go in an event?

Best practice would be to store *everything* about the event that has happened, but almost always storing the ID is good enough

Error handling

My golden rules:

- If a method cannot do what it says it does, it should throw an exception
- Avoid implicit conventions like "`findX` may return null but `getX` can't"
- Always use custom exceptions and extend the base `Exception` class
- Exception names and messages should be as detailed as possible
- Always declare exceptions using `@throws` annotations

```php
interface BookRepositoryInterface {
    /**
     * @throws BookCouldNotBeSavedException
     */
    public function save(Book $book): Book;
}

class BookCouldNotBeSavedException extends Exception {
    public static function create(Book $book, Throwable $previous = null): self {
        return new self(
            sprintf('Book with ID "%s" could not be saved.', $book->id->toString()),
            0,
            $previous,
        );
    }
}

class DoctrineBookRepository implements BookRepository {
    public function save(Book $book): void {
        try {
            $this->entityManager->persist($book);
            $this->entityManager->flush();
        } catch (Exception $exception) {
            throw BookCouldNotBeSavedException::create($book, $exception);
        }
    }
}
```

Bubbling up exceptions

```php
class ShowBookController {
    public function __invoke(QueryBusInterface $queryBus, Request $request) {
        // Map your request to $bookId
        $query = new GetBookQuery($bookId);
        try {
            $result = $queryBus->dispatch($query);
        } catch (BookWithGivenIdCouldNotBeFoundException $exception) {
            // Display a 404 page
        }

        // Do something with $result->book
    }
}
```

Shared components

- Generic value objects, like `Email` or `PhoneNumber`
- Interfaces for commands, queries and events
- General-purpose utilities like services for handling pagination

Golden rules:

- Other bounded contexts *can* access this, but this *cannot* access others
- Treat it as you could make an external library out of it and reuse it in any of your projects, so it can't contain anything specific to the current one

Testing

Behavior-Driven Development

> BDD is designed to test an application's behavior from the end user's standpoint, whereas TDD is focused on testing smaller pieces of functionality in isolation

Experiment with BDD-oriented testing tools, like PHPSpec (unit) or Behat (functional)

Sylius uses both, check out their code!

Static analysis

More precise way of documenting your domain

```php
/** @var positive-int $age */
$age = 1;

/** @var non-empty-string $title */
$title = 'Some title';

/** @var list<Book> $books */
$books === array_values($books); // Lists are homogenous arrays with sequential integer keys starting at 0.
```

More than just null checks

- Define custom types with `@psalm-type`
- Define conditional assertions with `@psalm-assert-if-true`
- Limit usage with `@psalm-internal My\Namespace` to prevent leaking into other layers
- Check for `@throws` annotations so you can be sure exceptions are always annotated correctly

What did we learn today?

- Maintainable software development is complex
- I wanted to give you some basic ideas about how you might get started

What’s left?

Well, a lot! We didn’t talk about a ton of things

- How to deal with assets?
- Where should authentication and authorization live?
- There are plenty of "side discussions" to be had, like "who are your exceptions for?" or "what does null mean in your domain?"

Thank you!

If you have questions now or later, feel free to stop me and say hi 👋

https://tech.reverse.hr