

[Open in app ↗](#)

≡ Medium



ⓘ Your payment method has expired. [Update to keep your membership](#)

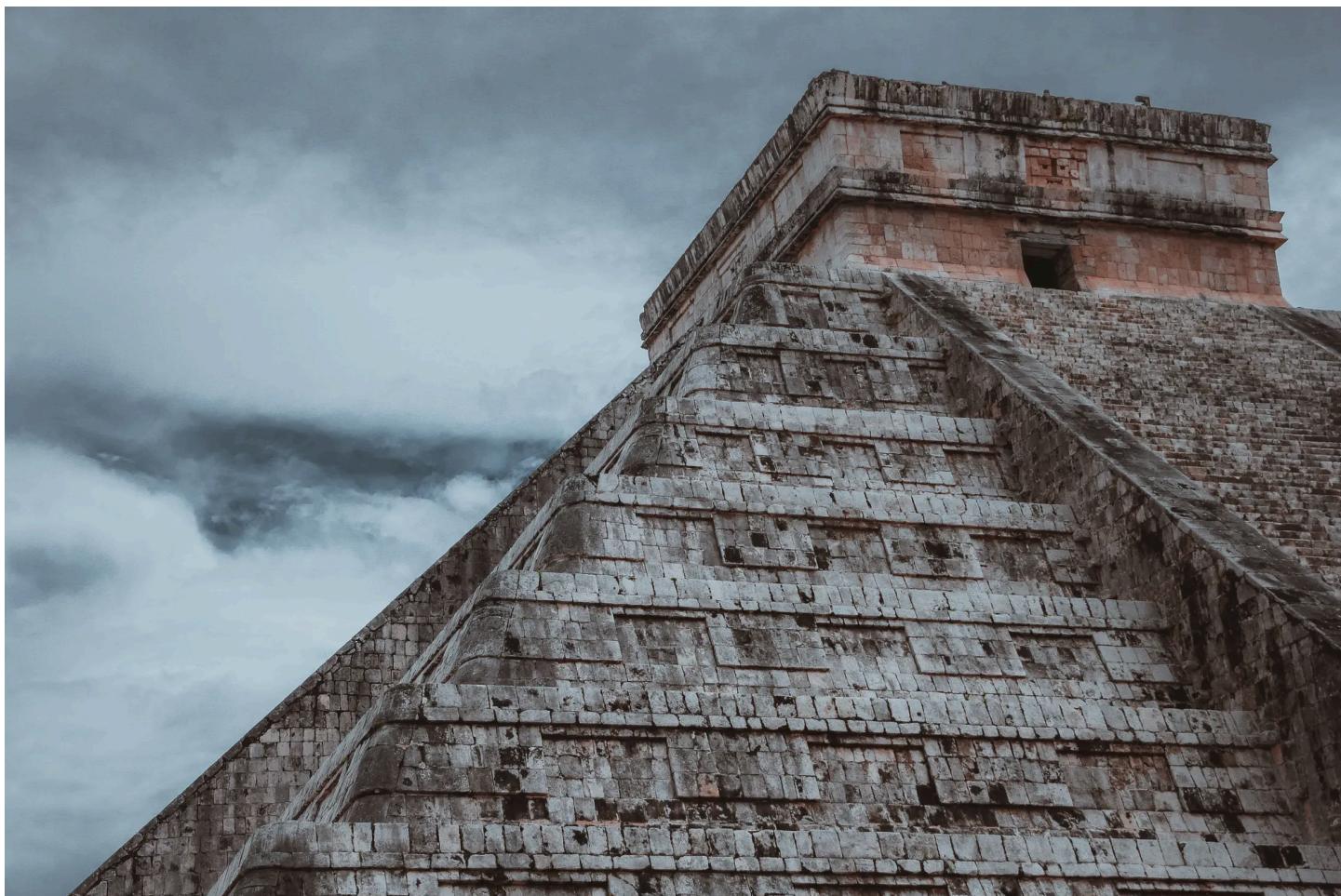
SSENSE-TECH · [Follow publication](#)

Domain-Driven Design: Everything You Always Wanted to Know About it, But Were Afraid to Ask

10 min read · May 15, 2020



Pablo Martinez

[Follow](#)[Listen](#)[Share](#)[More](#)

As one's codebases grow, it is inevitable that their complexity will increase. As this happens, it tends to get more difficult to keep code organized and structured as originally intended, this is known as Software Entropy. Over numerous iterations, maintaining good separation of concerns and properly decoupling classes and modules becomes more challenging if no strict architectural guidelines are enforced.

In the traditional Model-View-Controller (MVC) architecture, the "M" layer would hold all the business logic, but would not provide clear rules on how to maintain proper responsibility boundaries. Several patterns came up to mitigate this problem, but still there was always risk of logic and responsibility leakage between components, making maintainability and stability trickier as the model evolved.

On the other hand, communication with business experts, requirements gathering, and consensus between technical and non-technical teams to properly design and implement a system that solves a business problem is a constant iterative process where things can easily get misinterpreted, and ultimately derail the project from its original goals.

Naming things, for example, has always been one of the most difficult challenges Software Developers face. We should be clear enough for other developers to understand our intentions in the code, while using appropriate naming choices that can facilitate a conversation with business stakeholders.

Domain-Driven Design (DDD) attempts to solve these challenges, by reconciling the technical and non-technical forces that collide in a software project, and proposing a set of practices and patterns that facilitate building a successful system.

So what is Domain-Driven Design?

Let's start by defining what the word domain means in this context. I like to define it as

"A specific sphere of activity or knowledge that defines a set of common requirements, terminology, and functionality on which the application logic works to solve a problem."

Domain-Driven Design is an approach to software design that glues the system's implementation to a constantly evolving model, leaving aside irrelevant details like

programming languages, infrastructure technologies, etc...

It focuses mainly on a business problem and how to strictly organize the logic that solves it. This approach was first described by Eric Evans in his book [Domain-Driven Design Tackling Complexity in the Heart of Software](#).

Now that we know the definition of DDD and what its goals are, let's dive into the three main pillars of this methodology.

. . .

Strategic Design: Splitting your design so as to not lose your mind

As the implementation evolves through many iterations, and the system's complexity grows progressively, it can be daunting to maintain control over it. Therefore, a rigorous strategy to comprehend and control large systems is fundamental. Breaking down the model into Bounded Contexts that interact with each other — which themselves have their own unified model both in concept and in code — is an effective way to avoid complexity pitfalls.

Bounded Context

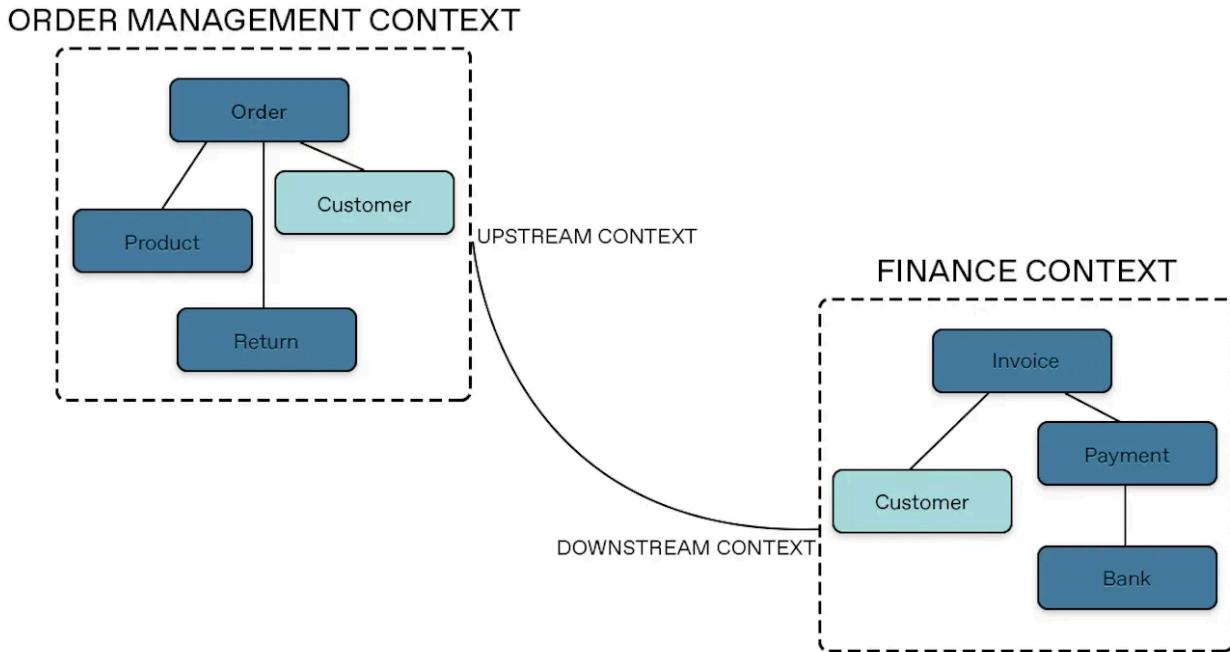
A Bounded Context is a conceptual boundary around parts of the application and/or the project in terms of business domain, teams, and code. It groups related components and concepts and avoids ambiguity as some of these could have similar meanings without a clear context.

For example, outside of a Bounded Context, a “letter” could mean two very different things: either a character or a message written on paper. By defining a boundary and context, you can determine its meaning

In many projects, teams are split by Bounded Contexts, each of them specializing on its own domain expertise and logic.

Context Mapping

Identifying and graphically documenting each Bounded Context in the project is called Context Mapping. Context Maps help better understand how Bounded Contexts and teams relate and communicate with each other. They give a clear idea of the actual boundaries and help teams visually describe the conceptual subdivisions of the system's design.



Relationships between Bounded Contexts can vary, depending on design requirements and other project-specific constraints, some relationships will be omitted from this article with the exception of the following four:

Anti-corruption Layer:

The downstream bounded context implements a layer that translates data or objects coming from the upstream context, ensuring that it supports the internal model.

Conformist:

Downstream bounded contexts conform and adapt to upstream contexts, having to change if needed. In this case the upstream context has no interest in meeting requirements downstream.

Customer/Supplier:

Upstream supplies downstream with a service, and downstream context acts as a customer, determining requirements and requesting changes upstream to meet their needs.

Shared Kernel:

Sometimes it's inevitable that two (or more) contexts overlap, and end up sharing resources or components. This relationship requires both contexts to be in continuous synchronization when changes are needed, therefore it should be avoided if possible.

• • •

Collaborative Modeling: Rich communication and effective collaboration

DDD proposes to model the domain effectively by doing a collaborative approach, involving all parties that hold not only technical, but business knowledge as well. As Evans describes it, the Domain Model “is not only the knowledge in a domain expert’s head; it is a rigorously organized and selective abstraction of that knowledge.”

Developers collaborate with domain experts with the intention of constantly refining the Domain Model, forcing them to learn important details and principles of the business problem they are trying to solve, instead of just producing code mechanically.

To enable this collaboration between business and technical teams, the Domain Model should use a language that joins business with technical jargon, and finds a middle ground that all team members can understand and agree on, this is called a Ubiquitous Language. Using a well-defined Ubiquitous Language will improve every interaction between tech and business teams, making them less ambiguous and more effective.

Ultimately this Ubiquitous Language will be embedded in the code.

• • •

Tactical Design: The Nuts and Bolts of DDD

Implementing associations between domain objects and describing their functionality feels easy at a glance, but proper distinction of their meaning and reason of existence should be done in a clear and intuitive way. DDD proposes a set of constructs and patterns to achieve it.

Entities

Objects that have a unique identity and possess a thread of continuity are called Entities, they are not defined solely by their attributes, but more by *who they are*. Their attributes might mutate and their life cycles can drastically change, but their identity persists. Identity is maintained via a unique key or a combination of attributes guaranteed to be unique.

In the e-commerce domain for example, an `order` has a unique identifier and it goes through several different stages: open, confirmed, shipped, and others, therefore it's considered a Domain Entity.

Another example could be a Customer:

```

1  export class Customer {
2
3      private id: number;
4      private name: string;
5
6      protected constructor(name: string) {
7          // A uuid guarantees a unique identity for the Customer Entity
8          this.id = uuidv4();
9          this.name = this.setName(name);
10     }
11
12     private setName(name: string): string {
13         // Business invariant: Customer name should not be empty
14         if (name === undefined || name === '') {
15             throw new Error('Name cannot be empty');
16         }
17         return name;
18     }
19
20     public static create(name: string): Customer {
21         return new Customer(name);
22     }
23 }
```

[Customer.ts](#) hosted with ❤ by GitHub

[view raw](#)

Value Objects

Objects that describe characteristics, and that do not possess any unique identity are called Value Objects, they care only about what they are, not who they are.

Value Objects are attributes of, and can be shared by multiple entities, for example: two customers can have the same shipping address. Although there's a risk — if one

of their attributes needs to change, all the entities that share them would get impacted. To prevent this, Value Objects must be immutable, forcing the system to replace them by fresh new instances, when an update is required.

Also, Value Object creation should always depend on the validity of the data used to create them and how it respects business invariants. Therefore, if data is invalid, the object instance will not be created. For example, in North America, a postal code with non-alphanumeric characters would violate a business invariant and would trigger an exception on the Address creation.

```
1  export class Address {  
2  
3      private readonly streetAddress: string;  
4      private readonly postalCode: string  
5  
6      protected constructor(streetAddress: string, postalCode: string) {  
7          this.streetAddress = this.getValidStreetAddress(streetAddress);  
8          this.postalCode = this.getValidPostalCode(postalCode);  
9      }  
10  
11     private getValidStreetAddress(streetAddress: string): string {  
12         // Business invariant: street address should not be longer than 128 characters  
13         if (streetAddress.length > 128) {  
14             throw new Error('Address should not be longer than 128 characters');  
15         }  
16         return streetAddress;  
17     }  
18  
19     private getValidPostalCode(postalCode: string): string {  
20         // Business invariant: Should be a valid canadian postal code  
21         const pattern = /[a-z]\d[a-z][ \-]?[a-z]\d/g;  
22         if (!postalCode.match(pattern)) {  
23             throw new Error('Postal code should only contain alphanumeric characters and  
24         }  
25         return postalCode;  
26     }  
27  
28     public getStreetAddress(): string {  
29         return this.streetAddress;  
30     }  
31  
32     public getPostalCode(): string {  
33         return this.postalCode;  
34     }  
35  
36     public static create(streetAddress: string, postalCode: string): Address {  
37         return new Address(streetAddress, postalCode);  
38     }  
39  
40     public equals(otherAddress: Address): boolean {  
41         // Value Objects equality is based on their propertie's values  
42         return objectHelper.isEqual(this, otherAddress);  
43     }  
44 }
```

Address.ts hosted with ❤ by GitHub

[view raw](#)

Services

In many cases the Domain Model requires certain actions or operations that are not directly related to an Entity or a Value Object, forcing these into their implementations causes a distortion of their definition. Services are classes that offer stateless operations. They are commonly named as verbs, as opposed to nouns for Entities and Value Objects, and are named based on the Ubiquitous Language.

Services should be carefully crafted, always ensuring that they do not strip Entities and Value objects of their direct responsibilities and behavior. They should also be stateless, such that clients can use any given instance of a Service disregarding that instance's history during the application's lifetime. Having Entities and Value Objects with no domain logic is considered an anti-pattern called the Anemic Domain Model.

Domain Objects and their Life Cycle

Domain objects commonly have complex life cycles, they are instantiated, go through several changes, interact with other objects, execute operations, get persisted, get reconstituted, get deleted, and so on. Maintaining their integrity while ensuring that the system doesn't mismanage their complex lifecycle is one of the main challenges that implementing a proper Domain Model represents.

Minimizing the relationship and interaction between domain objects to maintain manageable levels of complexity within the Domain Model is difficult, especially in complex business domains, or as Eric Evans describes it:

"It is difficult to guarantee the consistency of changes to objects in a model with complex associations. Invariants need to be maintained that apply to closely related groups of objects, not just discrete objects. Yet cautious locking schemes cause multiple users to interfere pointlessly with each other and make a system unusable"

Aggregates

To mitigate the aforementioned challenge, an aggregation of Entities and Value Objects is needed, restricting the violation of business invariants.

Aggregates are collections of related Entities and Value Objects, clustered together representing a transactional boundary. Each Aggregate has an Entity that faces outwards and controls all access to the objects inside the boundary, this entity is called the Aggregate Root, and it's the only object that other objects can interact with. No objects within the Aggregate can be called directly from the outside world, thus maintaining consistency within.

Business Invariants are business rules that guarantee the integrity of an Aggregate and its contents, in other words, it is a mechanism that ensures that their state is always consistent with the business rules. For example, an order can never be placed when the stock quantity of a certain product is zero.

```
1  export class Order {
2
3      private id: number;
4      private isConfirmed: boolean;
5      private total: number;
6      private shippingAddress: Address;
7      private customer: Customer;
8      private items: Product[];
9      private payments: Payment[];
10
11     constructor(
12         customer: Customer,
13         shippingAddress: Address,
14         items: Item[],
15         payments: Payment[]
16     ) {
17         // Generate a unique identifier (UUID) for the Order Entity
18         this.id = uuidv4();
19         this.isConfirmed = false;
20         this.total = 0;
21         this.customer = customer;
22         this.shippingAddress = shippingAddress;
23         this.items = items.length ? items : [];
24         this.payments = payments.length ? payments : [];
25     }
26
27     private getPaymentsTotal(): number {
28         return this.payments.reduce((accumulator, payment) => accumulator + payment.total, 0);
29     }
30
31     public addPayments(payment: Payment): void {
32         this.payments.push(payment);
33         this.total += payment.total;
34     }
35
36     public addItems(product: Product): void {
37         // Business invariant: an order should not have items which are not in stock
38         if (!product.getStockQuantity()) {
39             throw new Error(`No stock for product id: ${product.id}`);
40         }
41         this.items.push(product);
42     }
43
44     public confirm(): void {
45         // Business invariant: only fully paid orders can be confirmed
46         if (this.total === this.getPaymentsTotal()) {
47             throw new Error('Total amount paid does not equal order total');
48         }
49     }
50 }
```

```

49     this.isConfirmed = true;
50 }
51 }
```

Order.ts hosted with ❤ by GitHub

[view raw](#)

Factories

Creating complex object and aggregate instances can be a daunting task, and can also disclose too much of the object's internal details. Using Factories, we can solve this problem and provide the necessary encapsulation.

A Factory should be able to construct domain objects or aggregates in one atomic operation, requiring all data needed to be provided by the client when called, and enforcing all invariants on the created object. This activity is not part of the Domain Model, but still belongs in the Domain Layer as it is part of the business rules that apply to the system.

```

1  export class OrderFactory implements Factory {
2
3      private customerEntity: Customer;
4      private addressValue: Address;
5      private productsRepository: Repository;
6      private paymentsRepository: Repository;
7
8      constructor(customerEntity: Customer, addressValue: Address, productsRepository: Re
9          this.customerEntity = customerEntity;
10         this.addressValue = addressValue;
11         this.productsRepository = productsRepository;
12         this.paymentsRepository = paymentsRepository;
13     }
14
15     public async createOrder(customerName: string, addressDto: AddressDto, itemDtos: It
16         try {
17             const customer = this.customerEntity.create(customerName);
18             const shippingAddress = this.addressValue.create(addressDto.streetAddress,
19                 const items = await this.productsRepository.getProductCollection(itemDtos);
20                 const payments = await this.paymentsRepository.getPaymentCollection(payment
21
22                     return new Order(customer, shippingAddress, items, payments);
23                 } catch(err) {
24                     // Error handling logic should go here
25                     throw new Error(`Order creation failed: ${err.message}`);
26                 }
27             }
28 }
```

OrderFactory.ts hosted with ❤ by GitHub

[view raw](#)

Repositories

To be able to retrieve objects from our persistence, be it in-memory, filesystem, or database, we need to provide an interface that hides the implementation details from the client, so that it does not depend on the infrastructure specifics, but merely on an abstraction.

Repositories provide an interface that the Domain Layer can use to retrieve stored objects, avoiding tight-coupling with the storage logic and giving the client an illusion that the objects are being retrieved directly from memory.

It's important to mention that all repository interface definitions should reside in the Domain Layer, but their concrete implementations belong in the Infrastructure Layer.

Isolating the Domain from Other Concerns

The part of the code that is written to specifically solve domain problems is a fraction of the whole codebase. If this portion is intertwined with code that solves other concerns, it will be difficult to understand and improve. Clearly separating domain logic from all other functionality will reduce the leakage and will avoid confusion in a large and complex system.

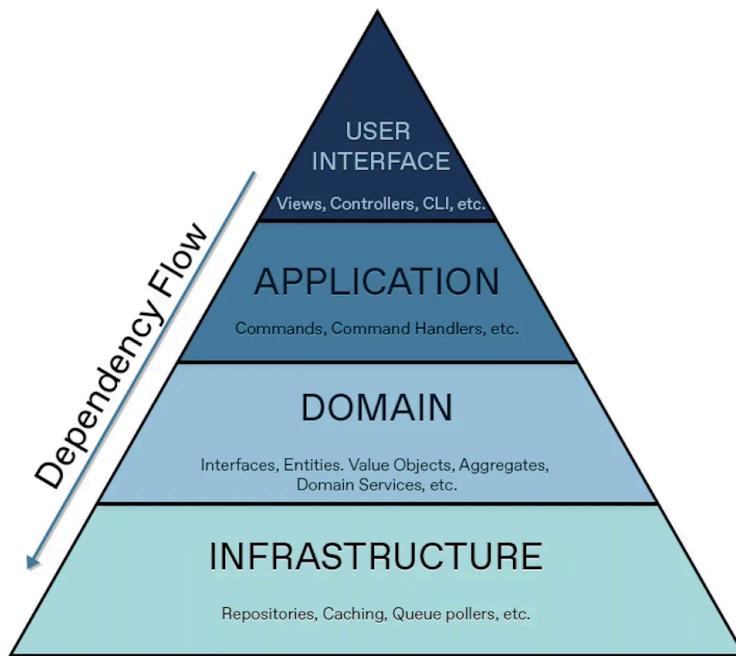
DDD proposes a Layered Architecture as a way to separate concerns and avoid responsibility confusion by splitting the codebase in 4 main layers: User Interface, Application, Domain, and Infrastructure.

The main rule here is that components in each layer should depend only on components in the same layer or any layer beneath it. Upper layers can use

components of lower ones just by calling their public interfaces and lower layers can *only* communicate upwards by Inversion of Control (IoC).

- **User Interface Layer:** is responsible for displaying data and capturing user's commands.
- **Application Layer:** serves as an orchestrator of domain work, it does not know domain rules, but organizes and delegates domain objects to do their job. It is also the only layer accessible to other bounded contexts.
- **Domain Layer:** holds the business logic and rules, as well as the business state. It is where the Domain Model lives.
- **Infrastructure Layer:** implements all the technical functionalities the application needs to support the higher layers, persistence, messaging, communication between layers, etc...

Even though not all layers are required in every system, the existence of the Domain Layer is a prerequisite in DDD.



Conclusion

All in all, DDD is a holistic approach for solving business problems through rich collaboration with domain experts and strict design patterns, it is not a universal

solution for all software projects, but it can provide significant benefits when properly applied.

There are several books on this subject, and various concepts were purposely left out of this article, but If I have sparked your interest in Domain-Driven Design, I invite you to start by reading the [blue](#) and [red](#) books, in that order, which are the starting point of this massive topic.

• • •

Editorial reviews by [Anas Trabulsi](#), [Mario Bittencourt](#), [Mikhail Levkovsky](#), [Deanna Chow](#), [Liela Touré](#), & [Prateek Sanyal](#).

Want to work with us? Click [here](#) to see all open positions at SSENSE!

Software Architecture

Design Patterns

Software

Software Engineering

Domain Driven Design

Some rights reserved 

SSENSE

Follow

Published in SSENSE-TECH

1.6K followers · Last published Aug 30, 2025

Ideas and research from the software, data & product teams behind the global fashion platform SSENSE.



Follow

Written by Pablo Martinez

755 followers · 47 following

Responses (11)



Mr. Akash

What are your thoughts?



Thomas De Craemer

May 25, 2020

...

Interesting article! I have one question regarding your last image, why would the Domain Layer depend on the Infrastructure Layer? In a typical onion architecture, the Domain Layer forms the core of an application and has no outer dependencies.



9



3 replies

[Reply](#)

Johan Tovar

Feb 17, 2021

...

Excelente trabajo, un tema bastante complejo e interesante y muy bien abordado en este artículo!



10



1 reply

[Reply](#)

fong Liu

Jun 1, 2020

...

Great article!

However, I have a question about your repository design. From my experience, a repository per an aggregate; therefore, I only upsert an aggregate in one repository save.

I'm curious that if you treat your orderItems and orderPayment as... [more](#)



3



1 reply

[Reply](#)

[See all responses](#)

More from Pablo Martinez and SSENSE-TECH



In SSENSE-TECH by Pablo Martinez

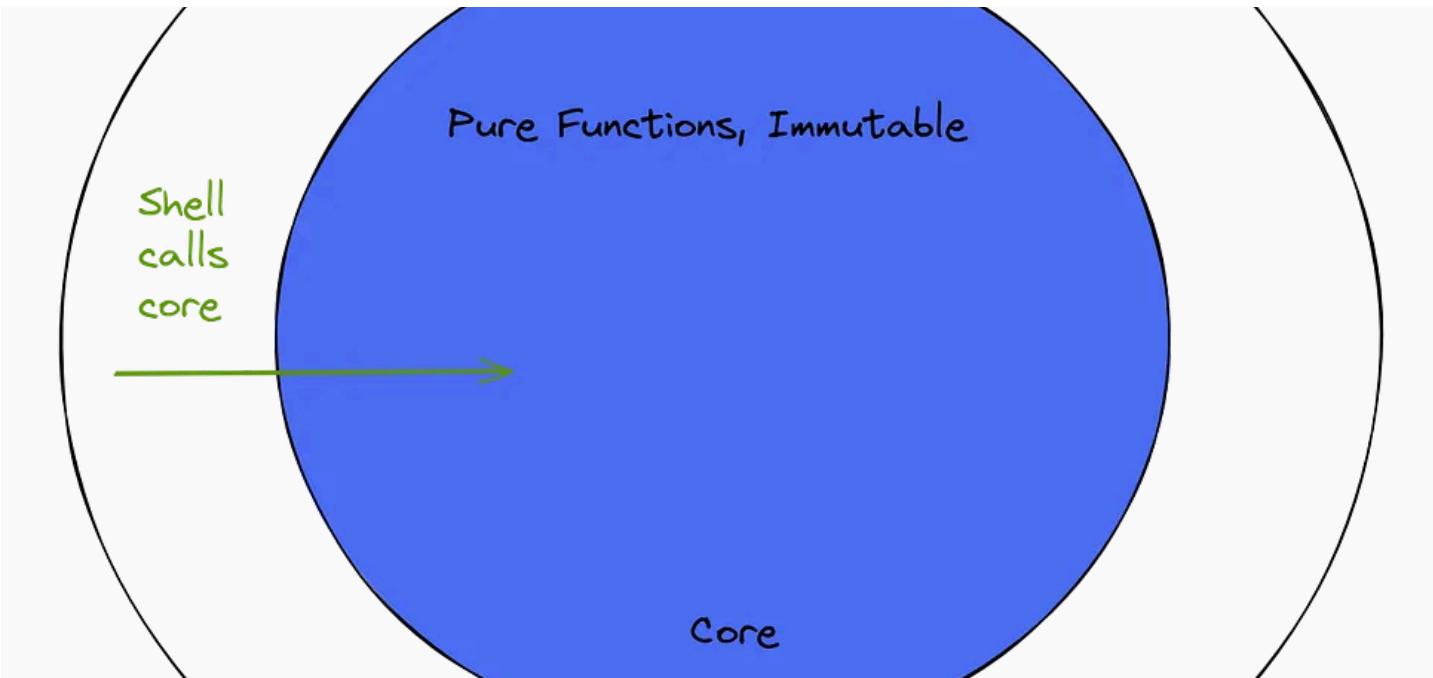
Hexagonal Architecture, there are always two sides to every story

The Hexagonal Architecture, also referred to as Ports and Adapters Architecture

Jul 9, 2021 2.7K 28



...



 In SSENSE-TECH by Mario Bittencourt

A Look at the Functional Core and Imperative Shell Pattern

If you come from an enterprise background, you have most likely been working with Java or C# and automatically associate the analysis and...

May 19, 2023  224  1



SSENSE+
Account Details
Email Preferences
Addresses
Appointments
Logout

PAYMENT METHOD
VISA **** -----

SomeCity, SomeState
Canada
111-123-1234

SomeCity, SomeState
Canada
111-123-1234

SHIPPED

THE NORTH FACE
Black Nuptse Down Coat
Size: S
SKU: 232802F06116801

\$300.00
\$600.00

TRACK ORDER

TOTAL ORDER SUMMARY	
Subtotal	\$300.00
Shipping total	\$0.00
GST (5 %)	\$15.00
PST (9.975 %)	\$29.93
ORDER TOTAL	\$344.93 CAD

 In SSENSE-TECH by Sam-Nicolai Johnston

Event Sourcing: A Practical Guide to Actually Getting It Done

Event Sourcing—the pattern for storing data as events in a log and using them to reconstruct its state—has been used for many years...

Mar 22, 2024

281

2



In SSENSE-TECH by Pablo Martinez

Dependency Injection Vs Dependency Inversion Vs Inversion of Control, Let's set the Record Straight

Dependency Injection, Dependency Inversion, and Inversion of Control are 3 terms that, although related, are commonly confused and...

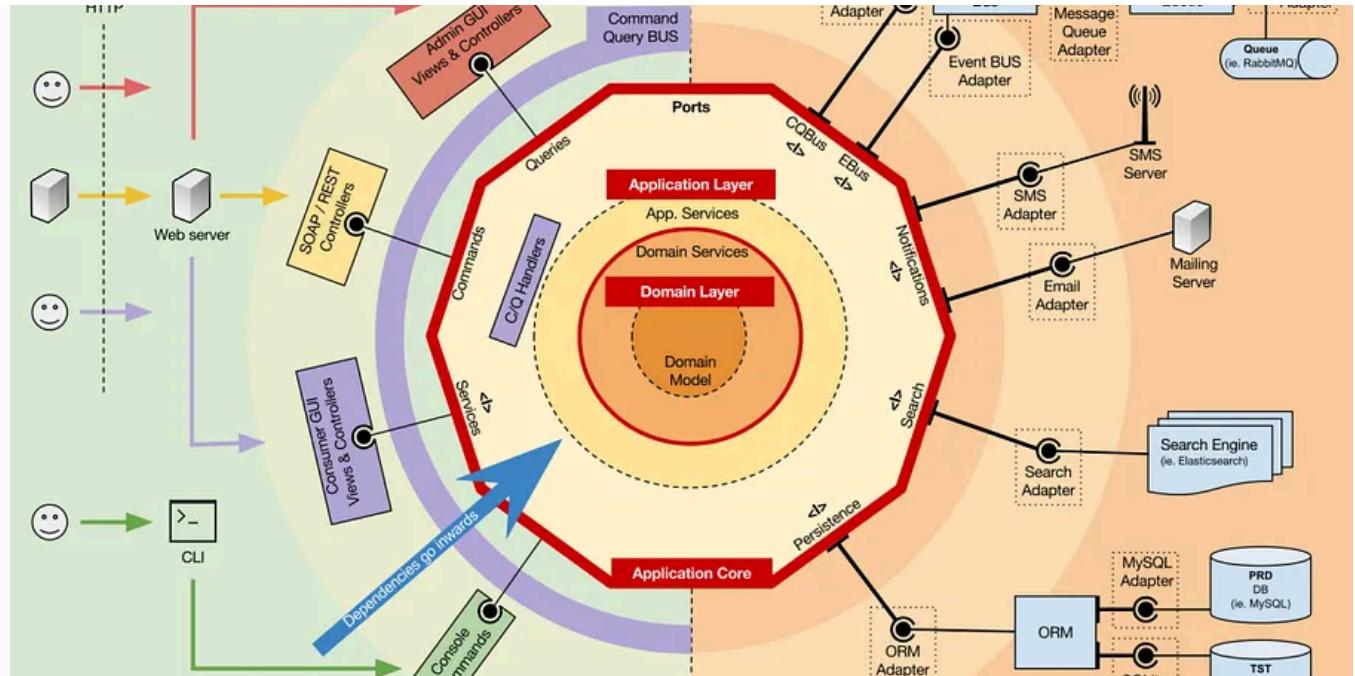
Feb 4, 2022

1.1K

7

[See all from Pablo Martinez](#)[See all from SSENSE-TECH](#)

Recommended from Medium

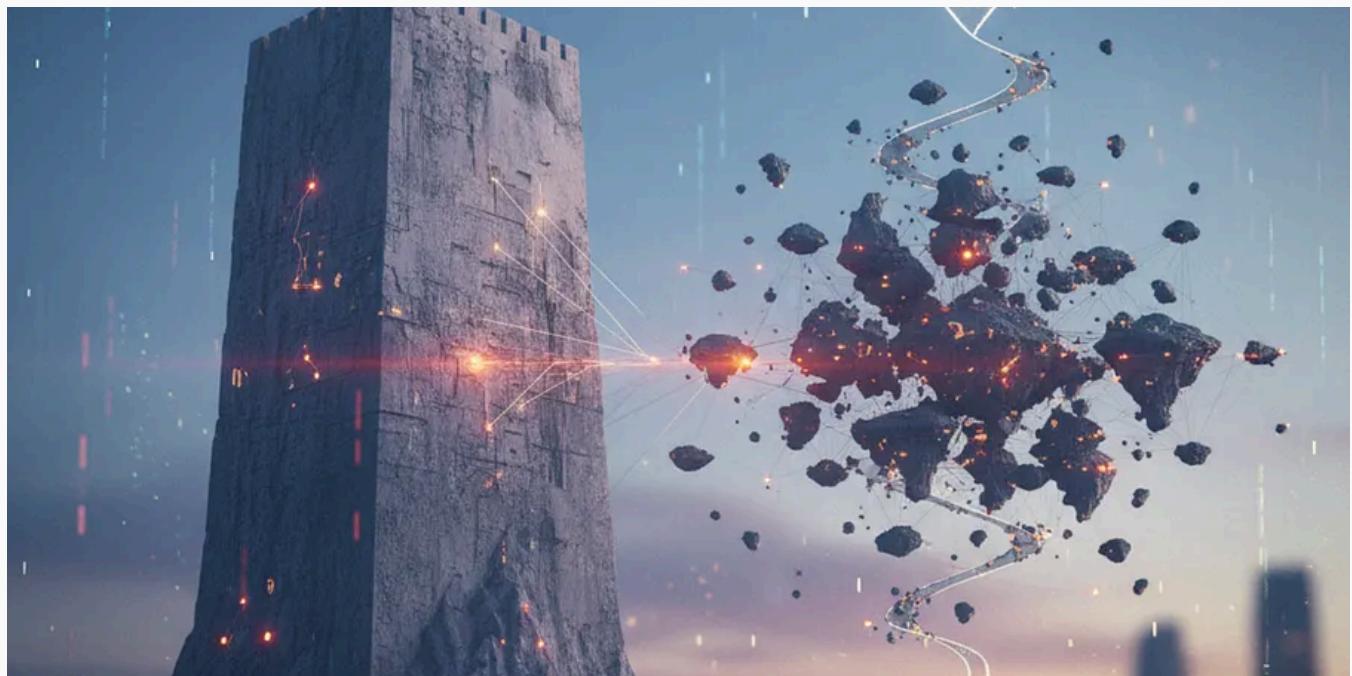


Hieu Nguyen

Understanding Hexagonal Architecture Through a Practical Application

Introduction

Jun 22 1



Yash Batra

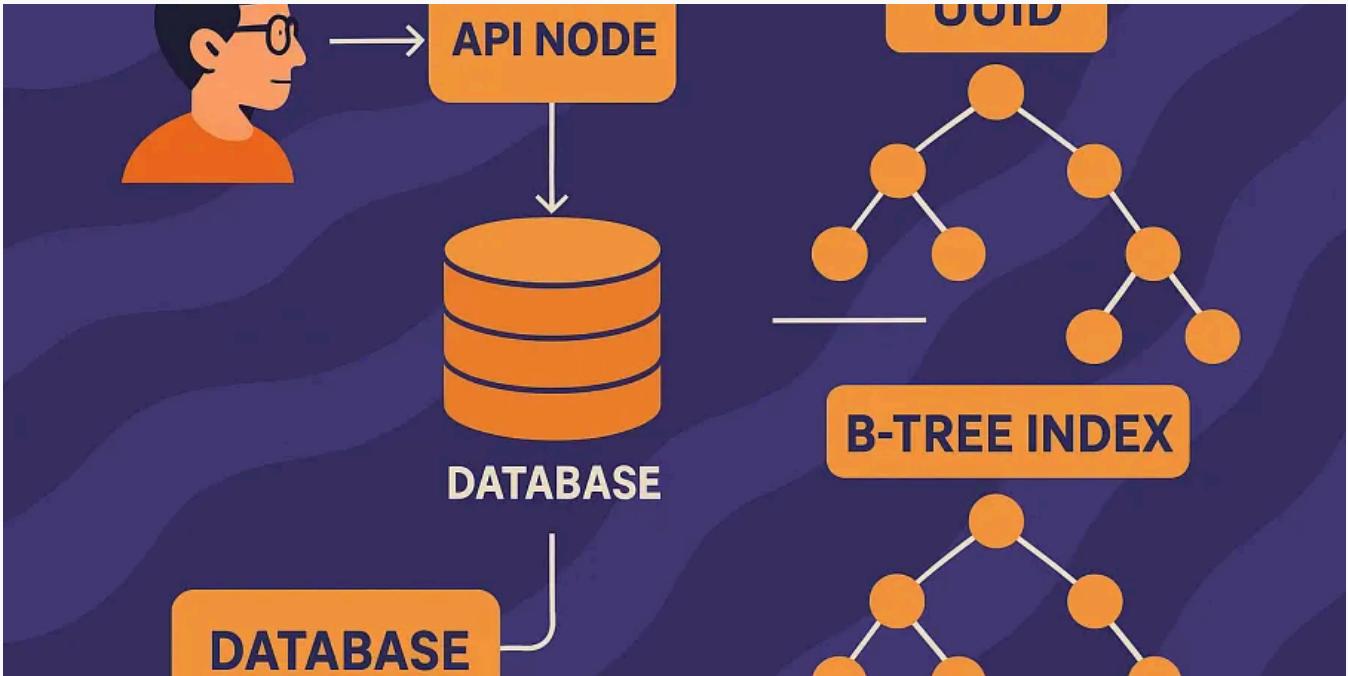
How Netflix Accidentally Proved Monoliths Scale Better Than Microservices

For a decade, the dominant Silicon Valley mantra has been: monoliths don't scale; microservices do. Netflix itself helped popularize this...

★ Sep 22 1.6K 62



...



The Concurrent Mind

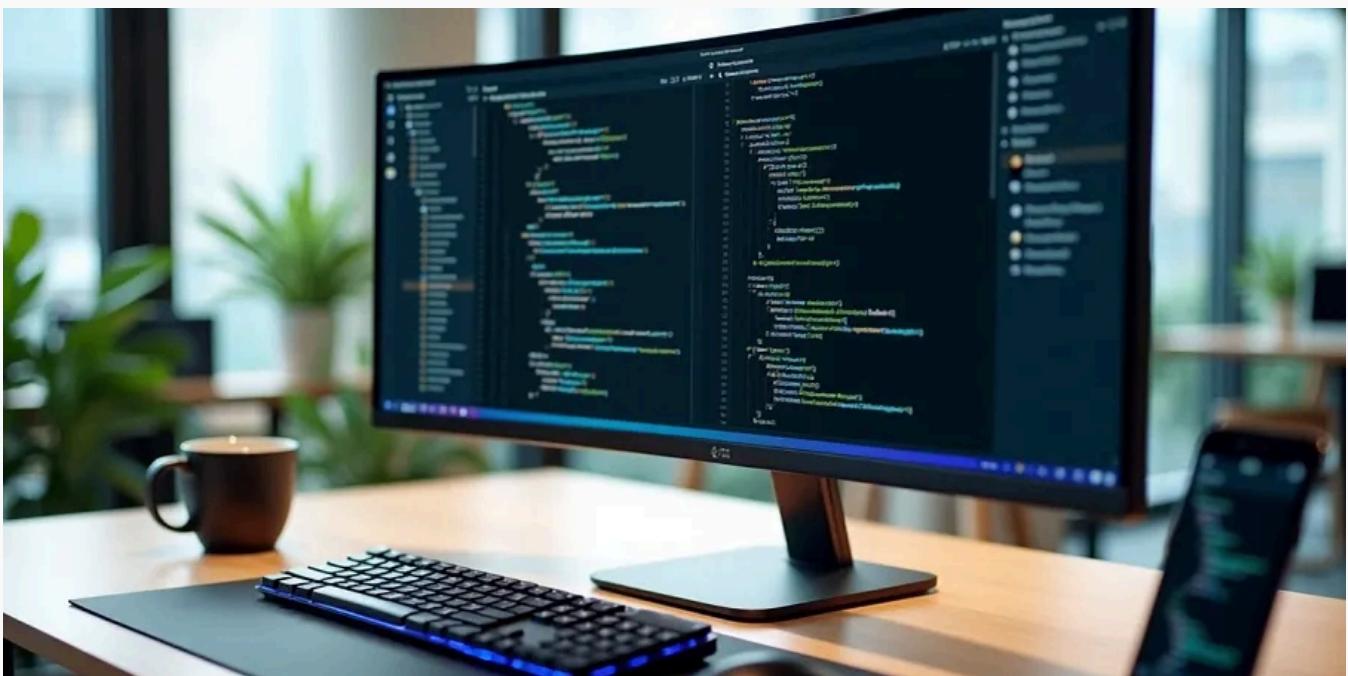
Stop Using UUIDs as Primary Keys. It Is Killing Your Database Throughput

Your database is not slow because it is old. It is slow because every new row lands in a random place that your CPU cannot predict.

★ Nov 24 741 43



...



Kerim Kara

Microsoft Visual Studio 2026: Why C# Developers Are Switching to Blazor Fullstack

4d ago 153



NODE.JS SERIES

TOP 10 CONSIDERATIONS FOR NODE.JS DDD: PERFORMANCE VS COMPLEXITY

Dive Deep into Advanced Techniques



[More Information](#)
arunangshudas.com

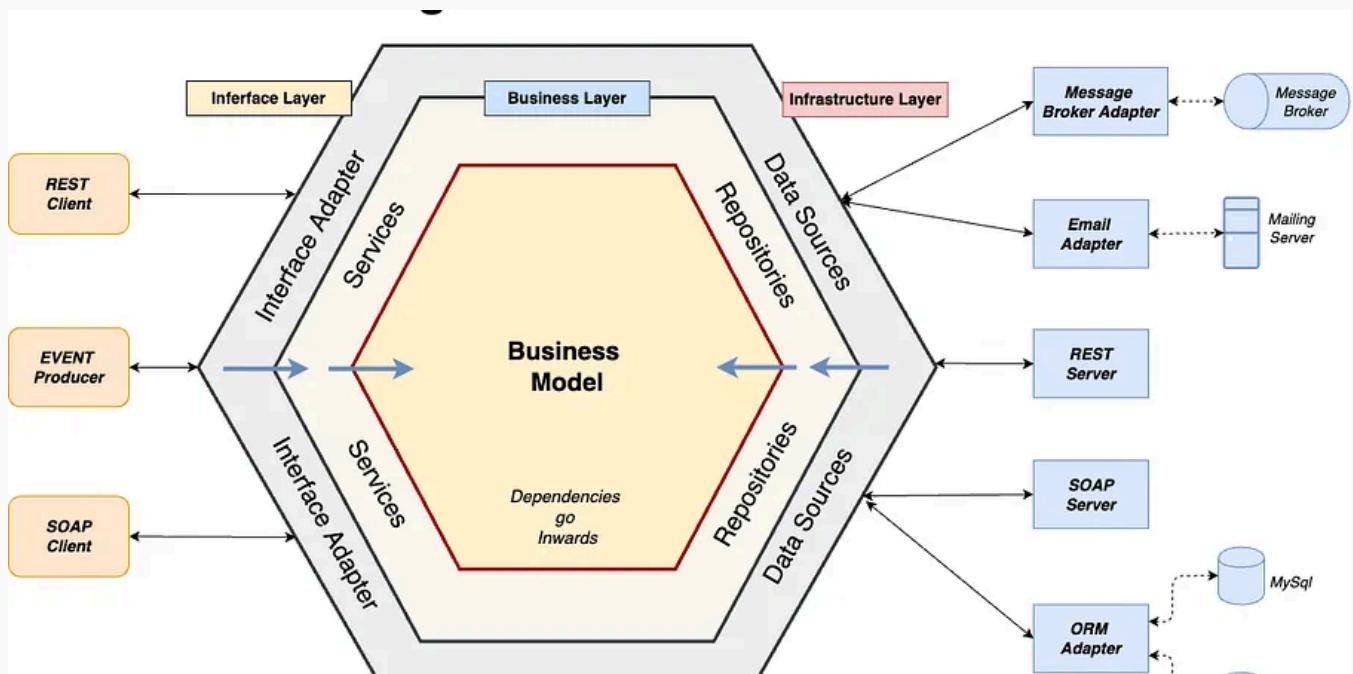


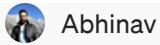
Arunangshu Das

Top 10 Considerations for Node.js DDD: Performance vs Complexity

When you start building a Node.js application, everything feels simple at the beginning. You set up an Express server, connect to a...

Aug 27 2





Abhinav

Why Senior Engineers Are Choosing Hexagonal Over Layered Architecture

Most developers start with the classic Layered Architecture—controller, service, repository, database.

◆ Jul 22

👏 773

💬 52



...

[See more recommendations](#)