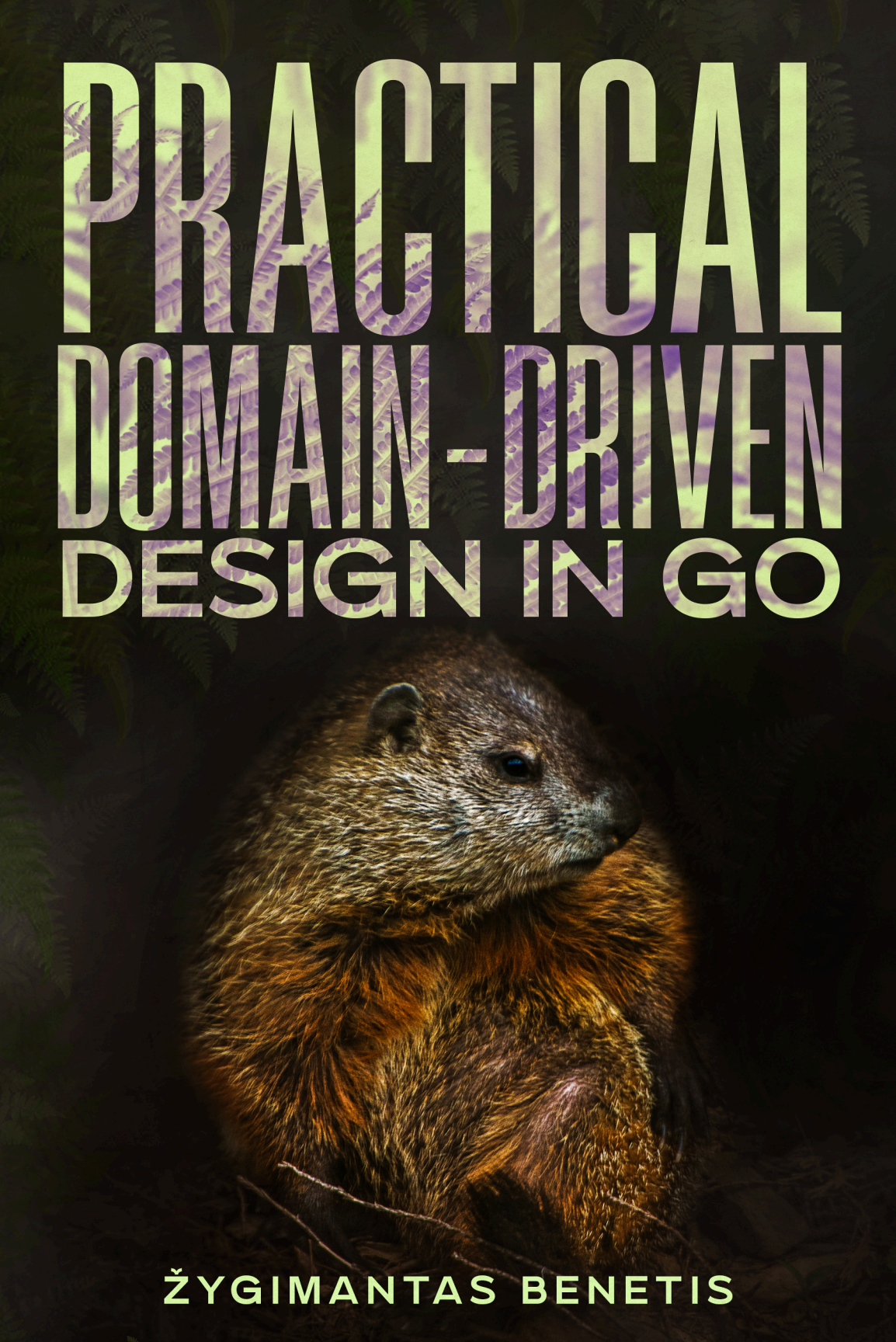
PRACTICAL DOMAIN-DRIVEN DESIGN IN GO



ŽYGIMANTAS BENETIS

Practical Domain-Driven design in Go

Žygimantas Benetis

First partial release 2024-10-14
Last release 2025-06-01

Disclaimer: This book is an early draft and content is subject to change. Around 35-40% is completed. A lot of TODO statements still exist.

Contents

Acknowledgments

While others have provided valuable feedback and reviews, I am solely responsible for any opinionated takes or mistakes in this book.

I would like to thank the following individuals who have contributed to making this book possible:

- **Giedrius Visokinskas**
- **Bartosz Witkowski**

Quite a few libraries and tools and tools were used. Credit goes to:

- LaTeX and the contributors of its many packages
- Excalidraw for creating illustrations
- Excalidraw Libraries, including:
  - Software Architecture by @Youri Tjang
  - System Design Components by @Rohan Pithadiya
  - Office Items by @Robin Müller

Cover:

- @oliviaprodesign

Disclaimer

The code, examples, and information contained in this publication (collectively, the "Materials") are provided on an "as is" and "as available" basis, without any warranties of any kind, whether express or implied, including but not limited to any warranties of merchantability, fitness for a particular purpose, non-infringement, accuracy, completeness, or that the Materials are error-free or will meet your requirements.

You assume all responsibility and risk for the use or misuse of the Materials. The author, contributors, and publisher do not guarantee that the Materials are suitable for your intended purposes or free from defects. You are solely responsible for verifying the code's functionality and accuracy in your own environment and ensuring compliance with all applicable laws and regulations.

Under no circumstances shall the author, contributors, or publisher be liable for any direct, indirect, incidental, consequential, special, exemplary, or punitive damages (including, but not limited to, loss of data, loss of profits, business interruption, or procurement of substitute goods or services) arising out of or relating to the use of, or inability to use, the Materials, even if advised of the possibility of such damages.

Some jurisdictions do not allow the exclusion or limitation of certain damages; in those jurisdictions, liability is limited to the fullest extent permitted by law.

The contents of this publication are for educational and informational purposes only and do not constitute legal, business, or professional advice. You agree to indemnify and hold harmless the author, contributors, and publisher from and against any and all claims, damages, liabilities, costs, and expenses (including reasonable attorneys' fees) arising out of or related to your use of the Materials, your violation of these terms, or your violation of any third-party rights.

This disclaimer, and any dispute arising from it or the Materials, shall be governed by the laws of the jurisdiction in which the author is located, without regard to conflict of law principles, and you agree to submit to the personal and exclusive jurisdiction of the courts located in that jurisdiction.

All rights not expressly granted are reserved by the author and publisher, who may, at any time and without notice, revise or update the Materials and this disclaimer.

If your publication references or includes open-source code or third-party libraries, ensure that the corresponding license terms and notices are properly included.

Chapter 1

Before you begin

1.1 Source code

This book provides the complete source code in a separate zip file, with only key snippets included in the text. To fully understand the examples, explore and interact with the provided code. Exercises offer guidance and ideas for practice. Since books with extensive code blocks can be hard to follow, I leave much of the code exploration to you.

1.2 Domain-Driven design by Eric Evans

DDD was popularized by Eric Evans in his influential book "Domain-Driven Design: Tackling Complexity in the Heart of Software." [1]. This book is heavily based on Evans book.

1.3 Differences from usual DDD

Application layer is small

The application layer is intentionally kept minimal. Its sole purpose is to connect dependencies from other layers and initiate the application. Most commonly, this is not the case, as the application layer in traditional DDD projects often contains use cases that coordinate business logic, enforce application workflows, and act as a mediator between the domain and infrastructure layers. More about application layer in ??.

Warning

This is an opinionated take on application layer.

1.4 Legend

Tip

This is a tip box. Contains general recommendations for positive results.

Warning

This is warning box. Contains warnings about biases, opinioted takes or pitfalls.

Important

This is important box. Contains things to take a note of.

Exercise 1

This is exercise box. Contains exercises reader should attempt to do.

Code block 1: Code example

```
1 func main() {
2   fmt.Println("BUILT... FOR TWO...")
3 }
```

Chapter 2

About this book

2.1 Intended Audience

This book is for developers who want to explore practical ways to use DDD in Go programming language. Whether you're just starting to learn Go or already familiar with it. It focuses on hands-on exercises and shopping cart example to help you understand how Domain-Driven Design (DDD) can be applied in Go projects.

2.2 Prerequisites

Book is designed for readers with at least intermediate-level software engineering skills. To get the most out of the material, be familiar with these topics:

- **Golang:** A good understanding of language. Everything in *A Tour of Go* up to but not including generics.[1].
- **Databases:** Know how to use relational databases for CRUD operations. This includes writing basic SQL queries and understanding how to interact with a database from code.
- **Unit Testing:** Be able to write and run unit tests using testing frameworks. Understand how to create test cases, use assertions.
- **HTTP APIs:** Know how to design and build APIs for frontends or other applications.
- **Distributed Systems:** Basic understanding of distributed systems. Specifically understanding how asynchronous messaging works.

[1] A Tour of Go – `https://go.dev/tour/list`

2.3 What You Will Learn

By the end of this book, you will understand:

- How to structure Go projects using DDD-inspired layers.
- Practical implementation of aggregates, repositories, and domain events.
- Building and exposing APIs, integrating databases, and using messaging systems like Kafka.
- Testing strategies for domain-driven applications.

The book avoids topics like strategic domain modeling or how to collaborate with domain experts. It focuses on implementation and codebase architecture.

2.4 Purpose and Approach

This book takes a practical, hands-on approach to applying DDD principles in Go projects. Rather than focusing on abstract theory, it emphasizes real-world implementation through a case study: a Shopping Cart application. This case study is used throughout the book to demonstrate how DDD concepts translate into Go code.

The book alternates between explaining general software design principles and their direct application in the case study. Code snippets are included where necessary, but you are encouraged to explore the complete source code provided separately for a deeper understanding.

Chapter 3

Introduction to Domain-Driven Design

3.1 What is Domain-Driven Design?

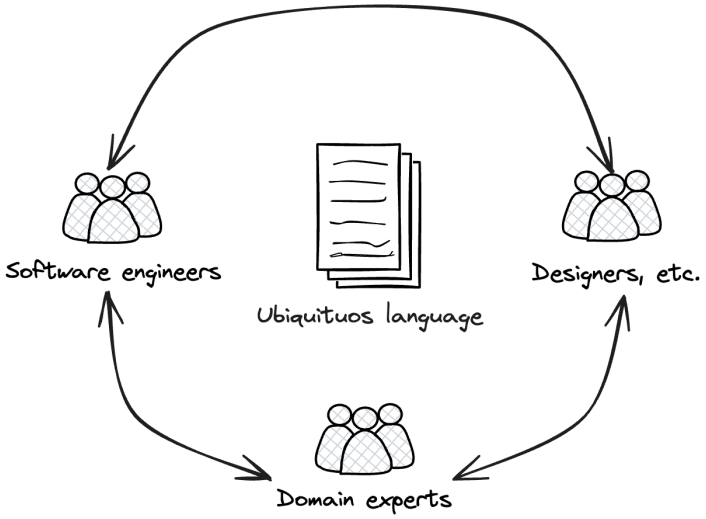


Figure 3.1: Collaboration between domain experts and the rest of the team.

Domain-Driven Design (DDD) is an approach to software development that prioritizes a thorough understanding of the business domain in which the software operates. DDD aims to leverage the knowledge of domain experts to create a model that accurately represents the real-world processes and rules of the domain.

At its core, DDD focuses on creating a shared understanding between technical and non-technical stakeholders, ensuring that the software development process is

guided by the domain experts' [1] insights. This approach involves a collaborative process where developers and domain experts work together to define a common language, known as the *ubiquitous language*[2]. It is consistently used and updated throughout the project lifecycle.

One of the biggest wins from adopting a ubiquitous language is non-technical stakeholder buy-in. By using terms and concepts that are familiar and meaningful to business stakeholders, you create an environment where they feel included and empowered. They no longer have to rely on translators to bridge the gap between technical and business terminology. This shared language not only enhances collaboration but also increases trust and confidence in the development process, ensuring that the final product truly meets the business's needs.

The primary goal of DDD is to break down the complexity of software systems by focusing on the core domain and its logic.

[1] Domain experts are individuals who possess in-depth knowledge and understanding of a specific business domain. They range from Product Managers to Accountants.

[2] The ubiquitous language refers to a common vocabulary shared by all team members, both technical and non-technical. This language is used consistently throughout the project to describe domain concepts and processes, ensuring that everyone uses the same words to represent same objects and concepts.

3.2 Key Aspects

A fundamental aspect of Domain-Driven Design is the separation and isolation of the domain logic from other parts of the system, particularly the infrastructure. This isolation is beneficial for several reasons:

First, it allows the core domain logic to evolve independently of technological concerns. By isolating the domain, developers can focus on capturing and implementing business rules and processes without being constrained by the underlying technical infrastructure. This separation ensures that changes in technology (transport like HTTP, storage, ...) do not require changes in the domain logic. As a result, the system can adapt more easily to new technologies or changes in existing technologies without impacting the core business functionality.

Second, isolating the domain simplifies the testing and maintenance of the system. By having clear boundaries between the domain and infrastructure, teams can develop domain logic in a controlled environment without worrying about external dependencies.

Lastly, this isolation adds purpose and direction to system architecture. It allows teams to organize the system around business capabilities and concerns, rather than technical concerns. This organization not only helps in understanding and communicating the system's structure, but also aligns system design with business strategic goals.

3.3 When to DDD?

While Domain-Driven Design offers significant benefits, it also comes with certain tradeoffs that teams need to consider before adopting it.

3.3.1 Benefits

- **Alignment with business:** By focusing on domain, the codebase remains more aligned with the core business logic, resulting in solutions which support business needs well.
- **Communication:** Ubiquitous language fosters better communication between technical and non-technical stakeholders. There is no need to translate between *technical terms* and *business terms*. Additionally, naming things in software is hard. If naming can be done together with business stakeholders – this opportunity is too good to pass.
- **Manageable complexity:** DDD helps with correct division of complex domains into smaller bounded contexts. By collaborating with business to drive

this division – the risk of incorrect abstractions is reduced. Each bounded context can be developed and evolved independently. This increases team velocity and reduces the risk of introducing errors.

- **Scalability:** As the business evolves, the software can adapt more easily to new requirements and changes. Since software is divided by business domain concerns – it makes it much easier to split team's scope into multiple parts and assign new teams to it.

3.3.2 Challenges in securing organizational buy-in

- **Steep learning curve:** Requires a deep understanding of the domain and a continuous collaboration with domain experts.
- **Alignment and discipline:** DDD involves significant effort in terms of modeling, establishing and maintaining discipline around the ubiquitous language and bounded contexts.
- **Implementation complexity:** Additional code will introduce its own complexity in terms of designing and maintaining the domain model.

Domain-Driven Design is a powerful approach for creating software that puts business needs first and is capable of handling complex domains. However, it requires a significant investment. Consider and weigh these tradeoffs to determine if it is the right approach for the project.

Chapter 4

Introduction to case study

This chapter introduces minimum viable product for an online store. Main goal of this chapter is to show how domain-driven design enables manageable evolution. To keep the case study simple and clear, some complexities such as fraud integrations, quantities, refunds, pagination and etc. are intentionally excluded. Later chapters will introduce new business requirements, demonstrating how the domain and codebase evolve in response.

4.1 Business requirements

Overview

A store clerk creates products by specifying their price and description. Users can then browse through these products, adding any they wish to purchase into their shopping cart. At checkout, the user provides both shipping and payment details before confirming the order. Once payment is verified, the system finalizes the order. After the clerk ships the products, the order is marked as shipped.

Products

- Products are created by the store clerk, with a specified name and price.
- Not logged-in users can view available products.
- Users can filter products by minimum or maximum price, name, and other criteria.

Shopping cart

- A shopping cart may contain one or more products.
- Users can add products to the cart; each product has a maximum quantity of one per cart.
- Cart persist across sessions and devices.
- Checkout is completed for the products in the current cart.
- Single cart per user at all times.

Order

- An order is created after a successful checkout.
- Clerk updates the status of the order to shipped once order is dispatched.
- Orders can be canceled by clerk if product does not exist in store.

Users

- Only logged-in users can interact meaningfully with the system.
- Currently, authorization and permission models are not implemented, meaning all logged-in users have equal permissions.

4.1.1 Naïve approach

Consider the simplest architecture for an online store, as depicted in figure 4.1. This design associates each user directly with a single order, which aggregates multiple concerns: products, payments, invoices and so on.

The process operates as follows:

- A user adds products to their order and clicks the checkout button to place the order.
- A clerk then handles the shipment of the products and updates the order record with the latest shipping and fulfillment information.

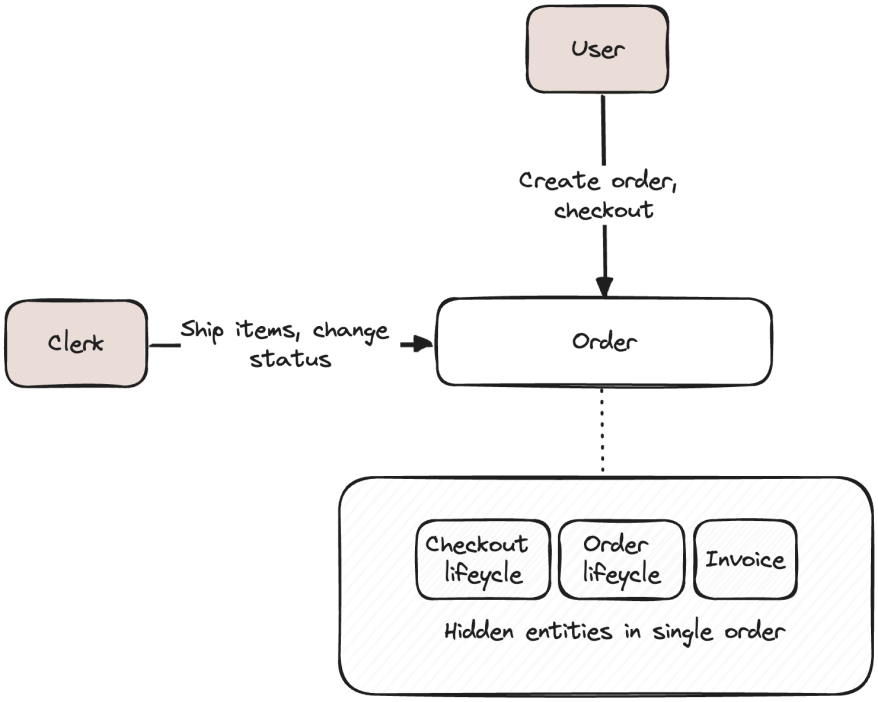


Figure 4.1: Online Store: Naïve approach

This violates the *single responsibility principle* and results in what is called a *God object*, a single entity burdened with too many responsibilities. As a result state management for order becomes complex. Order management will need to manage multiple independent lifecycles:

- Product selection

- Payment processing
- Invoices
- Shipping and fulfillment

While this architecture seems straightforward at first, it quickly becomes hard to maintain and evolve due to complex state management and ambiguous domain boundaries. Each new business requirement introduces additional complexity, and the system becomes increasingly difficult to understand and extend. To modify one part of the system, knowledge of the rest of order is required as small changes can break some workflows.

4.1.2 First iteration

Let's have business drive this architecture further. First, business requirements clearly mention a shopping cart and business understands what it is. Shopping cart has its own lifecycle, which is distinct from that of the order. By explicitly modeling the shopping cart and order as separate domain aggregates, software is better aligned with business concepts. Visible in figure 4.2.

The process operates as follows:

- A user interacts with the shopping cart by adding products and initiating payment through the payments API.
- Once the shopping cart is paid for, an order is created from the cart.
- A clerk then takes over the order, handling the shipment and updating the order status accordingly.

Each aggregate now has a distinct lifecycle, properties, and boundaries. The resulting architecture is easier to understand, clearly separates concerns, and is more resilient to future changes.

Tip

In real–world scenarios, naïve architectures typically require multiple iterations to achieve resilience and flexibility against future changes. Don't settle for an overly simplistic model. Iterate until the domain boundaries and aggregates cannot be improved upon. [4]

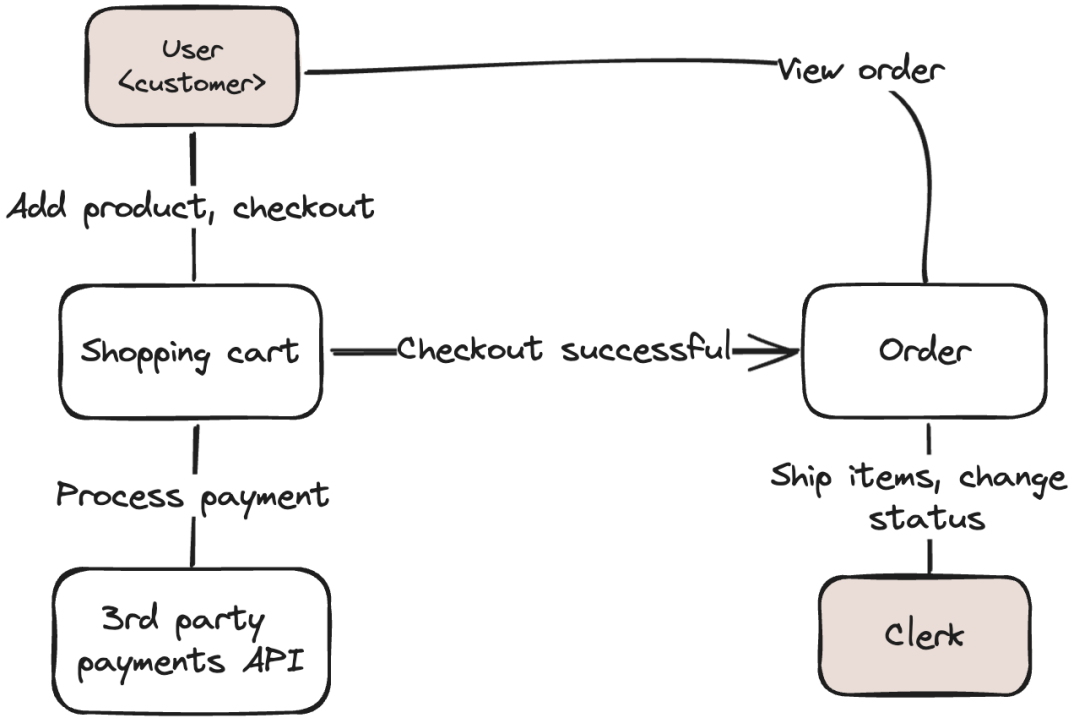


Figure 4.2: Online Store: Architecture after first iteration

Exercise 2

Inspect existing code. How is user represented in code? Is clerk represented in code? Should it be? What if we want to track who is handling the `order`? Is ubiquitous language choice good choice here?

Exercise 3

Try modelling `cancel order` flow on your own. What happens with payments?

Chapter 5

Layers: Business Domain Layer

One of the key principles of Domain-Driven Design is organizing the codebase around the domain model, where the business domain and its data structures form the core of the system.

The primary goal is to isolate business logic from other code, creating strong boundaries. Eric Evans, in his book *Domain-Driven Design* [1], introduces the idea of layers to decouple models from technical software concerns, ensuring the purity of domain logic.

By placing the domain at the center of the architecture and minimizing its dependencies, we ensure that changes in business logic propagate outward without being affected by external changes. In figure 5.1, the green boxes represent the domain layer, while the yellow boxes symbolize the infrastructure layer (abbreviated as "infra"). Dependencies only point inward towards the domain, reinforcing its independence.

This chapter will guide you through how to effectively structure the business domain layer, how to model domain services, decouple it from infrastructure concerns. This will result in a solid understanding of how to organize code around capabilities rather than individual entities.

Lastly, this chapter will set the foundation for the discussions about the infrastructure and application layers, which are covered in chapter 10 and **??**.

Exercise 4

Explore layers in accompanying online store codebase. See how layers are structured as folders or Go packages.

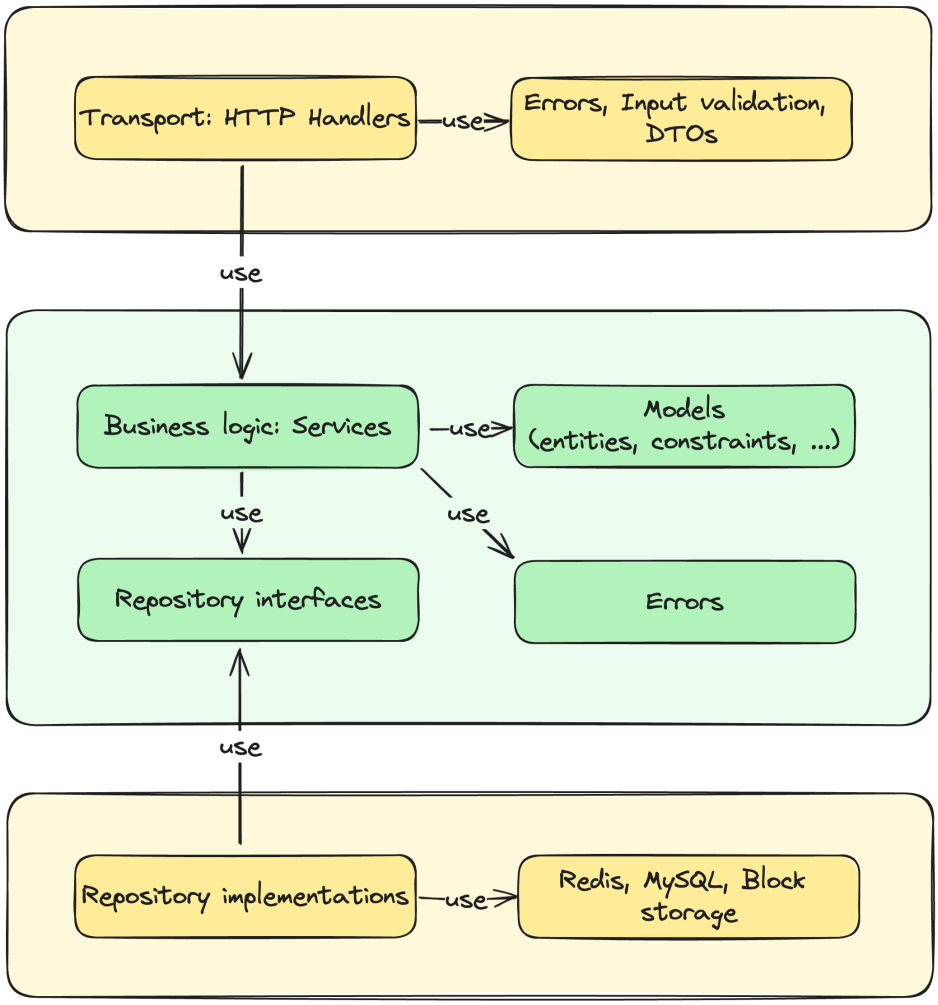


Figure 5.1: Layers: Isolated business domain

5.1 Domain models

At the heart of every system are the domain models. They capture most business concepts in data structures, such as an `Item`, as seen in code example 2.

In figure 5.1, you can see that these models are self-contained. Depending solely on other domain data structures or, at most, shared library types. Because most of the system relies on these data structures, it is important to get them right.

> Bad programmers worry about the code. Good programmers worry about data structures and their relationships.
>
> — Linus Torvalds

I believe Linus Torvalds's words fit perfectly in this context. Prioritize data structures and relationships. With good domain models, you leverage this approach even more. That is because – first your local program benefits and second – the rest of the layers in a system benefit from this too.

Another approach to navigating domain models is through the concept of *aggregates*. In DDD, the term *aggregate* refers to a cluster of domain entities and value objects with clearly defined boundaries. Aggregates encapsulate business logic and ensure that invariants—rules that maintain the integrity of the model—are consistently enforced across their related objects (more about invariants in chapter 7).

Aggregates represent the smallest units of query and the minimal units of work for functions within the application, helping to maintain model consistency and optimize performance. Additional details about aggregates are covered in chapter 6.

Tip

One of the most effective ways to unpack a complex domain is through event storming[a]. By identifying aggregates and the events that impact them, this method serves as a powerful tool for understanding and organizing domain models.

[a] `https://en.wikipedia.org/wiki/Event_storming`

5.1.1 Product examples

`Product` aggregate in code example 2 is expressed as an entity[1] and two fields. `Price` is expressed by the value object `Money`.

[1] Entity - because it has a unique ID and is compared to other items by that ID. More about equality and identity in chapter 8.

Code block 2: Product aggregate

```
1  package product
2
3  type Product struct {
4    ID    ID
5    Name  string
6    Price money.Money
7  }
8
9  func NewProduct(productID ID, name string, price money.Money) Product {
10   return Product{ID: productID, Name: name, Price: price}
11 }
```

Anytime we work with `Product` in any business context – this data type is used. It does't contain any information about how its stored or accessed, only what is relevant for business logic. Worth mentioning that this data type is only suitable for already existing `Products`, because ID is required parameter.

There will be a domain service which calls repositories and constructs a list of these `Products`. They can be used in business logic later on. More about this in following chapters.

Code block 3: Upload product

```
1  package product
2
3  type Upload struct {
4    Name  string
5    Price money.Money
6  }
7
8  func NewUpload(name string, price money.Money) Upload {
9    return Upload{Name: name, Price: price}
10 }
11
```

What if clerk wants to upload a new `Product`? Some data, like `ID` or other fields might only be available after `Product` is already created. Or some fields are not needed after they are validated. Due this, we need another data structure to represent an `Product` being uploaded. Take a look at `product.Upload` in code example 3. If looked from another perspective – this can be seen as a command. More about this in chapter 14.

Without any *actions* that happen to these types it is quite hard to understand what's happening. Feel free to hop to section 5.2 and inspect.

5.1.2 Clarify meaning of ”model” in codebase

In DDD, noun *model* can mean services, entities, value objects and modules [1]. Due the overloaded nature of this term, it might more sense to have a more specific name for model in codebase. Specifically, we will constraints ourselves and call entities and value objects as models and exclude services and modules from this definition. Entities and value objects tend to frequently mix up in same files and there is not much value in having different names for them in a project. More about services in section 5.2, more about entities and value objects in chapter 6.

Entities and value objects will be side-effect free[2] and only contain things related to business logic. This makes them easy to understand, extend, move around and in general quite powerful.

5.1.3 Models in Online Store 1.0

Given the requirements, there seem to be three main root aggregates: *Product*, *Shopping Cart*, *Order*. `Product` is briefly reviewed in subsection 5.1.1.

> Tip
>
> As a general rule its better to have architecture around capabilities instead of entities or aggregates. For better examples, this rule will be ignored for now. Example: Instead of just having `Order` in diagrams, we have `Place Order` and `Ship Order` capabilities.

Shopping Cart model

Cart represents a shopping cart to which products can be added for a specific user. Any time this user can fetch this cart and see what items are inside. Checkout can be initiated for this cart. Checkout status is updated and saved on the cart. If it is successful – Order is created. After payment is initiated, cart is no longer active. Meaning that adding new items to cart will create a new cart.

Struct representing this cart without items and with items visible in code example 4. *WithItems* version includes cart items as well cart itself. Most of the time this is what is needed to work with business logic. But there is performance price to pay for fetching items and its not always needed.

[2]Side-effect free means that the methods on these structs do not alter the state of the system or have observable interactions with outside systems. They only return values based on their inputs.

Tip

In Go, package names provide an intuitive way to show how models fit into the system. For example, `shoppingcart.WithItems` clearly indicates that the `WithItems` type belongs to the `shoppingcart` package. For the reader it reads *Shopping cart with items* which is exactly what we want. With `shoppingcart.Cart` there is a bit of repetition, but we can accept this tradeoff. More about package names in this blogpost [a].

[a] `https://go.dev/blog/package-names`

Cart properties can be simplified using a computed field, such as `State`. Having this computed field makes validation simpler, as services only need to verify the current state. There is no need to persist this field directly in the database; instead, timestamps are stored, and the `State` is calculated ephemerally upon retrieving the data from the repository.

Alternative would be to implement full state machine with explicit state transitions. But since this is `1.0 version` – simplicity is preferred.

Code block 4: Cart domain model

```
type Cart struct {
    ID                  ID
    UserID              model.UserID
    Status              Status
    CheckoutInitiatedAt *time.Time
    CheckoutAmount      *utils.Money
    CheckoutSucceededAt *time.Time
    CheckoutFailedAt    *time.Time
}

type WithItems struct {
    Cart  Cart
    Items []Item
}

func NewCartWithItems(cart Cart, items []Item) *WithItems {
    return &WithItems{Cart: cart, Items: items}
}
```

Exercise 5

Add function which calculates shipping fee of 5€ and 2% of total cart amount.

Exercise 6

What if we need to introduce holiday packaging as an additional service that products can be wrapped in? What would need to change?

Order model

TODO: more info about order. Maybe I need more explanation for all aggregates and whole architecture?

`Order` holds information about an order which is being fulfilled. It is already paid and in some kind of status where its being shipped or is already cancelled.

`Order` does not have `OrderItem` as `Cart` does, as seen in code example 5. This is done for simplicity sake at this point of the application. This introduces implicit coupling, as seen in figure 5.2. `Order` has `Cart` ID which can be used to fetch `Items` and they can be used to fetch `Products`.

This is a tradeoff in architecture and design – this coupling makes code simpler, but will make it difficult to change it in the future. `Product` and `shoppingcart.Item` changes will automatically break `Order`. However, that is the tradeoff which is acceptable for `1.0 version`. As architecture and business requirements will evolve, so will these models.

Code block 5: Order domain model

```
type Order struct {
  ID     ID
  UserID model.UserID
  CartID shoppingcart.ID
  Status Status
}

type WithProducts struct {
  Order Order
  Products []product.Product
}
```

Warning

Creating aggregates that contain only IDs, as shown in code example 5 goes against ideal principles of DDD. The proper way is to have complete copies of related data embedded. However, having IDs an acceptable performance trade-off for aggregates that do not require all related data. More aggregates can be created where this data is embedded.

Exercise 7

What kind of queries happen when `order.WithProducts` is returned? Think and inspect code. What can be done to make this more optimal?

Exercise 8

Advanced: What would need to happen so `product.Product` can evolve on its own without changes needed for `Order`. What about `shoppingcart.Item`?

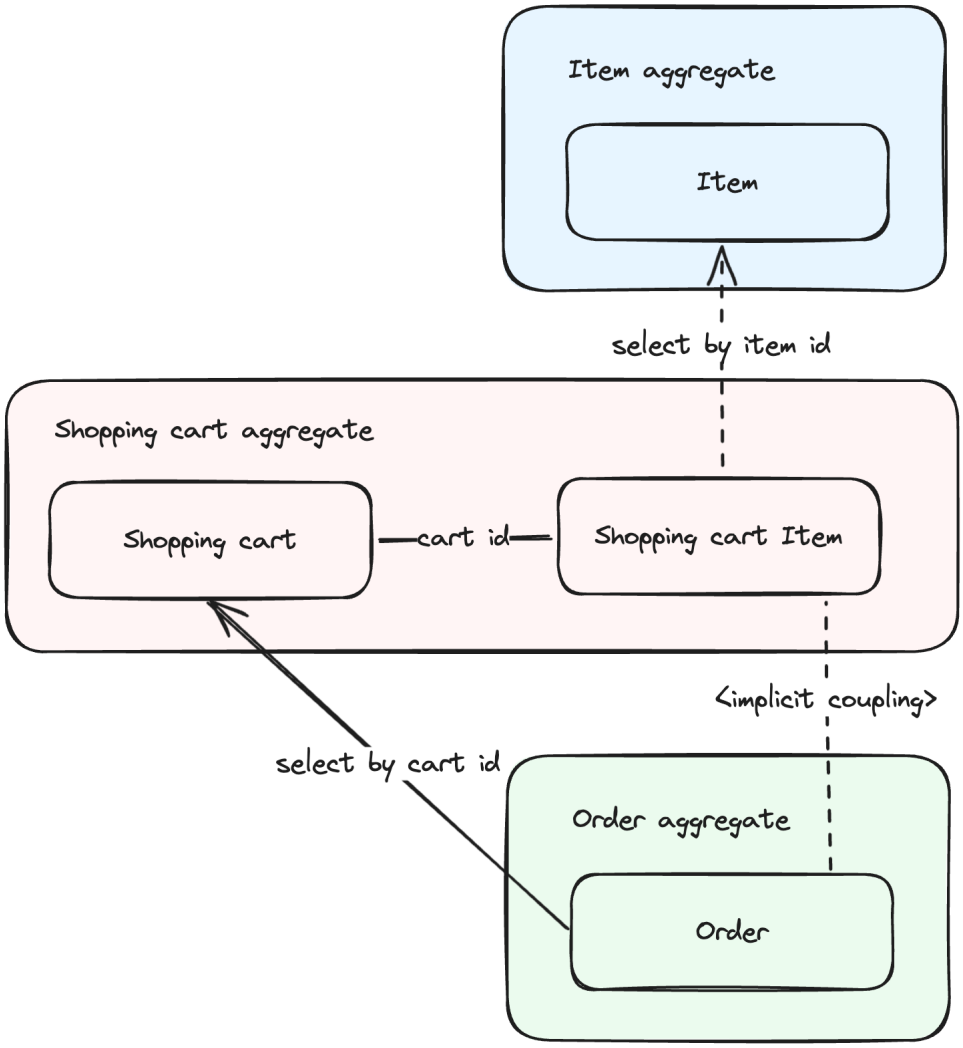


Figure 5.2: Online store 1.0 aggregates

5.2 Domain services

Based on existing entities , domain logic can be put into three services: *InventoryService, CartService, OrderService*. This design pattern is known as the *Domain Service* pattern[3]. If services become too large or or functionality is hard to fit into them – they can be split. Or another pattern like *Interactor* can be used.

Important

Domain services only contain business domain logic.

Possible domain operations or domain algebra can be inspected in code example 6.

Code block 6: What's possible in domain, by interfaces

```
type CartService interface {
  AddProducts(userID model.UserID, products []product.ID) error
  GetCart(filter shoppingcart.Filter) (shoppingcart.WithItems, error)
  InitiateCheckout(id model.UserID) (client.StripeClientSecret, error)
  CompleteCheckout(ctx context.Context, id shoppingcart.ID, status shoppingcart.Status) erro
}

type InventoryService interface {
  CreateProduct(product product.Create) (product.Product, error)
  ListProducts(filter product.Filter) ([]product.Product, error)
}

type OrderService interface {
  CreateOrder(ctx context.Context, userID model.UserID, cartID shoppingcart.ID) (order.Order
  ListOrdersWithProducts(filter order.Filter) ([]order.WithProducts, error)
  UpdateOrderStatus(orderID order.ID, status order.Status) error
}
```

These interfaces already signal what is possible to do with *Cart, Product and Order*. It quite likely that some adjustments will need to be made once more requirements are refined. But after all development is done, this serves as a great documentation on capabilities and makes business logic easy to understand.

From method signature it is visible that there can be only one cart active per one user. That is indeed a case. Calling HTTP endpoint to add products will use user authentication token to grab user ID. That ID is used to fetch cart and other entities.

[3]In Domain-Driven Design (DDD), a domain service is a service that encapsulates domain logic that doesn't naturally fit within an entity or value object. It helps structure business logic by handling operations that involve multiple entities or that are not naturally part of any entity.

Tip

Outlining service interfaces is a great way to start modelling business domain.

Exercise 9

Try modelling an interface of a refund service. Hint: One function might be enough.

Warning

Be aware that defining interfaces in this way goes against standard Go practices of returning concrete types and not defining interfaces on implementation side [a].
Having this service interface is opinionated take. We want tight coupling here and original reasoning to avoid this does not apply here.

[a] https://go.dev/wiki/CodeReviewComments#interfaces

5.2.1 Implementation example

Domain services don't store any state themselves, only orchestrate it. Repositories and external clients will store and manage state instead. Dependencies will range from repositories, to clients to other services.

Warning

Other services in dependencies can be a signal for coupling. In this case it is. At later chapters this will be solved by better boundaries and events.

AddProducts method which will add products to existing cart can be inspected in code example 7. If no active cart exists for current user – a new cart will be created.

ID for *Cart* is not generated by repository, but instead it is created here, in business layer. ID is passed to repository as is. More about Entity IDs section chapter 9.

Repository functions can be more generic, but usually they have some specific use cases. In this function both *Cart and Product* repositories are used. Cart is obvious and products repository is needed to validate if submitted products actually exist.

Cart items are created lastly. Nothing runs in database transaction here, because operations are designed to be idempotent. If they fail in the middle – repeating it will fix it.

This method shows that Cart is an aggregate as well, because entities *Cart* and *CartItems* are treated as one consistent *Cart* model and updated together.

Exercise 10

Add **remove items from cart** function to Cart service. Repository changes are optional.

Exercise 11

If shopping cart is empty – create new cart instead of returning `404`.

Code block 7: CartService implementation

```go
type CartServiceImpl struct {
  CartRepository    repository.CartRepository
  ProductRepository repository.ProductRepository
  StripeClient      client.StripeClient
  temporal          temporalclient.Client
}

func (s *CartServiceImpl) AddProducts(
  userID model.UserID,
  productIDs []product.ID,
) error {
  cart, found, err := s.CartRepository.GetCartByUserId(userID)
  if err != nil {
    return fmt.Errorf("failed to get cart by user ID: %w", err)
  }

  if found == false { // Creating cart is idempotent here
    id, err := typeid.New[shoppingcart.ID]()
    if err != nil {
      return err
    }

    cart = shoppingcart.CreateCart(id, userID)
    if err := s.CartRepository.CreateCart(cart); err != nil {
      return fmt.Errorf("failed to create cart: %w", err)
    }
  }

  // Validates if products exist
  products, err := s.ProductRepository.ListProducts(product.Filter{IDs: &productIDs})
  if err != nil {
    return err
  }

  if products == nil || len(products) != len(productIDs) {
    return domainerror.ProductNotFoundError{}
  }

  cartItems := make([]shoppingcart.Item, len(productIDs))

  for i, currItem := range products {
    id, err := typeid.New[shoppingcart.ItemID]()
    if err != nil {
      return err
    }

    cartItems[i] = shoppingcart.NewItem(id, currItem, cart.ID)
  }

  return s.CartRepository.AddItems(cartItems)
}
```

Chapter 6

(empty) Aggregates, Entities, Value Objects

TODO!

Chapter 7

Invariants

7.1 Introduction

An invariant is a logical assertion that must always hold true in a system. Invariants act as guardrails, ensuring that domain rules are respected and preventing invalid states from propagating through the application. Identifying and encoding invariants into the domain logic is important for maintaining correctness and clarity in a system.

7.2 `Money` type

In financial applications, handling monetary values accurately is crucial. A key invariant is that monetary amounts must not be negative in contexts where negative amounts are invalid (e.g., balances, prices). This ensures that the system does not process invalid transactions or represent impossible states.

Requirements

Let's consider the following requirements for a `Money` type:

- It represents a monetary amount in a specific currency.
- The amount should always be a non-negative value, representing the smallest unit of currency (e.g., cents).
- All operations on `Money` should adhere to this invariant.

Formally:

- $amount \in \mathbb{N}_0$, where $\mathbb{N}_0$ is the set of non-negative integers (including zero).
- $Currency$ represents the currency (e.g., EUR).
- $Money = (amount, Currency)$ represents a Money value.

The invariant condition is expressed as: amount ≥ 0

Implementation in Go We can enforce this invariant by defining a `Money` struct and associated methods that validate the amount upon creation and during operations.

Code block 8: Money type implementation

```go
type Amount int64 // smallest unit of currency (e.g., cents).
type Currency string

const (
  EUR Currency = "EUR"
)

type Money struct {
  Amount   Amount
  Currency Currency
}

func NewMoney(amount Amount) (Money, error) {
  if amount < 0 {
    return Money{}, domainerror.MoneyNegativeAmount{}
  }

  return Money{Amount: amount, Currency: EUR}, nil
}

func EmptyMoney() Money {
  return Money{Amount: 0, Currency: EUR}
}

func (m Money) Plus(other Money) Money {
  return Money{Amount: m.Amount + other.Amount, Currency: m.Currency}
}
```

In code example 8:

- The `NewMoney` function enforces the invariant $amount \geq 0$ by returning an error if a negative amount is provided. Using this constructor function ensures that all `Money` instances adhere to the same rules.

- Introducing a type alias for `Amount` increases type safety and makes the code more explicit by clearly indicating that the value represents a monetary amount in cents.
- Amounts are stored as `int64`, representing monetary values in the currency's smallest units (e.g., cents). This approach ensures precision and avoids issues related to floating-point arithmetic, which can introduce rounding errors.
- The `Plus` method adds two `Money` amounts together, maintaining the invariant since the sum of two non-negative integers is non-negative.
- This money pattern or implementation can sometimes referred as Fowler's Money pattern [2].

Tip

Prioritize readability. While enforcing invariants, aim to improve clarity and understanding, making your code easier to read and maintain. If the code becomes overly complex, it might be better to step back and do it in a simpler way. Not all business rules can or should be centralized [3].

Warning

Go does not have built-in constructors. To disallow incorrect states, we use the `NewType` convention for constructing instances that enforce invariants.

Exercise 12

Extend the `Money` type to support multiple currencies. Modify the `Plus` method to disallow operations between `Money` instances of different currencies. Ensure that `Money` remains simple and side-effect free.

7.3 Add products to cart example

In an online store, it is important to ensure that only existing products can be added to a user's shopping cart. This invariant guarantees that the cart contains only valid products, preventing issues such as processing orders for non-existent products or displaying incorrect information to the user.

Anytime a user wants to add product to their cart, we need to ensure the following precondition: All product IDs the user wants to add must exist in the inventory. This can be written as: $CartItems \subseteq InventoryProducts$.

Implementation in Go We enforce this invariant within a service method that handles adding all products to the cart. Before adding, we validate that each product exists.

Code block 9: Validate if products exist before adding them to cart

```
1  func (s *CartServiceImpl) AddProducts(
2    userID model.UserID,
3    productIDs []product.ID,
4  ) error {
5    // ...
6    products, err := s.ItemRepository.ListProducts(product.Filter{IDs: &productIDs})
7    if err != nil {
8      return err
9    }
10
11   if products == nil || len(products) != len(productIDs) {
12     return domainerror.ProductNotFoundError{}
13   }
14   // ...
15
16 }
```

In code example 9:

- The method retrieves the list of products based on the provided `productIDs`.
- It checks if the number of products retrieved matches the number of `productIDs` provided.
- If any products are missing, it returns an `ProductNotFoundError`, enforcing the invariant that only existing products can be added to the cart.

All of this happens in a domain code, which is kept minimal and easy to understand. More about domain code in chapter 5. It is also simple to test and error is easy to work with. More about isolating domain code in chapter 5.

Exercise 13

Extend the `AddProducts` method to handle a scenario where only some of the requested product IDs exist.

7.4 Type System

The type system plays a important role in a DDD codebase. It helps to define clear models and distinguish types between layers or domains. Without explicit

types, managing different domains that share the same entity but have different properties becomes difficult.

For example, consider two domains: `Profile` and `Shopping Cart`. Both have a similar entity to generic `User`, but the purpose and meaning is different.

In the `Shopping Cart` domain, `User` might mean anyone who has an item in a shopping cart, registered or not. `User` might have fields like a `anonymous ID` and `country`.

`Profile` domain might mean only the users which are registered and have their profile information filled in. It can have information like `ID` and `Name`.

These two representations of a user need to coexist, and in some parts of the codebase, they must interact. Types help guide the developer and prevent accidental mistakes. See the code example 10 for an illustration.

Code block 10: User vs ShoppingCartUser in Go

```go
type ProfileUser struct {
    ID   uuid.UUID
    Name string
}
type ShoppingCartUser struct {
    ID uuid.UUID
    ConfidenceRating fraud.Rating
}
```

In dynamically typed languages, the lack of compile-time type checking can blur distinctions between domain models. Without explicit type boundaries, it becomes easy to pass incompatible objects into functions expecting a different type. Mistakes of this kind are typically discovered only at runtime.

Code block 11: Example of potential type confusion

```ruby
class ProfileUser
  attr_accessor :id, :name

  def initialize(id, name)
    @id = id
    @name = name
  end
end

class ShoppingCartUser
  attr_accessor :id, :ip_assigned_country

  def initialize(id, ip_assigned_country)
    @id = id
    @ip_assigned_country = ip_assigned_country
  end
end

def view_cart(user)
  puts "Cart for #{user.anon_id}, Shipping in #{user.ip_assigned_country}"
end

user = ProfileUser.new('1234', 'Alice')
cart_user = ShoppingCartUser.new('1234', 'Germany')

view_cart(cart_user)  # Works fine
view_cart(user)       # Fails at runtime
```

By contrast, statically typed languages enforce explicit boundaries and structural distinctions between domain models. Such issues are typically detected during compilation, helping developers quickly identify and resolve potential mistakes.

Chapter 8

(empty) Identity and Equality

TODO! touch upon default values in Go.

Chapter 9

Entity IDs

An *entity* is an object that can be uniquely identified within a system. Although various techniques exist for assigning and managing such identifiers, it is common in practice for entity IDs to be automatically generated by the database and then relied upon directly, often with limited domain-driven design considerations.

Dedicated and unique entity IDs

In most software systems, entities are assigned dedicated identifiers intended to be unique. Nevertheless, even commonly used identifiers—such as personal IDs or SSNs—may not be as reliably unique as one might assume.[1] Faced with these imperfections, modern non-functional requirements often mandate the use of fully unique IDs. A practical response is to introduce dedicated identifiers to the domain. Such as Universally Unique Identifier (UUID), that guarantees global uniqueness. At first glance, introducing a dedicated global identifier might seem to conflict with DDD principles, which emphasize modeling concepts directly from the business domain. However, if the domain's ubiquitous language and bounded contexts can accept and integrate these global IDs, it can work well. From a system-wide perspective, having a single, universally recognizable ID for each entity is a productivity enabler. This global ID can be referenced across all layers and services, reducing confusion and ambiguity.

Globally unique IDs simplify many operational tasks. Logging, debugging, and cross-service tracing all become more efficient and reliable when one can follow a single ID through the entire stack. When entities cross domain boundaries—such as a *Cart* entity referenced within an *Invoices* bounded context—this same global UUID ensures a consistent and clear link. Even if another bounded context maintains a changed projection of an entity, it can still store the original global ID for

[1] `https://www.computerworld.com/article/1687803/not-so-unique.html`

later use.

One drawback of generating UUIDs purely in the infrastructure or database layer is that such identifiers may lack meaningful domain semantics. While they guarantees uniqueness, they do not inherently convey business intent. It might be more valuable to generate IDs within the domain layer itself. This way, the format, semantics, and even lifecycle of the ID can be influenced by the domain model, leading to identifiers that are both uniquely identifiable and contextually meaningful.

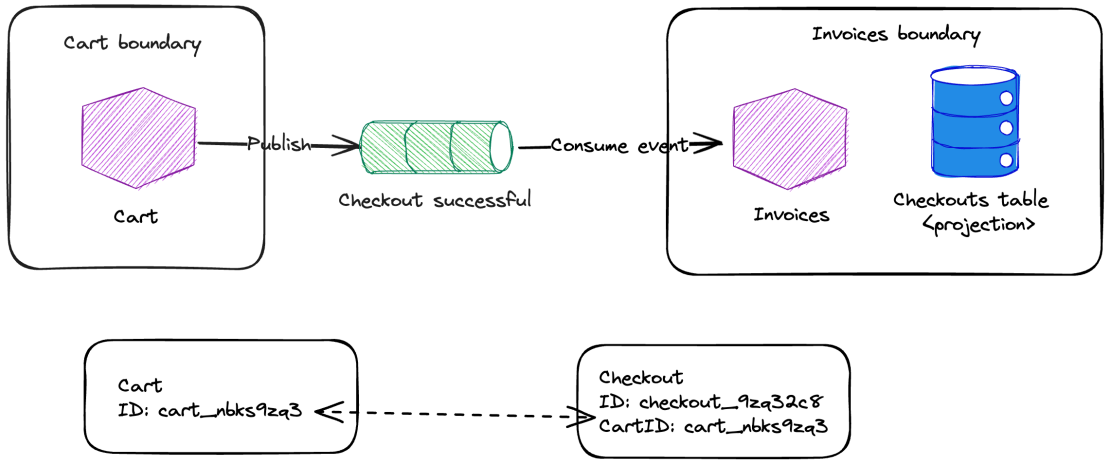


Figure 9.1: Entity IDs: Globally unique IDs help convey meaning across boundaries

For example, consider an *Invoices* microservice that generates invoices from carts after successful checkouts.[2] Even if the invoices domain only has a partial understanding of what a *Cart* is, it can still store and reference the globally unique *Cart ID*. This is visible in figure 9.1.

9.1 Human-Readable UUIDs

Stripe has an elegant solution[3]: they prefix UUIDs with the entity type. This adds additional domain meaning to those IDs. Context like this is helpful to the reader, and this approach still retains all the benefits of UUIDs. For example:

Generic: `123e4567-e89b-12d3-a456-426614174000`

[2]Within the invoices boundary, these completed carts might simply be referred to as "checkouts."

[3]https://docs.stripe.com/api/checkout/sessions

With context: `cart_01j3cckmw5fnnbks9zq32c8wg0`

Stripe goes one step further in terms of user experience: for API keys, they also prefix them with either `_live` or `_test` to indicate if it is a production or test environment key. For example, their API keys look like:

More context: `sk_test_s1Oinnhs1cMZOo3xTUK1SpCw0050l85dpQ`

This way, they add even more context to the keys and improve the developer experience.

9.2 TypeID

To generate IDs similar to those discussed, a TypeID Go package [4] will be used. Its a great library to provide needed functionality. Given an entity like *Cart*, TypeID can be configured as:

Code block 12: TypeID configuration

```
type Prefix struct{}

func (Prefix) Prefix() string { return "cart" }

type ID struct {
    typeid.TypeID[Prefix]
}

type Cart struct {
    ID  ID
    ...
}

```

And new IDs generated with:

Code block 13: Generate new IDs

```
id, err := typeid.New[shoppingcart.ID]()
```

It is quite likely that this ID will be generated in business domain. After generation it will be passed down to repository layer and saved in storage.

[4] https://github.com/jetpack-io/typeid

Exercise 14

In online an store example, change UserID from string type to typeid enabled UUID.

9.3 Performance considerations

Performance limitations places non-functional constraints on business domain capabilities as well. Given codebase complex enough to use DDD, there is usually a need to work with distributed systems. Luckily, UUIDs can also result in better performance than traditional *auto-increment* IDs.

Ordered vs Random

UUIDv4 relies solely on random numbers, ensuring uniqueness without any coordination across systems. However, this randomness comes with a trade-off in database performance. When used as primary keys, UUIDv4 values are inserted into database indexes (e.g., B-trees) in a non-sequential manner, causing frequent tree rebalancing and resulting in slower inserts.

By contrast, UUIDv7 incorporates a time-ordered component, producing identifiers that are lexically sorted when generated. This ordering makes index updates more efficient because new records tend to append at the end of the index rather instead of expensive restructuring. As a result, writes become more performant.

However, the inclusion of timestamp data in UUIDv7 does mean that the generation time is embedded within the identifier. This exposes various timings and activity levels about the system. Usually this is not a problem, but worth keeping in mind.

Distributed Systems

In distributed systems, UUIDv7 brings a lot of benefits. By using a timestamp-based approach combined with randomness, multiple nodes can generate globally unique and lexically ordered identifiers independently, without a central coordination service. This avoids bottlenecks and single points of failure that arise with strictly incremental identifiers.

Chapter 10

Layers: Infrastructure layer

Infrastructure layer's purpose is to provide capabilities and support business domain.

Simplified, infrastructure layer can be divided into two: what invokes business domain and what domain uses to interact with everything else. Business domain is usually invoked from communication protocols. Other domain needs like storage or 3rd party API support is also provided by infrastructure layer.

Code written in infrastructure layer will usually not be generic. Shared layer will contain fully generic code, more detail in section section 10.4.

Infra layer will usually contain most specialized code which is not related to domain. Handlers to invoke specific business function or repository to return specific query results for business logic. Those repositories are not part of domain, because business does not concern with what database or storage is being used. It is a useful split and makes business domain model smaller in scope.

In this chapter, we will explore how communication mechanisms, such as HTTP, connect to the domain layer with the help of the infrastructure layer. We will also examine how repositories and external services are implemented to provide stateful operations for the domain.

10.1 Communications: HTTP

The specific technology used for data transportation (`HTTP`, `gRPC`, `Kafka`, etc.) typically does not impact the core business domain. These communication details should be kept separate from the domain logic.

Important

In DDD, the goal is to clearly isolate the domain from communication layers, primarily to maintain simplicity and clarity. Explicitly enabling easy switching between communication technologies is a not a goal.

10.1.1 Code generation

Each layer defines codecs either by auto-generation or manual definition in Go, depending on the situation. The declarative approach of auto-generating is typically preferred, as it ensures consistency with the specification and reduces the chance of errors. However, the downside is that the entire specification must fit into a predefined data structure and generated code might not always be optimal.

For HTTP, OpenAPI can be used to describe web service specification, visible in block code example 14. Generated code can be seen in block code example 15.

Go has good library ecosystem that can be leveraged here. Specifically, `oapi-codegen` [1], that allows working with HTTP in more declarative style. This library allows generating Go handlers and structs from OpenAPI specification. Flow visible in figure 10.1.

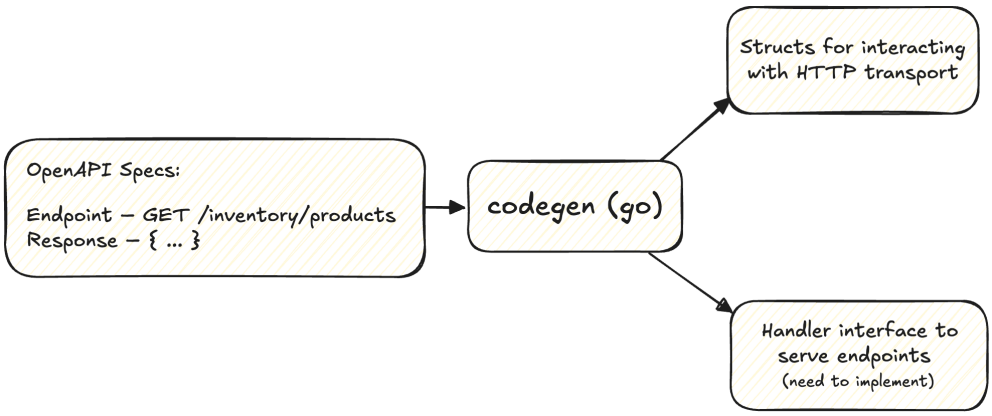


Figure 10.1: Generated Go code to interact with HTTP

[1] `https://github.com/oapi-codegen/oapi-codegen`

Code block 14: OpenAPI specification for List inventory products

```yaml
paths:
/inventory/products:
    get:
    tags:
        - Inventory
    summary: List all products in the inventory
    operationId: listInventoryProducts
    parameters:
        - in: query
        name: ids
        schema:
            type: array
            items:
            type: string
        required: false
        description: Filter by product IDs
      ...
      responses:
      '200':
        description: Inventory products retrieved successfully
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/Product'
      ...
components:
  schemas:
    Product: # OpenAPI specification for Product.
        required:
        - name
        type: object
        properties:
        productID:
            type: string
        name:
            type: string
        price:
            $ref: '#/components/schemas/Money'
```

Code block 15: Code generated from OpenAPI specification

```
type StrictServerInterface interface {
  ListInventoryProducts(
        ctx context.Context,
        request ListInventoryProductsRequestObject
    ) (ListInventoryProductsResponseObject, error)
}


type Product struct {
  Name      string  `json:"name"`
  Price     *Money  `json:"price,omitempty"`
  ProductID *string `json:"productID,omitempty"`
}

```

Exercise 15

Try adding TranslatedName string field to Product and re-generate Go code with `make oapi`.

10.1.2 Handler implementation

Handlers are generated from the OpenAPI specification but need to be integrated into the code manually.

A strict server interface, which is automatically generated from the OpenAPI spec, defines a set of contractually required methods that the server must implement to handle incoming HTTP requests. This interface is typically enforced at the HTTP framework level, ensuring that the application will not compile unless all required handler functions are properly implemented. This approach is beneficial because it guarantees that the entire infrastructure layer related to HTTP adheres to specific types. Typed endpoints and HTTP protocol-specific data structures provide strong type guarantees, reducing the chances of runtime errors.

`ListInventoryProducts` handler is implemented in block code example 16.

Code block 16: ListInventoryProducts handler implementation

```
func (h *Handler) ListInventoryProducts(
    _ context.Context,
    request httpgen.ListInventoryProductsRequestObject,
) (httpgen.ListInventoryProductsResponseObject, error) {
    filter, err := ParseFilterParams(request)
    if errors.As(err, &FilterMinHigher{}) {
        return &httpgen.ListInventoryProducts400JSONResponse{
            BadRequestJSONResponse: httpgen.BadRequestJSONResponse{
                Message: pointer.ToPtr("Min value must be lower than max value"),
            },
        }, nil
    }
    if err != nil {
        return nil, err
    }

    products, err := h.InventoryService.ListProducts(*filter)
    if err != nil {
        return nil, err
    }
    return httpgen.ListInventoryProducts200JSONResponse(
        dtos.NewListProducts(products).Products,
    ), nil
}
```

Where `httpgen.ListInventoryProducts200JSONResponse` is a data container to be directly transformed to json. `200` in the name signals that this response is for `200` response previously defined in OpenAPI spec. Error handling will be covered in section 12.1. HTTP JSON response can be inspected in block code example 17.

Code block 17: HTTP Response for ListProducts

```
[
    {
        "productID": "product_01j332dx0eeb69vexx4y7pn0td",
        "name": "Generic basil sauce",
        "price": {
            "amount": 3.7,
            "currency": "EUR"
        }
    }
]

```

`Handler` acts as container with domain services which can be called and it must implement all auto-generated interfaces. Handler will have receiver methods

as implementations, while container itself can look like this:

Code block 18: HTTP handlers container

```
type Handler struct {
  InventoryService service.InventoryService
  CartService      service.CartService
  OrderService     service.OrderService
}
```

Exercise 16

Implement previously added `TranslatedName` field by returning capitalized name. Run application and confirm it works.

10.1.3 Transforming domain layer structures to infrastructure layer

Generated data structured with purpose to interact with HTTP need to be connected to domain. This means that transformation $entity.Product \rightarrow httpgen.Product$ needs to be implemented, visible in figure 10.2.

Figure 10.2: Crossing boundary means type transformation is needed

Code in the last paragraphs used `dtos` package to transform domain types to HTTP ones. Code snippet below shows how domain entity is transformed to DTO understandable by HTTP protocol. This additional transformation step and decoupling allows for these two types to evolve at different paces [1], visible in figure 10.3.

Code block 19: List Products DTO for HTTP

```
package dtos

type ListProductsResponse struct {
  Products []httpgen.Product
}

func NewListProducts(products []product.Product) ListProductsResponse {
    httpProducts := make([]httpgen.Product, len(products))
  for i, currProduct := range products {
    httpProducts[i] = newProduct(currProduct)
  }

  return ListProductsResponse{
    Products: httpProducts,
  }
}

func newProduct(product product.Product) httpgen.Product {
  return httpgen.Product{
    ProductID: pointer.ToPtr(product.ID.String()),
    Name:      product.Name,
    Price:     pointer.ToPtr(ToMoney(product.Price)),
  }
}

```

In figure 10.3, `TranslatedName` might not have any domain meaning and is just used as a helper for UI to render products correctly. While parameter `InStock` might be hidden from user and only used for different purposes like ordering and etc.

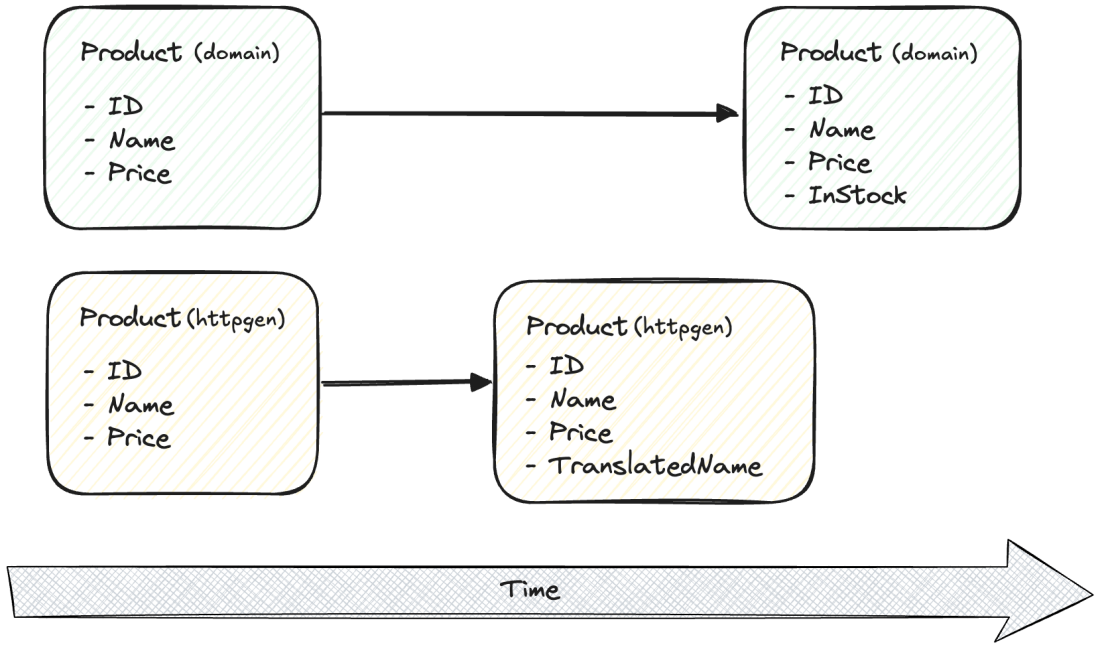


Figure 10.3: Domain split allows for types to evolve at different pace

10.1.4 HTTP type conversions to domain layer

Transformations are needed to opposite direction. For types residing in generated HTTP request types to be connected to the rest of application. Specifically things like parameters and request bodies. They need to be validated and converted to language understandable by domain layer. More about invariants in chapter 7.

figure 10.4 shows how HTTP layer type is transformed to `product.Upload` domain type. This `httpgen.UploadProduct` allows separating `Product` creation login from already created `Product`. This separation also allows including various validation parameters and logic specifically for `product.Upload`.

Code block 20: Upload Product HTTP model

```
1  package httpgen
2
3  type UploadProduct struct {
4      Name  string `json:"name"`
5      Price Money  `json:"price"`
6  }
```

Code block 21: Upload Product domain model

```
package product

type Upload struct {
  Name  string
  Price money.Money
}
```

Code block 22: upload inventory product handler

```
package http

func (h *Handler) UploadInventoryProduct(
  _ context.Context,
  request httpgen.UploadInventoryProductRequestObject,
) (httpgen.UploadInventoryProductResponseObject, error) {
  price, err := decoders.Money(request.Body.Price)
  if err != nil {
    return nil, err
  }

  uploadProduct := product.NewUpload(
    request.Body.Name,
    price,
  )

  _, err = h.InventoryService.UploadProduct(uploadProduct)
  if err != nil {
    return nil, err
  }

  return nil, nil
}
```

After `product.Upload` domain struct is built – it can be passed to inventory service's `UploadProduct` method.

Figure 10.4: Opposite direction transformation to cross boundary

Exercise 17

Add optional parameter to `UploadProduct` domain function named `Hidden` to indicate if product should be shown. However, in HTTP keep `UploadProduct` the same.

10.2 Repositories

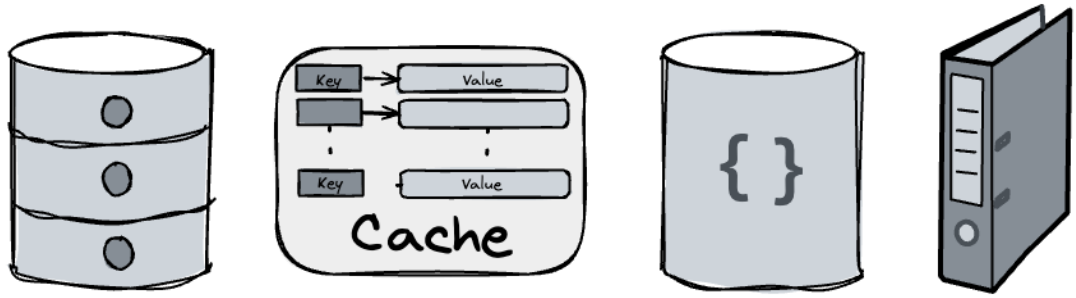


Figure 10.5: Data storage in various repositories

Data storage is essential part for most software. From caching to cold storage – a lot of logic relies on data being persisted and read. In this Online Store project, domain logic is separated from repositories.

Its not the only pattern to deal with this problem. For example Ruby on Rails' ActiveRecord has quite a concise API. While ActiveRecord leaks to all application layers, Rails makes it work. That is if you can accept the constraints and trade-offs. This approach can work well in monoliths, but might fall short in distributed context [2]. More about interaction with non-local data in section 10.3.

Repositories are a useful pattern to enable DDD. From domain perspective, it abstracts storage. With certain cases, domain logic could work with pen and paper as a repository. This cuts a lot of unnescessary code from domain. Repository code also tends to evolve at different pace than that of a domain.

There are downsides of using repositories as well. First it requires more code to add them and to change later on. Second, effort is needed to split domain and repository into two layers. Both code and tests. Also, to test domain code, some kind of repository implementation is needed, because working with mocks is not practical. More about in chapter 11.

10.2.1 Inferfaces

Repositories can be implemented with various storage options. With Shopping Cart use case, primary choice for persistance will be SQLite.

Starting with repository interface defined in domain. In code example 23, `CartRepository` interface defines various methods for retrieving,

[2] In distributed contexts, ActiveRecord is no longer the only source of data. Due it leaking to all application layers, it becomes quite bothersome to interact with non-local data

updating or creating data. This generally serves as easy to understand place on what is the scope and capabilities of this repository. Worth noting that some shopping cart repository capabilities are more generic and others are very specific for some domain case.

Let's take a look at previously mentioned example: `GetWithItems` is used retrieve cart with items quite commonly. It accepts a filter which has some generic parameters and can be used in various places in a service. More about filters in chapter 12.

As mentioned before, some capabilities like `InitiateCheckout` are quite specific for one case. In practice there is usually no point to try and fully generalize all repository functions. Its fine to have some duplication, specific single uses and do refactoring once it becomes a problem.

Code block 23: Cart repository interface in domain

```
1  package repositories
2
3  type CartRepository interface {
4    GetCartByUserId(userID model.UserID) (shoppingcart.Cart, bool, error)
5    GetCart(cartID shoppingcart.ID) (shoppingcart.Cart, bool, error)
6    GetWithItems(filter shoppingcart.Filter) (shoppingcart.WithItems, bool, error)
7    CreateCart(cart shoppingcart.Cart) error
8    AddItems(items []shoppingcart.Item) error
9    InitiateCheckout(
10     id shoppingcart.ID,
11     initiatedAt time.Time,
12     checkoutAmount money.Money,
13   ) error
14   CompleteCheckout(
15     id shoppingcart.ID,
16     completedAt *time.Time,
17     failedAt *time.Time,
18   ) error
19 }
20
```

Warning

Be aware that defining interfaces in this way goes against standard Go practices of returning concrete types and not defining interfaces on implementation side [a].
Having this repository interface is opinionated take. We want tight coupling here and original reasoning to avoid this does not apply here.

[a] https://go.dev/wiki/CodeReviewComments#interfaces

10.2.2 Mapping aggregates to tables

Let's take a look at more complicated example of fetching shopping cart with items in code example 25. As previously discussed, this repository function needs to return data from multiple tables. Specifically it needs to return shopping cart information, what items are in shopping cart and what exactly are those items. In other words: to display this `shoppingcart.WithItems` aggregate – we need to fetch from three database tables: `carts`, `cart_items`, `products`.

Code block 24: Structs to represent models in repository context

```
1  // Autogenerated from database
2  // jetmodel.CartItems
3  type CartItems struct {
4    ID        string `sql:"primary_key"`
5    ProductID string
6    CartID    string
7  }
8
9  type CartItemWithProduct struct {
10   jetmodel.CartItems
11   jetmodel.Products
12 }
```

TODO: figure to show tables

1. Setup dependencies:

Code block 25: GetWithItems implementation

```
func (r *CartRepositoryDB) GetWithItems(
  filter shoppingcart.Filter,
) (shoppingcart.WithItems, bool, error) {
  ...
  conditions := make([]BoolExpression, 0)
  if filter.UserID != nil {
    conditions = append(
      conditions,
      Carts.UserID.EQ(String(string(*filter.UserID))),
    )
  }
  ...
  var carts []jetmodel.Carts
  if err := stmt.Query(r.db, &carts); err != nil {
    return shoppingcart.WithItems{},
      false,
      fmt.Errorf("failed to query cart: %w", err)
  }
  ...
  qItems := CartItems.
    SELECT(CartItems.AllColumns, Products.AllColumns).
    FROM(CartItems.LEFT_JOIN(Products, Products.ID.EQ(CartItems.ProductID))).
    WHERE(CartItems.CartID.EQ(String(cart.ID.String())))

  var itemsCombined []CartItemWithProduct
  err = qItems.Query(r.db, &itemsCombined)
  if err != nil {
    return shoppingcart.WithItems{},
      false,
      fmt.Errorf("failed to query cart items: %w", err)
  }

  var domainItems []shoppingcart.Item
  for _, combined := range itemsCombined {
    ...
    prod := product.NewProduct(
      pID,
      combined.Products.Name,
      moneyPrice,
    )
    ...
    item := shoppingcart.NewItem(
      cItemID,
      prod,
      cart.ID,
    )
    domainItems = append(domainItems, item)
  }

  return shoppingcart.WithItems{
    Cart:  cart,
    Items: domainItems,
  }, true, nil
}
```

10.2.3 Transactions

Domain model and domain layer is usually incorrect place to manage database transactions [5]. In this case, adding multiple items to a cart means that this has to happen in transaction. `AddItems` function in code example 26 runs these queries inside transaction. No logic concerned with database transactions is leaked to domain layer.

Code block 26: AddItems SQLite repository implementation

```
1  func (r *CartRepositoryDB) AddItems(items []shoppingcart.CItem) error {
2      return r.db.Transaction(func(tx *gorm.DB) error {
3          rowItems := make([]CartItemRow, len(items))
4
5          for i, currItem := range items {
6              rowItems[i] = NewCartItemRow(
7                  currItem.ID, currItem.Item.ID, currItem.CartID,
8              )
9          }
10
11         return tx.Create(&rowItems).Error
12     })
13 }
```

In real world there will be cases when transactions want to escape nicely contained repositories. For most cases modelling data and boundaries differently could help, but its not always the case and usually there is no time for that. To handle these exceptions, special methods can be exposed to public repository APIs or new repositories created just for that purpose.

TODO! more examples

10.3 External and internal clients

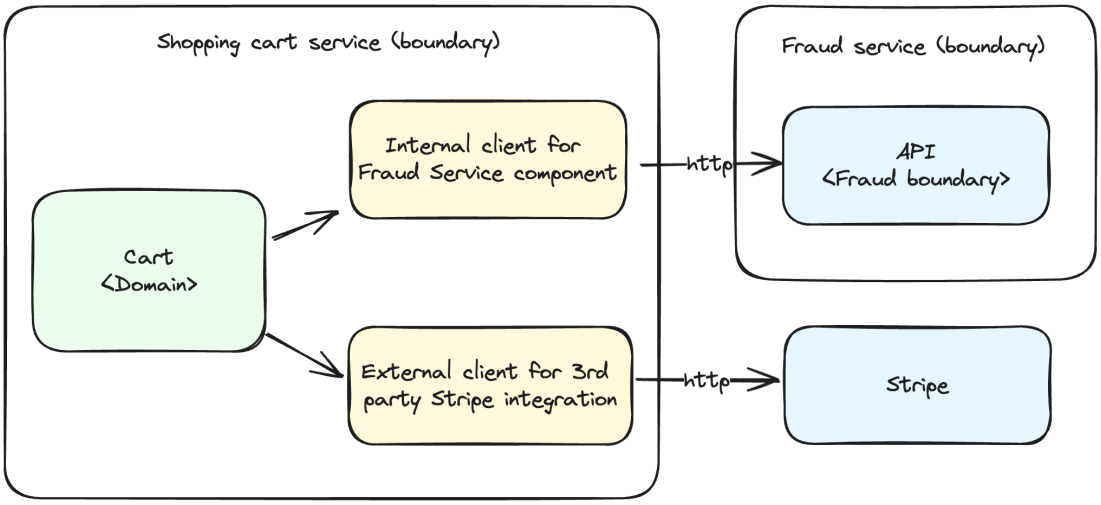


Figure 10.6: Interacting with other components over HTTP

While repositories handle the persistence of data within the current boundary, a significant part of the codebase also needs to interact with other boundaries. These boundaries can range from third-party APIs to internal authorization services, necessitating an efficient method of communication.

APIs function like remote repositories, albeit with the added constraints of network communication. We can use the concept of clients—specialized components designed to interact with other APIs. These clients can be categorized into two types: internal and external. This can be seen in figure 10.6.

External clients manage communication with third-party APIs. *Internal clients* manage communication with internal services[3]. These clients initiate actions or request information from other boundaries.

In a service-oriented architecture, storage systems like databases are generally isolated to the service they belong to and accessed through repositories.

The client pattern allows communication with other services in a synchronous way. For more information on asynchronous communication, see chapter 14.

[3]The term "service" is overloaded here. The domain service pattern is used to define business logic, while "service" in this context refers to a component of a software system. Typically, these components are developed and deployed separately, often representing different boundaries.

Important

Directly accessing another service's internal data through non-API ways is considered an anti-pattern.

10.4 Shared layer

Shared layer's purpose is to provide code which can be used in any other layer. It is essentially a collection of internal libraries.

All shared code is not the same. Some abstractions will die as they were defined incorrectly and some of them will live on. Lifecycle can be imagined as: from codebase code is extracted to internal/pkg. That's where it escapes DDD tags and just becomes a library. After that next step is to become public so other packages can use it if needed. Last step it is extracted into full library with its own development lifecycle. This timeline is visible in figure 10.7

As time passes, shared code becomes less likely to change. That's why this migration to library is good. A package which is a library itself can be used by others projects, enabling productivity.

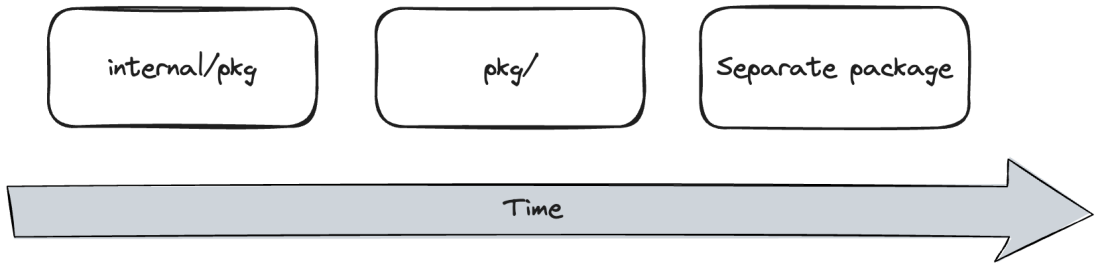


Figure 10.7: Shared code location migration over time

10.5 Summary

Layers allow domain code to be kept isolated, easy to understand and change. Layers also allow separating code in terms of code evolution over time.

Business domain layer contains all business related data structures and logic. Infrastructure layer will depend on business layer.

Infrastructure layer will use endpoints or event consumers to invoke business logic. It will act as a controller and call specific business use case. Infra layer will also implement repository and client interfaces. Implementation will contain code which queries data from database or calls other services or 3rd party APIs.

Application layer will act as glue to tie all application together. From dependency injection to starting HTTP servers.

Shared layer will contain all internal and external library like functions. It will be used by all layers and will change less frequently.

Chapter 11

(work-in-progress) Testing: Overview and Examples

Testing is a crucial part of software development. It provides fast feedback, builds confidence in the system, and helps ensure that software behaves as expected. Without effective testing, even small issues can cascade into significant problems in production, making testing essential.

In the context of Domain-Driven Design, testing becomes simpler and more direct due to the isolation of domain code. By design, DDD emphasizes clear boundaries and explicit models, resulting in code that is naturally easier to test. As reviewed before, domain code is focused on business logic, free from infrastructure concerns, and operates within a well-defined boundary.

However, testing business logic often requires interaction with infrastructure components such as repositories and external clients. These dependencies introduce challenges.

This chapter advocates for a practical and effective solution: *testing with actual implementations*. This approach involves using the real repository implementations and interacting directly with the actual database during tests. By doing so, we simplify the testing process and ensure tests closely reflect real-world usage. Let's see why this approach is effective and how it compares to alternative strategies.

11.1 Testing with actual implementations

Testing with actual implementations allows your tests to closely replicate production conditions. Instead of relying on artificial stand-ins, these tests use real repositories and interact with an actual database. As a result, they capture nuances such as constraints, indexing, and transactional behavior, greatly reducing

the risk of false positives. The process can be surprisingly direct: simply set up the necessary data, run the domain logic, and verify the outcomes without managing a host of mock objects or in-memory data stores.

This method also keeps your test environment closer to the real deployment conditions. Since you are using the same code and infrastructure as in production, there is less overhead in maintaining separate testing constructs. Fewer moving parts mean fewer places for your tests and production code to drift apart over time.

However, this approach brings its own challenges. Because it involves real databases, tests will naturally run slower than their in-memory counterparts or mocks. The added complexity of managing database setup, teardown, and environmental dependencies can also introduce flakiness. Additionally, using actual implementations blurs traditional testing boundaries, making it harder to distinguish between what should be a unit test versus an integration test. In DDD context, these tests also ignore layer dependency rules. Domain tests will include dependencies from infrastructure layer, like repositories.

Below is a summary of the benefits and trade-offs:

- **Simplified Test Setup**: Eliminating mocks and in-memory makes testing straightforward and easy to work with.
- **Realistic Scenarios**: Direct database interactions reveal issues related to database specifics, and transactional integrity—problems that alternative methods might never uncover.
- **Reduced Maintenance**: Relying on actual code and infrastructure prevents duplication and ensures that tests stay aligned with ongoing changes. Additionally, there is no need to write code which is only used for tests.

At the same time:

- **Slower Execution**: Real database operations are more slow. Test run times increase.
- **Potential Flakiness**: Managing and isolating a database can lead to issues which result in fragile test runs.
- **Blurred Boundaries**: Writing tests which interact with database can make it unclear where unit testing ends and integration testing begins. In DDD context, these tests also ignore layer boundaries and one-direction dependency restrictions.

Alternative Strategies

There are other testing strategies that can be considered. Specifically, but not limited to:

- **Mocking Dependencies**: Each layer can be mocked. Instead of calling repositories and executing queries – only check if calls to repository were made.
- **In-Memory Repositories**: In-memory repositories simulate repository behavior without interacting with an actual database. This approach speeds up testing by avoiding the overhead of database setup and operations. Biggest downside is implementing in-memory repositories just for tests.

11.2 Testing library: goconvey

This project will use **goconvey** [1] as its primary testing library. GoConvey is designed to support behavior-driven development (BDD), making it an excellent fit for business logic. Its expressive syntax allows for writing clear and concise tests that mirror the domain language, promoting readability and maintainability.

While the default `testing` package is robust and widely used, it lacks the expressiveness and convenience provided by goconvey.

[1] goconvey – https://smartystreets.github.io/goconvey/

11.3 Domain tests: testing inventory service

Code block 27: Inventory service tests

```
func TestInventoryServiceImpl(t *testing.T) {
  convey.Convey("Inventory service", t, func() {
    productRepo := repository.NewProductRepositoryDB(dbtest.Db)
    svc := service.NewInventoryService(productRepo)

    convey.Convey("when creating a new item", func() {
      price, _ := money.NewMoney(100)
      createItem := product.Create{
        Name:  "Test Product",
        Price: price,
      }

      createdItem, err := svc.CreateProduct(createItem)

      convey.Convey("when created successfully", func() {
        convey.So(err, convey.ShouldBeNil)
        convey.So(createdItem.Name, convey.ShouldEqual, createItem.Name)
        convey.So(createdItem.Price, convey.ShouldResemble, createItem.Price)
        convey.So(createdItem.ID, convey.ShouldNotBeEmpty)

        convey.Convey("it can be listed", func() {
          items, err := svc.ListProducts(
            product.Filter{IDs: &[]product.ID{createdItem.ID}},
          )
          convey.So(err, convey.ShouldBeNil)
          convey.So(len(items), convey.ShouldEqual, 1)
          convey.So(items[0], convey.ShouldResemble, createdItem)
        })
      })

      convey.Convey("when item price is negative", func() {
        createItem.Price, _ = money.NewMoney(-100)
        _, err := svc.CreateProduct(createItem)
        convey.So(err, convey.ShouldNotBeNil)
        convey.So(
          err.Error(),
          convey.ShouldEqual,
          "Price must be positive",
        )
      })
    })
  })
}
```

Code in code example 27 does the following:

- Setup dependencies: Initialize required dependencies: repositories and services. An instance of the repository `productRepo` is created using a real database connection, `dbtest.Db`. The inventory service `service` is then created, using the repository.
- Create a new `Product`: Initialize a `product.Create` struct with the product's name and price, call `service.CreateItem()` to create the item in the repository as well.
- List the created `Product`: List the created product. Call `service.ListProducts()` using a filter with the newly created product's ID. Assert that the correct product is returned and that it matches the product that was previously created.
- Test invariants: Another `Convey` block is added to test for invalid input, specifically a negative price. The price of `createItem` is set to a negative value, and `service.CreateProduct()` is called again. The test verifies that an error is returned, with the correct error message indicating that the price must be positive. This tests domain layer logic for negative price handling.

The use of nested `Convey` blocks makes the test cases more readable, reflecting a clear sequence of actions and expectations. Also, by interacting directly with the real repository and database, the tests are easy to read and comprehend.

11.4 Repository tests: testing `Product` repository

Code block 28: Create products for tests

```
1    convey.Convey("ProductRepository", t, func() {
2      ...
3      convey.Convey("ListProducts", func() {
4        convey.Convey("when no products exist", func() {
5          products, err := productRepo.ListProducts(product.Filter{})
6          convey.So(err, convey.ShouldBeNil)
7          convey.So(len(products), convey.ShouldEqual, 0)
8        })
9
10       convey.Convey("with products", func() {
11         productID, _ := typeid.New[product.ID]()
12         productID2, _ := typeid.New[product.ID]()
13         productID3, _ := typeid.New[product.ID]()
14
15         err := productRepo.CreateProduct(product.Product{
16           ID:    productID,
17           Name:  "Test Product 1",
18           Price: money.MustNewMoney(100),
19         })
20         convey.So(err, convey.ShouldBeNil)
21
22         err = productRepo.CreateProduct(product.Product{
23           ID:    productID2,
24           Name:  "Test Product 2",
25           Price: money.MustNewMoney(200),
26         })
27         convey.So(err, convey.ShouldBeNil)
28
29         err = productRepo.CreateProduct(product.Product{
30           ID:    productID3,
31           Name:  "Special Product 3",
32           Price: money.MustNewMoney(300),
33         })
34         convey.So(err, convey.ShouldBeNil)
35         ...
```

Step-by-step for code example 28:

- Setup dependencies: Initialize the `productRepo` repository instance with a real database connection (`dbtest.Db`). Perform cleanup to ensure a fresh, consistent state before each test.

- Create multiple `Products`: Create three distinct products in the repository and verify that each `CreateProduct()` call succeeds.
- List products without filters: Call `productRepo.ListProducts()` with no filters to confirm all created products are returned as expected.
- Filter products by `ID`: Use `IDs` filter to ensure only the requested products are returned.
- Filter products by price range: Use a `PriceRangeFilter` to retrieve products within a specified minimum and maximum price. Verify that filtering logic handles domain constraints correctly.

These repository tests interact directly with the database-backed storage layer. They confirm that products are consistently created, retrieved, and filtered on relevant scenarios.

The second part involves testing typical repository operations. The tests include listing all products without filters to confirm that all items are returned, applying an ID filter to ensure only specific items are fetched, and using a price range filter to validate that items within the specified range are correctly retrieved. These tests collectively ensure that the repository layer handles common operations and filtering logic effectively, interacting directly with the database.

Code block 29: Product repository tests

```
convey.Convey("should return all products with no filter", func() {
  products, err := productRepo.ListProducts(product.Filter{})
  convey.So(err, convey.ShouldBeNil)
  convey.So(len(products), convey.ShouldEqual, 3)
})

convey.Convey("should return products filtered by ID", func() {
  filter := product.Filter{
    IDs: &[]product.ID{productID, productID3},
  }
  products, err := productRepo.ListProducts(filter)
  convey.So(err, convey.ShouldBeNil)
  convey.So(len(products), convey.ShouldEqual, 2)
  convey.So(products[0].ID, convey.ShouldBeIn, []product.ID{productID, productID3})
  convey.So(products[1].ID, convey.ShouldBeIn, []product.ID{productID, productID3})
})

convey.Convey("should return products filtered by price range", func() {
  minPrice := money.MustNewMoney(150)
  maxPrice := money.MustNewMoney(250)
  filter := product.Filter{
    PriceRangeFilter: &product.PriceRangeFilter{
      Min: &minPrice,
      Max: &maxPrice,
    },
  }
  products, err := productRepo.ListProducts(filter)
  convey.So(err, convey.ShouldBeNil)
  convey.So(len(products), convey.ShouldEqual, 1)
  convey.So(products[0].Price.Amount, convey.ShouldEqual, 200)
})
```

11.5 Tooling: Taming complexity due bi-directional layer dependencies in tests

Testing with real implementations means that previously established one-directional dependencies are ignored. This is a real issue and can bring a lot of complexity in the codebase. To address this, good tooling is needed.

TODO!

It can be setup to enforce arrows discussed seen in figure 5.1.

Chapter 12

(work-in-progress) Between layers: Errors, Filters, Validations

12.1 Error handling

12.2 Filtering

In Patterns of Enterprise Application Architecture [2], Martin Fowler talks about *Query object*. *Query object* is an interpreter that can transform itself to SQL query in repository implementation. This allows decoupling specific repository implementations from business domain and is useful in DDD context.

Say a list of items needs to be filtered by few parameters. Filter itself can be declared in domain. Doing this makes filter easy to understand and use in business logic.

Code block 30: Filter

```
type Filter struct {
    IDs              *[]ID
    NameWildcard     *string
    PriceRangeFilter *PriceRangeFilter
}

type PriceRangeFilter struct {
    Min *utils.Money
    Max *utils.Money
}
```

12.3 Ordering

12.4 Validations

12.4.1 Input validation

12.4.2 Model validation

Chapter 13

(work-in-progress) Bounded contexts and context mapping

13.1 Introduction

DDD provides methods to effectively manage complexity. Primarily by structuring software around clearly defined business domains. This approach is built around the concept of the *bounded context* [1], a strategic[1] principle that helps teams clearly define and encapsulate meaning within the software system.

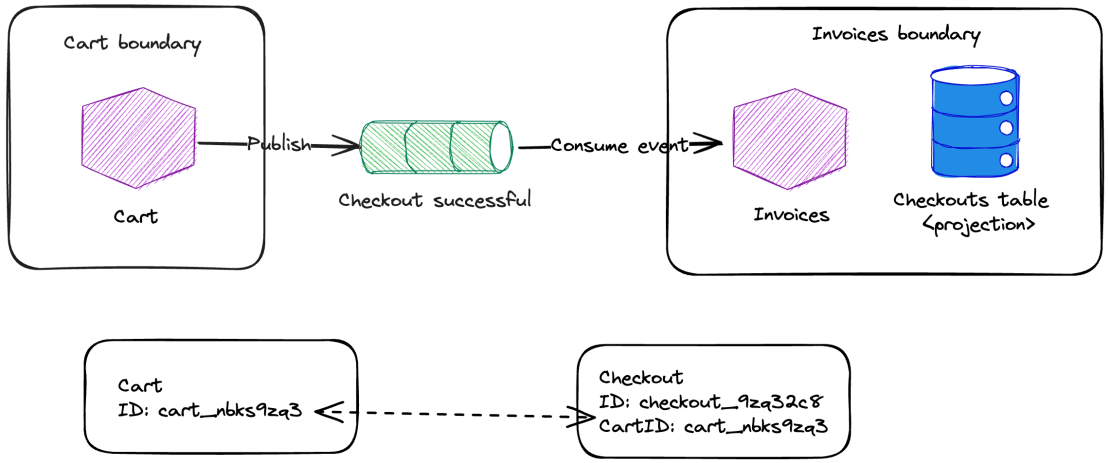


Figure 13.1: Shopping cart and invoice boundaries

[1]In DDD, *strategic* refers to decisions and practices that align software architecture closely with business goals, helping organizations effectively manage complexity and guide high-level decision-making.

13.2 Bounded context

A bounded context is an explicit boundary within which a particular domain model has consistent and clearly defined meanings. Within a bounded context, ubiquitous language, rules, and logic are clearly specified and understood uniformly. This isolation ensures consistency, reduces ambiguity, and improves communication clarity.

Let's take a look at diagram seen in chapter 9, visible in figure figure 13.1. In *Cart* boundary cart record has various states and a specific meaning to capture shopping cart. However, in invoices boundary *Cart* is treated as a *Checkout*. This is because in invoices boundary only successfully completed shopping carts exist. They have a different meaning that exists in this specific boundary.

13.3 Context map

13.4 Identifying bounded contexts

Chapter 14

(empty) Domain Events and Services

Chapter 15

Durable execution

Durable execution is a design approach that ensures processes can continue running or resume from a saved point after an interruption or error. Rather than re-consuming messages or requiring clients to repeat failed requests, system relies on saved state at regular checkpoints. In the event of a failure, the process can pick up precisely where it left off, ensuring that the overall workflow is eventually executed despite intermittent issues.

15.1 Motivation and overview

Today, almost all systems are distributed, which means failures are always a possibility. These failures can come from transient network issues, hardware problems, or unexpected runtime errors. When failures occur, systems often end up in a partially failed state that must be restored. Addressing this adds significant complexity to system itself.

Durable execution addresses these challenges by saving the state of long-running processes. This approach simplifies error handling and improves the overall reliability and resilience of the system. The remaining challenge is integrating it seamlessly to the codebase.

Why bother? Key points:

- **Resilience and consistency:** Every workflow will eventually complete. Even if a failure occurs mid-process, the system automatically manages retries and resumes from the last saved state, so no process is left in an incomplete state.
- **Simplicity:** The complexity of state persistence and error recovery is handled by the durable execution mechanism. This abstraction allows to focus

on domain business logic without having to architect custom solutions for fault tolerance.

15.2 Temporal in Go

Temporal is an open-source workflow orchestration framework that manages durable workflows. It is a popular choice within the Go community for handling distributed state and long-running processes. While other alternatives exist, this book focuses on Temporal.

In Temporal, workflows orchestrate and manage long-running business processes with stateful logic, while activities are stateless, discrete tasks that execute specific units of work.

For more information on how Temporal works at a high level, you can refer to the official documentation [1]

15.3 Payments integration workflow

Before diving into Temporal usage examples in online store, let's review the payment integration flow:

- A customer initiates checkout on the online store. Shopping cart is in initiated state, Stripe secret is returned to frontend.
- Stripe frontend integration helps to capture payment details. Together with backend secret, this data is sent it to Stripe for further processing. Stripe processes the payment and sends a webhook event containing the payment status back to online store backend. Visible in figure 15.1.
- Upon receiving and verifying the webhook, code example 31, the application triggers the `CartService.CompleteCheckout` function.
- This function fetches the current state of the shopping cart from persistent storage to confirm that the checkout process has been initiated.
- A Temporal workflow is then launched to orchestrate next actions. Depending on the payment status, the workflow executes the appropriate activities. `CheckoutSuccess` updates shopping cart state and invokes `CreateOrder` domain functions. This ensure that these actions are consistent, and resilient to transient failures. `CheckoutFailed` persists shopping cart state as failed and exits.

[1] `https://docs.temporal.io/evaluate/understanding-temporal`

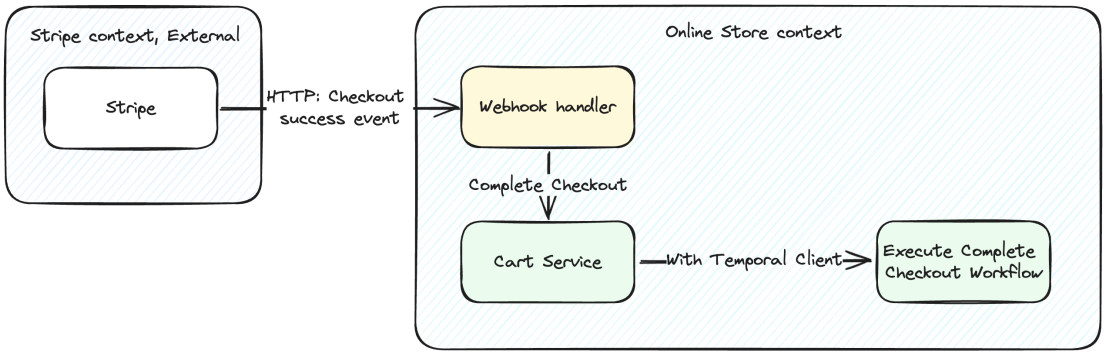


Figure 15.1: Stripe hook → Online Store → Complete checkout

Handling Stripe request

Let's take a look at how shopping cart checkout is completed. A webhook is defined to handle Stripe responses. When Stripe processes a payment event, it sends an HTTP POST request (a webhook) to http server containing the event details. HTTP handler verifies the webhook signature for authenticity, extracts the payment status, and triggers the checkout process.

Code block 31: Payments webhook HTTP handler

```
func (h *Handler) HandlePaymentWebhook(
  w http.ResponseWriter,
  r *http.Request,
) {
  //...
  cartIDRaw := paymentIntent.Metadata["cart_id"]
  //...
    case "payment_intent.succeeded":
    err = h.handlePaymentSucceeded(r.Context(), cartID)
  //...
}
```

15.4 (work-in-progress) Integrating into DDD codebase

Integrating Temporal into a DDD codebase is challenging. Temporal's approach to workflow orchestration and state management doesn't neatly align with the typical "clean" separation of domain. Parts of the workflow code must be stored and executed in the Temporal worker, forcing serialization and external state tracking.

Temporal introduces new abstractions—specifically *Activity* and *Workflow*. It

might be tempting to assign activities to the infrastructure layer and workflows to the domain layer, but this kind of separation does not work. This will limit how Temporal can be used and create unnecessary confusion. There's no rule preventing us from having workflows solely in the infrastructure layer when needed, or have activities in domain only.

Given this, we need to treat these Temporal abstractions as flexible elements that can appear in any layer. In essence, we should think of them as a Domain-specific language. We have two options:

1. Create additional abstractions to clearly separate Temporal concerns between the domain and infrastructure.

2. Accept a certain degree of infrastructure-specific DSL within our domain code.

Latter approach should bring less confusion and keep things simpler.

CartService, Temporal and layers

!TODO DDD layer explanation !TODO Review partial failed states

See figure 15.2.

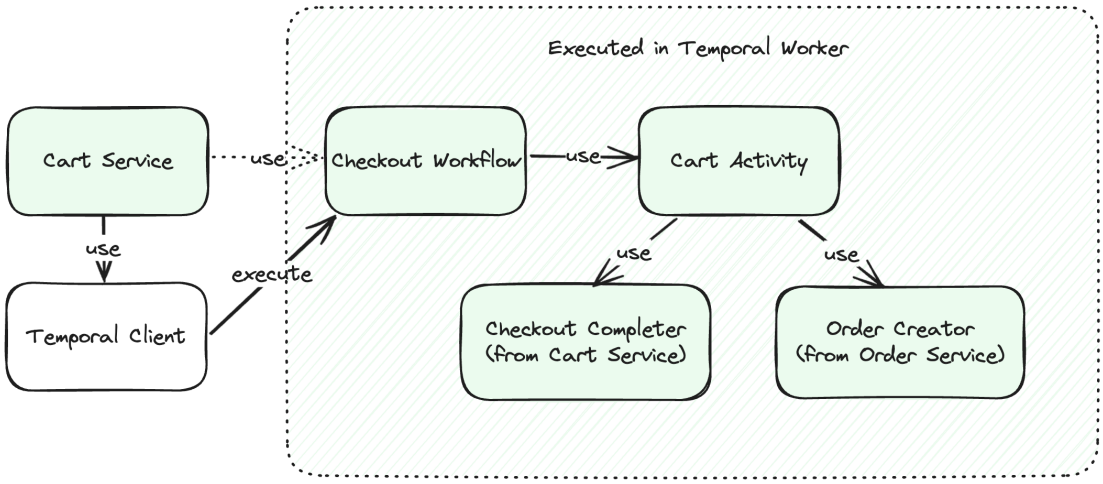


Figure 15.2: Temporal: DDD

Chapter 16

(empty) Bonus chapters

16.1 Transaction script

16.2 In-Memory repositories for testing

Testing with real repositories can be too slow due persistent implementations. Or there might be other constraints which prevent from referencing infrastructure layer from domain tests. Whatever the reason, there are alternatives. One of them being in-memory repository implementation [5].

TODO

Glossary

ActiveRecord An Object-Relational Mapping (ORM) framework used in Ruby on Rails that simplifies database interactions by mapping database tables to classes and rows to objects, allowing developers to perform database operations using object-oriented paradigms without writing raw SQL.. 55

OpenAPI The OpenAPI Specification defines a standard, language-agnostic interface to RESTful APIs, allowing both humans and computers to discover and understand the capabilities of a service without access to source code.. 44, 48

SQLite SQLite is a self-contained, serverless, zero-configuration, transactional SQL database engine.. 55

Stripe Stripe is a financial services and software company, providing payment processing for online businesses.. 82, 83

Temporal An open-source platform for running durable, long-running workflows. Temporal allows to build fault-tolerant and stateful applications by handling retries, state persistence, and recovery from failures automatically. https://temporal.io. 82

Acronyms

API Application Programming Interface. 43

DDD Domain-Driven Design. 6, 8–11, 19, 21, 36, 39, 42, 55, 62, 68, 69, 77

DSL Domain-specific language. 84

DTO Data Transfer Object, an object that carries data between processes. 50

HTTP HyperText Transfer Protocol. 2, 45, 47, 48, 50, 52

JSON JavaScript Object Notation, a lightweight data-interchange format based on a subset of JavaScript.. 48

UUID Universally Unique Identifier. 39, 40, 42

Bibliography

[1] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley, 2003. ISBN: 978-0-321-12521-7. URL: https://www.safaribooksonline.com/library/view/domain-driven-design-tackling/0321125215/.

[2] Martin Fowler. *Patterns of Enterprise Application Architecture*. Erste Auflage. Boston: Addison-Wesley, 2003.

[3] Ted Neward. *The Fallacies of Enterprise Computing — blogs.newardassociates.com*. https://blogs.newardassociates.com/blog/2016/enterprise-computing-fallacies.html. [Accessed 25-01-2025].

[4] Barry M O'Reilly. *Residues: Time, Change, and Uncertainty in Software Architecture*. Leanpub.

[5] Vaughn Vernon. *Implementing Domain-Driven Design*. 1st. Addison-Wesley Professional, 2013. ISBN: 0321834577.