



Why a slim domain model is superior in web

2024-02-23

ConFoo.CA
DEVELOPER CONFERENCE

Hi, I'm Toby! You might know me from ...



2004



2014




FRONTASTIC.CLOUD
Frontastic closes Pre-Series-A over 1.8 million euros -
FRONTASTIC – agile Frontend as a Service

2019

- Web since 1996, PHP since 2000
- Principal Engineer at commercetools Frontend
<https://commercetools.com>

The pitch

- Weblogic does not fit well into a fat domain model
 - Weblogic is not complex enough to justify a fat domain model
- 

Oh those buzzwords

My abstract contains quite some buzz words:


- Slim (/fat) domain model
- Aggregate root & entity (-bound logic)
- Hexagonal architecture

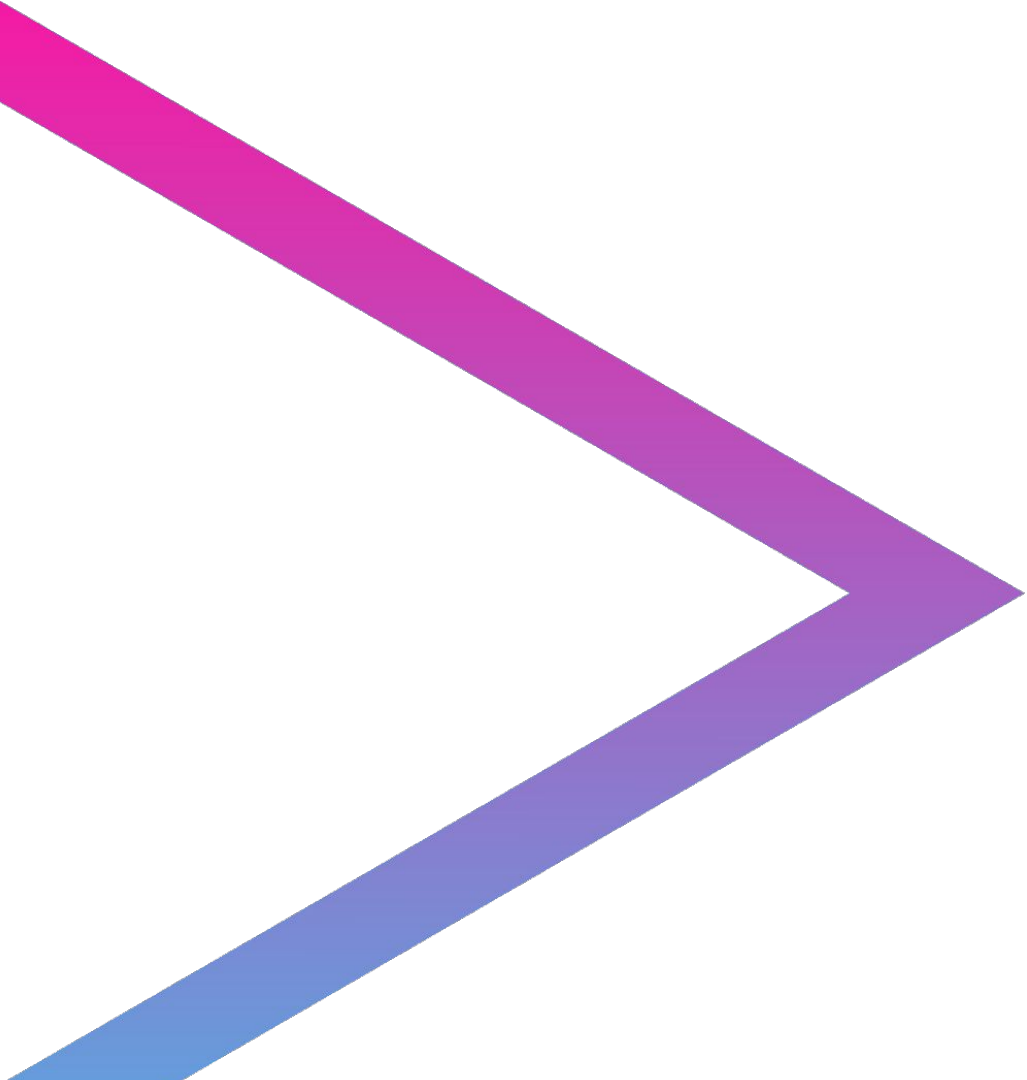
The intention of this slide is to scare you so much, that you won't be scared by the next slides.

Let's start with some definitions first!

$$\mathcal{G} = \Sigma (\oplus), \forall \oplus \exists \boxtimes | \boxtimes \nexists \mathbb{M}$$

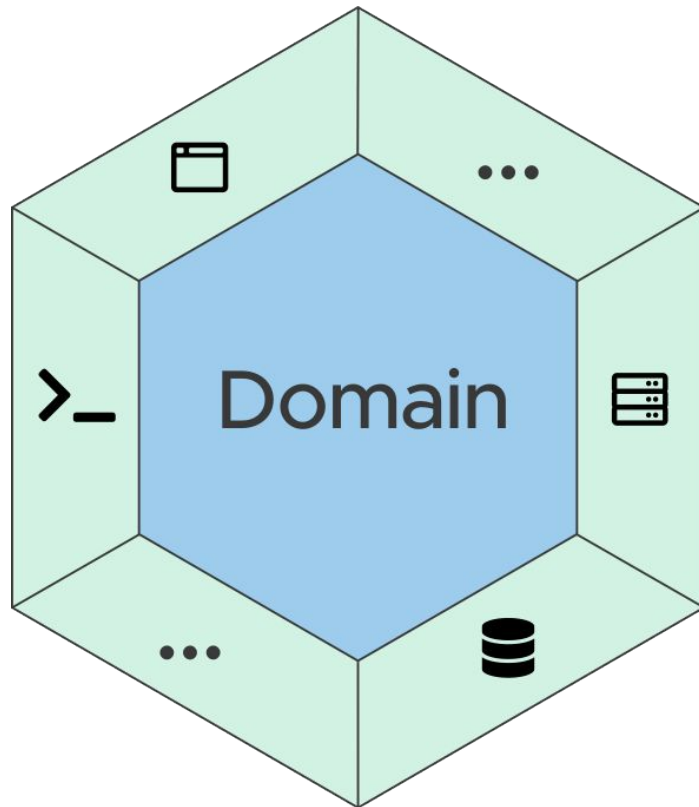
Agenda

1. Hexagonal Architecture
 2. Domain Model
 3. Fat vs. Slim Domain Model
 4. Issues by example
 5. Conclusion
- 
- A decorative graphic at the bottom of the slide consisting of several overlapping, wavy bands of color. From left to right, the colors transition from a bright magenta/pink to a deep purple, and finally to a light blue.




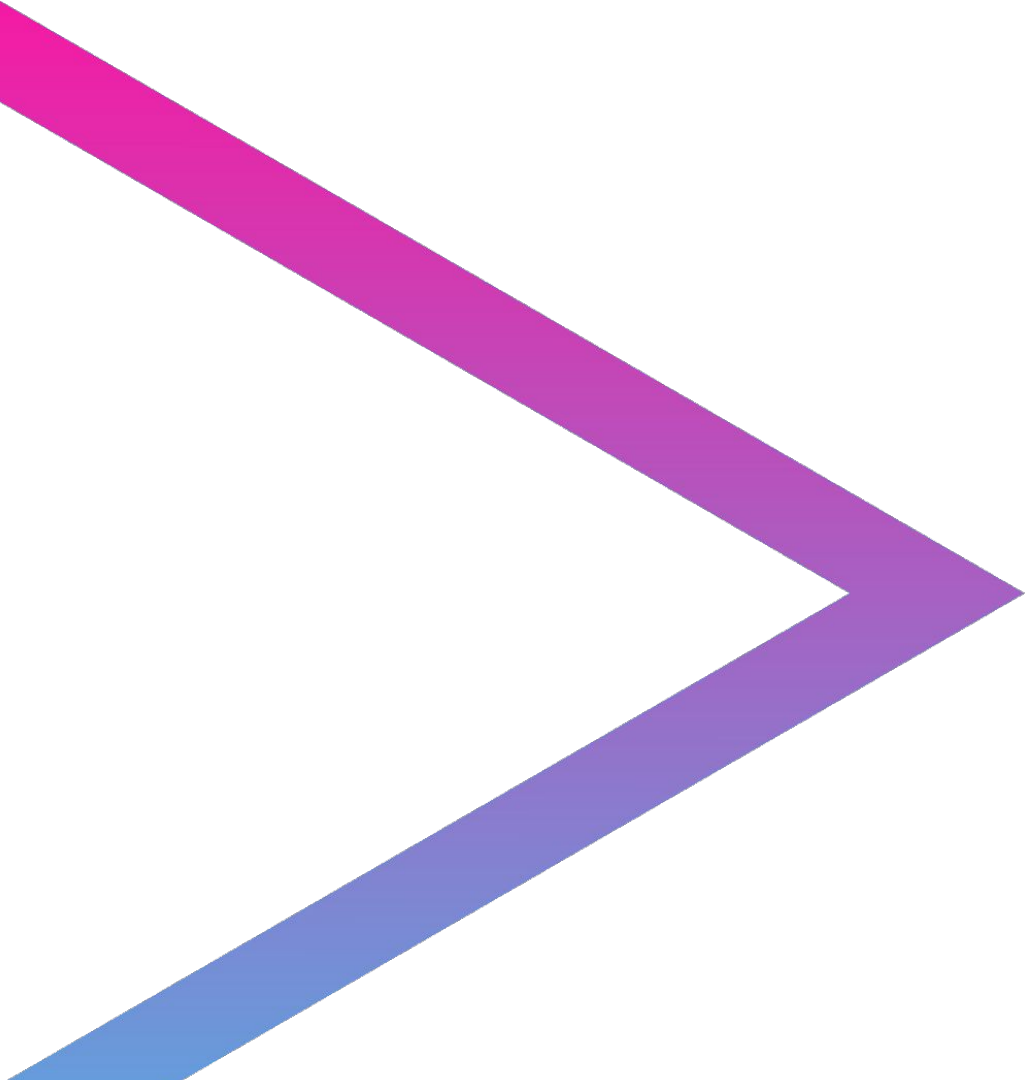
Hexagonal Architecture

Hexagonal Architecture



Hexagonal Architecture


- Decoupling of domain and *infrastructure*
 - Testability by decoupling
 - Make *infrastructure* replaceable
 - “Portability”
- 
- A decorative footer graphic consisting of overlapping, wavy bands of color in shades of pink, purple, and blue, spanning the width of the slide.




Domain Model

Domain Model

- Business logic
 - Written in code

 - Ideally: Following a common “style”
- 

Domain Model: By example

- Bank:
 - Account balancing
 - Interest calculation
 - Medical:
 - Decision support systems
 - Manufacturing:
 - Material requirement planning
 - Supply chain optimization
- 



Web vs. Non-Web

Web application

- Optimized for working on the internet
- Built for horizontal scaling
- Built for proper response times on the web
- Typical: database ↔ code ↔ frontend

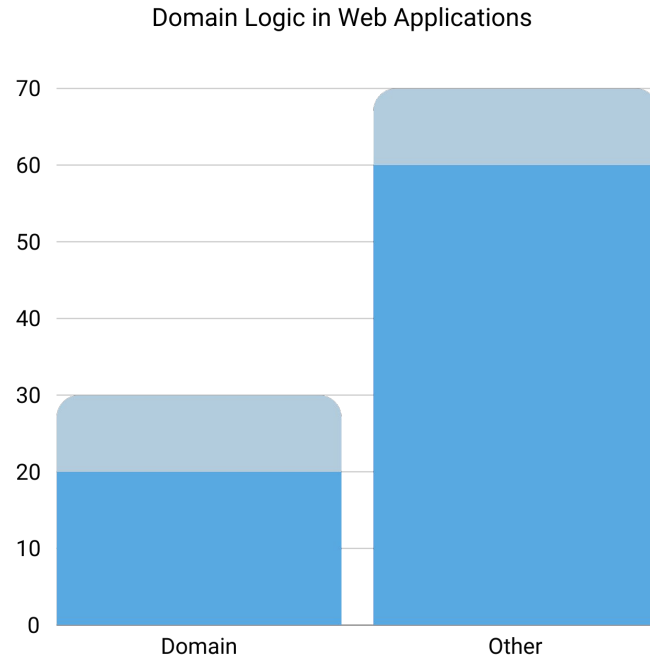
Application with a web-style API

- Internet usage is not primary goal
- Web-style API is just there to ease
- Typically not built for horizontal scaling

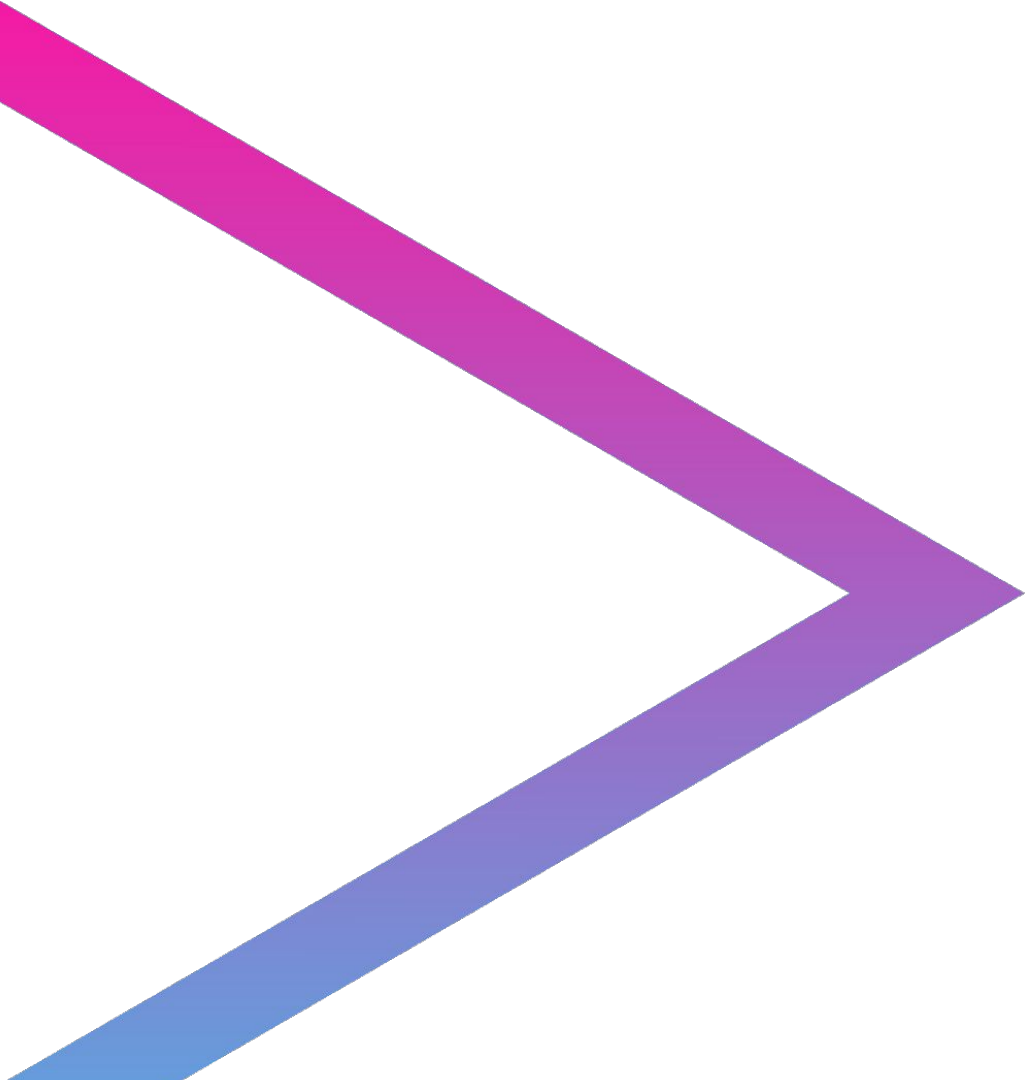
Web Domain Model: By Example

- Content management:
 - Publishing flow
 - Content composition
- eCommerce:
 - Product catalogue (incl. TikTok shops, apps, ...)
 - Checkout flow
 - Story telling
- E-Learning:
 - Course enrollment
 - Content consumption

Web Application Business Logic



* educated, defensive guess



Fat vs. Slim

Fat (Rich) vs. Slim (Anemic)

Non-inclusive terminology.

Let's adapt our language

- Fat (rich) → **Comprehensive**
 - Slim (anemic) → **Streamlined**
- 

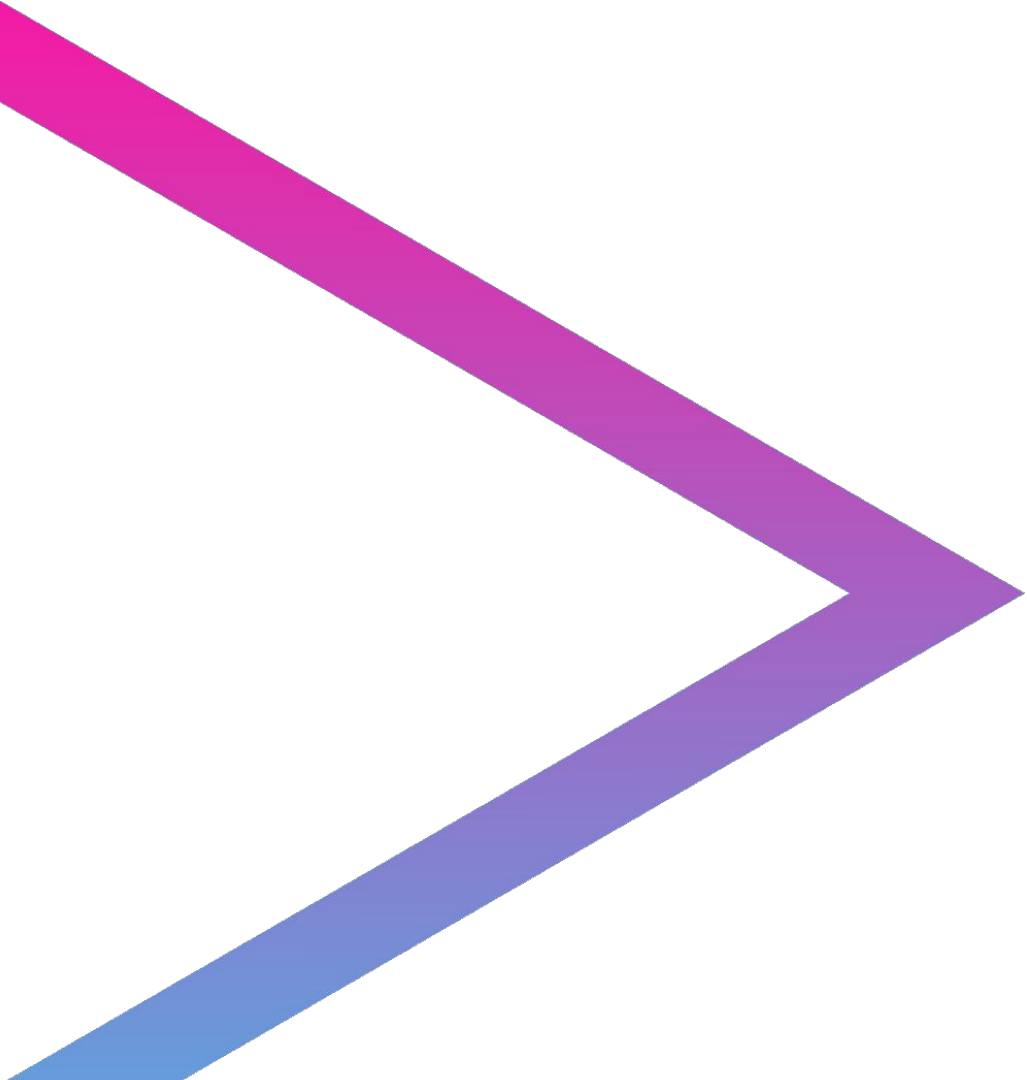
Comprehensive vs. Streamlined

Comprehensive domain model

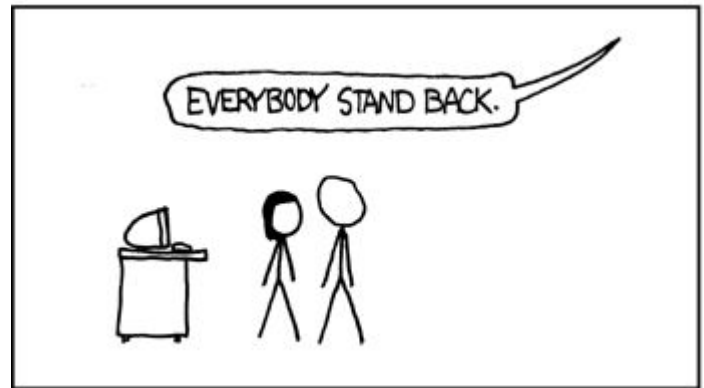
- Close encapsulation (data+logic)
- Deeply nested object trees
- Focus on modelling the real world in code

Streamlined domain model

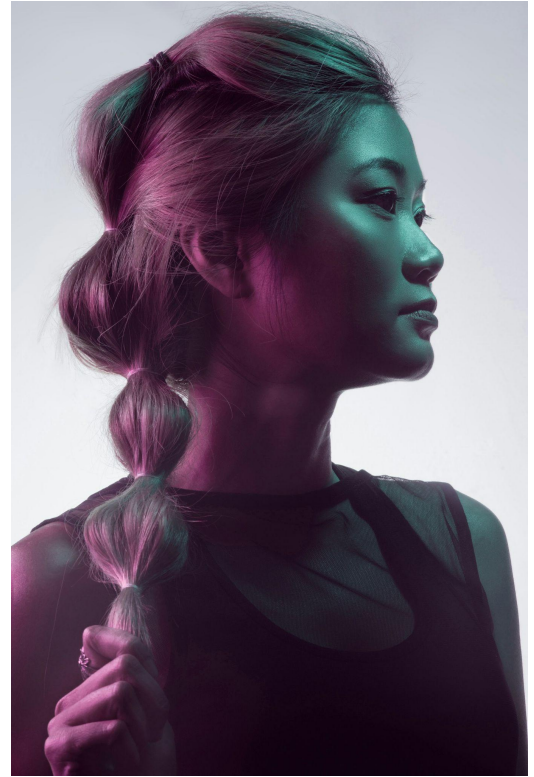
- Data & logic more separated
- Focus on read/write separation
- Maybe more “procedural”? 🤔



By example



User



Comprehensive User Model

```
class User
{
    private string $email;
    public function __construct(string $email)
    {
        $this->setEmail($email);
    }
    public function setEmail(string $email): void
    {
        if (filter_var($email, FILTER_VALIDATE_EMAIL) === false) {
            throw new \InvalidArgumentException('Invalid email');
        }
        $this->email = $email;
    }
}
```

Comprehensive User Service


```
class UserService
{
    public function __construct(private UserRepository $userRepository)
    {
    }
    public function createUser(string $email): User
    {
        $user = new User($email);
        $this->userRepository->save($user);
        return $user;
    }
}
```

Comprehensive User Model

```
class User
{
    public function setEmail(string $email): void
    {
        if (filter_var($email, FILTER_VALIDATE_EMAIL) === false) {
            throw new \InvalidArgumentException('Invalid email');
        }

        [$user, $domain] = explode('@', $email);
        if (checkdnsrr($domain, 'MX') === false) {
            throw new \InvalidArgumentException('Invalid email');
        }
        $this->email = $email;
    }
}
```

The Injection Issue

- Comprehensive works best when all logic is local
 - As soon as an external logic is required:
 - ~~Retrieve service globally (e.g. Singleton)~~
 - ~~Inject service into model~~
 - Extract logic into service
- 

Streamlined User Model

```
class User
{
    public string $email;
    public function __construct(string $email)
    {
        $this->email = $email;
    }
}
```



Streamlined User Service

```
class UserService
{
    public function __construct(private UserRepository $userRepository)
    {}
    public function createUser(string $email): User
    {
        $this->validateEmail($email);

        $user = new User($email);
        $this->userRepository->save($user);
        return $user;
    }
    private function validateEmail($email): void
    {
        /* ... */
    }
}
```

Streamlined User Service - Reusability

```
class UserService
{
    public function __construct(
        private UserRepository $userRepository, private EmailValidator $emailValidator)
    {}
    public function createUser(string $email): User
    {
        $this->emailValidator->validateEmail($email);

        $user = new User($email);
        $this->userRepository->save($user);
        return $user;
    }
}
```

Orders



Comprehensive Orders

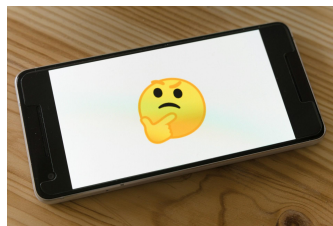
```
namespace Comprehensive;

class User
{
    /** @var Order[] */
    private array $orders = [];
}

class Order
{
    private string $id;
    private \DateTimeImmutable $orderDate;
    private Address $deliveryAddress;
    private Address $billingAddress;
    /** @var OrderItem[] */
    private array $orderItems = [];
    private int $sumCents;
}
```

How do you display a (paged) order overview page?

- Load the entire User object tree
 - Delivers much more information to the frontend than needed
 - Paging requires logic in the code
- Use ORM lazy / partial loading features
 - “Dark magic”
 - Non-functional domain model
 - “Accidental” use of domain function → full model loaded
- Introduce a dedicated model for that purpose



Streamlined Orders

```
namespace Streamlined;  
class OrderOverview  
{  
    public string $orderId;  
    public \DateTimeImmutable $orderDate;  
    public int $sumCents;  
}
```

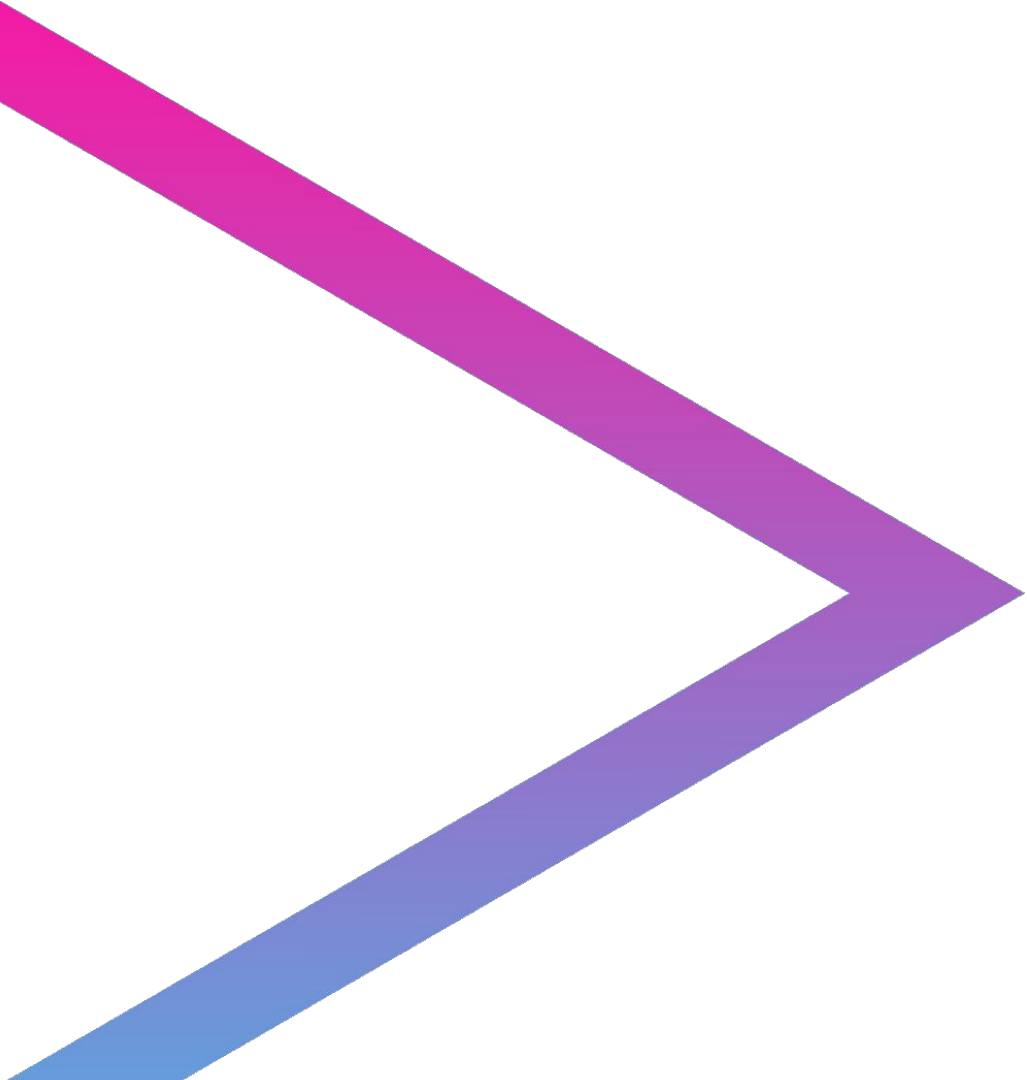

How do you display a (paged) order overview page?

- Service dispatches to the database
 - Probably with hand-written SQL
- Efficiently load the exact view you need

But wait ...


What about all the complex domain logic in checkout and order processing?

- You might want a dedicated service for that
 - But that is not a “web application”
- I know an awesome headless commerce system “by accident”:
<https://commercetools.com>




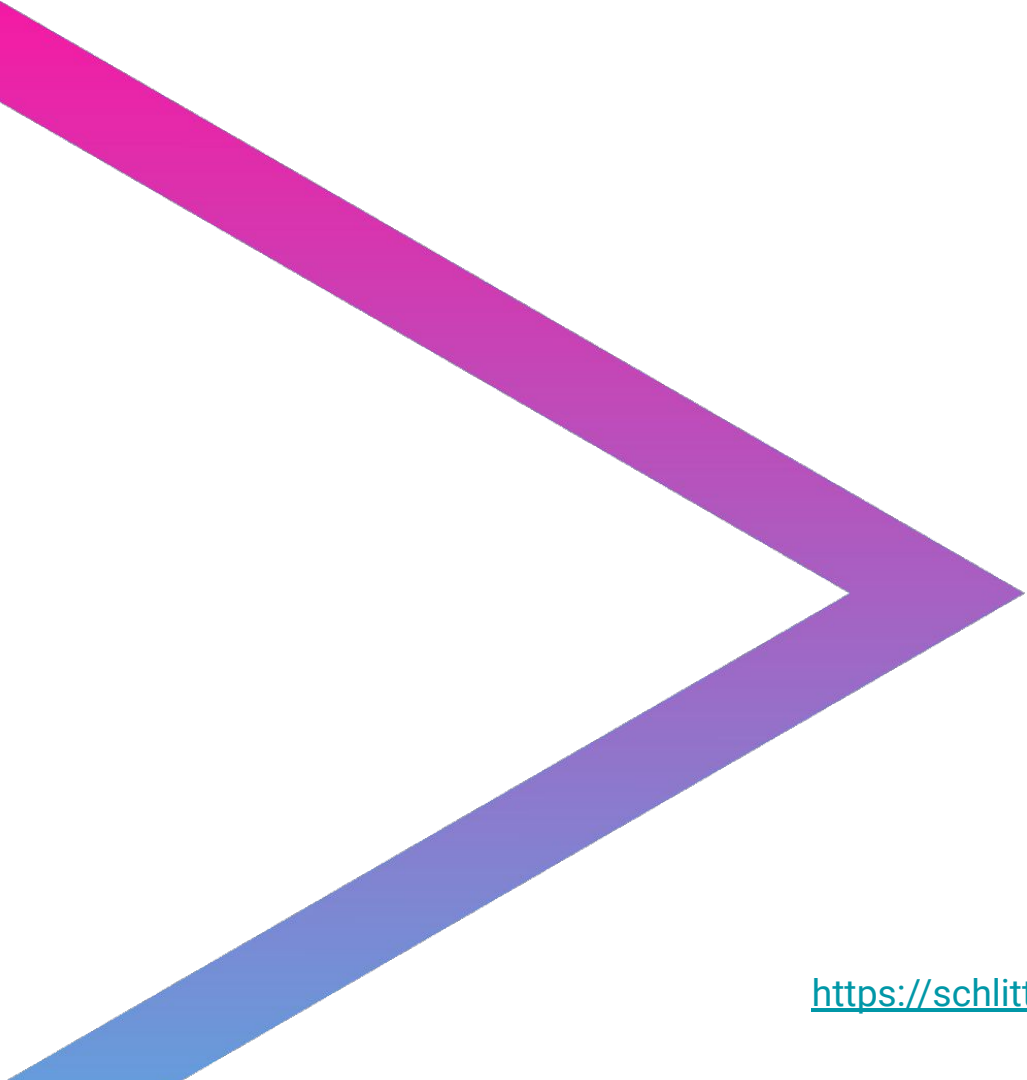
Summary

But ...

- ... Fowler says a streamlined domain model is an anti pattern
 - I disagree
 - There is no general wrong & right
 - ... can't I put any domain logic on my data objects?
 - Anything that is “eternal truth”
 - Anything that does not require external services
 - ... do I need to change my whole coding style now?
 - Of course not!
 - I just want to make you aware that there is more
- 

Conclusion

- Modern web applications access distributed business logic
 - A single web application itself typically does not contain enough logic to justify a comprehensive (was: “fat”) domain model
 - A comprehensive domain model might actually hinder you to build the web application logic you want
 - Attaching “headless” expert systems for sophisticated business logic is a great way to streamline your web application
- 



Q/A

<https://schlitt.info>