ASP.NET Core Clean Architecture, DDD, and Modular Monoliths

Steve Smith

@ardalis

steve@nimblepros.com | NimblePros.com

NimblePros

Better software development. Delivered.

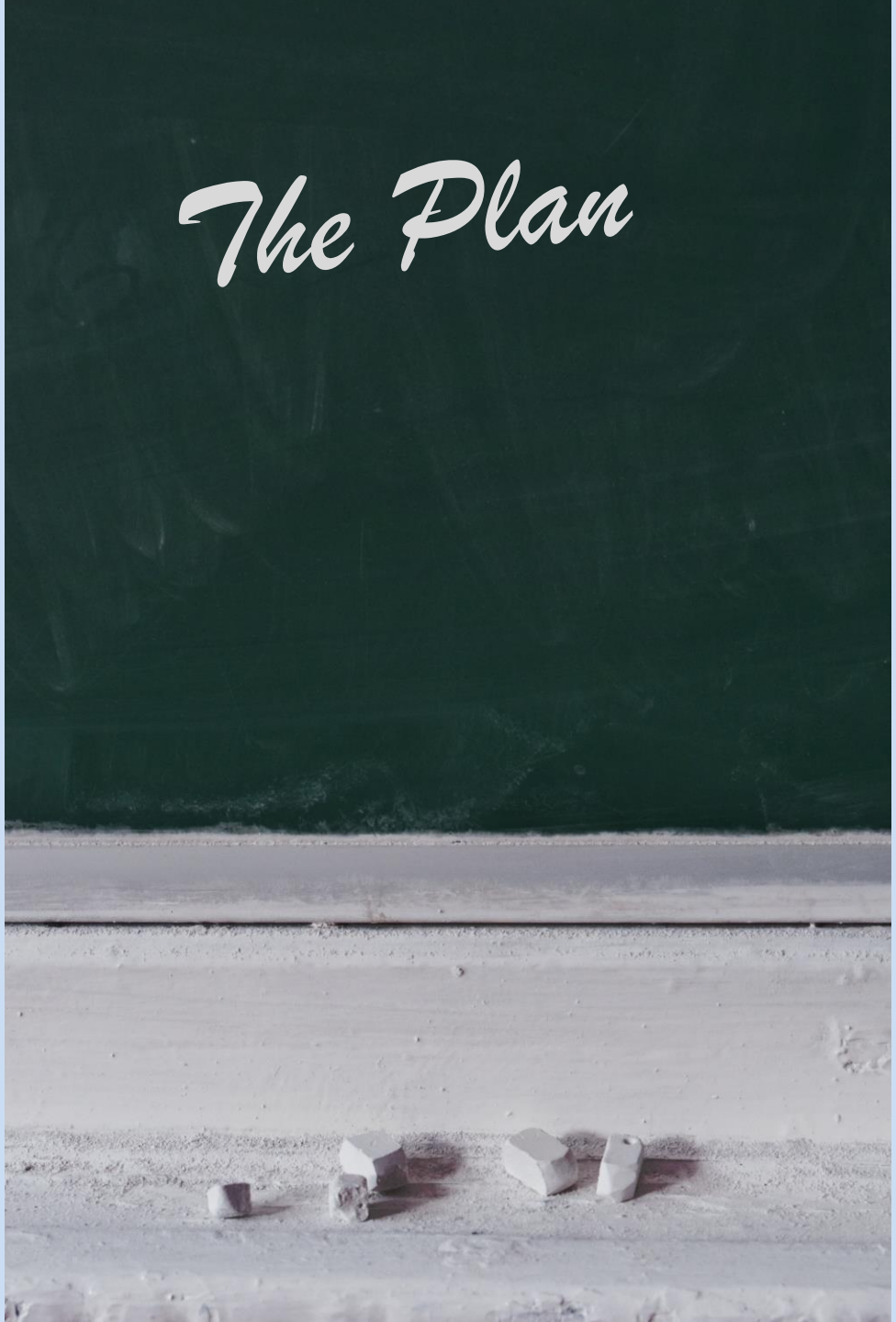
Plan for the day

09:00 Get Started!

12:30 Lunch!

17:00 Done!

17:01 **Open Question Time**



Topics

Some of the things we'll cover today

- Clean Architecture
- DDD Patterns
- Building Modular Monoliths Intentionally

Feedback

Introducing the Clean Architecture Application

A (simple) Guestbook application

Features

- Domain Model
  - Guestbook
    - Each Guestbook is unique, but any given app instance will just be one of them.
  - Entries
    - Guests leave entries which include their email and a message
- Front End
  - APIs Only (Access with .http file, Swagger, Postman, etc.)
- Use Cases
  - Create a Guestbook
  - Add a new Entry
  - Notify previous entries

Demo: Basic Working App

Uses:

ASP.NET Core

Ardalis Clean Architecture Template

Mission: Implement New Entry Use Case

Guestbook New Entry

- Validate Entry includes required email address and message
- Load appropriate Guestbook
- Add the Entry to the Guestbook
- Save the Entry
- Email signer thanking them for their message

Later additional feature:

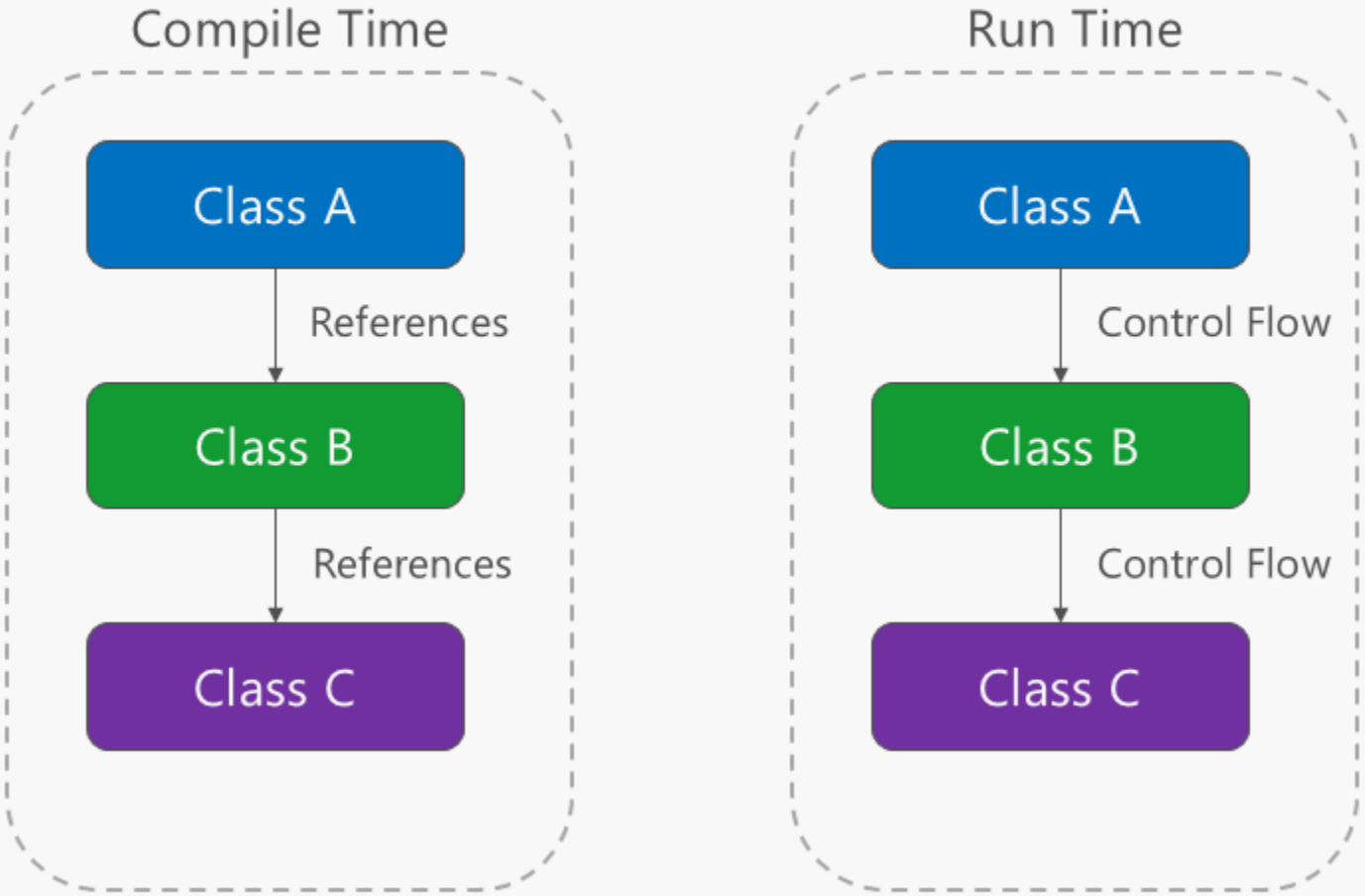
- Email previous signers notification of the new entry

But first let’s cover some basics

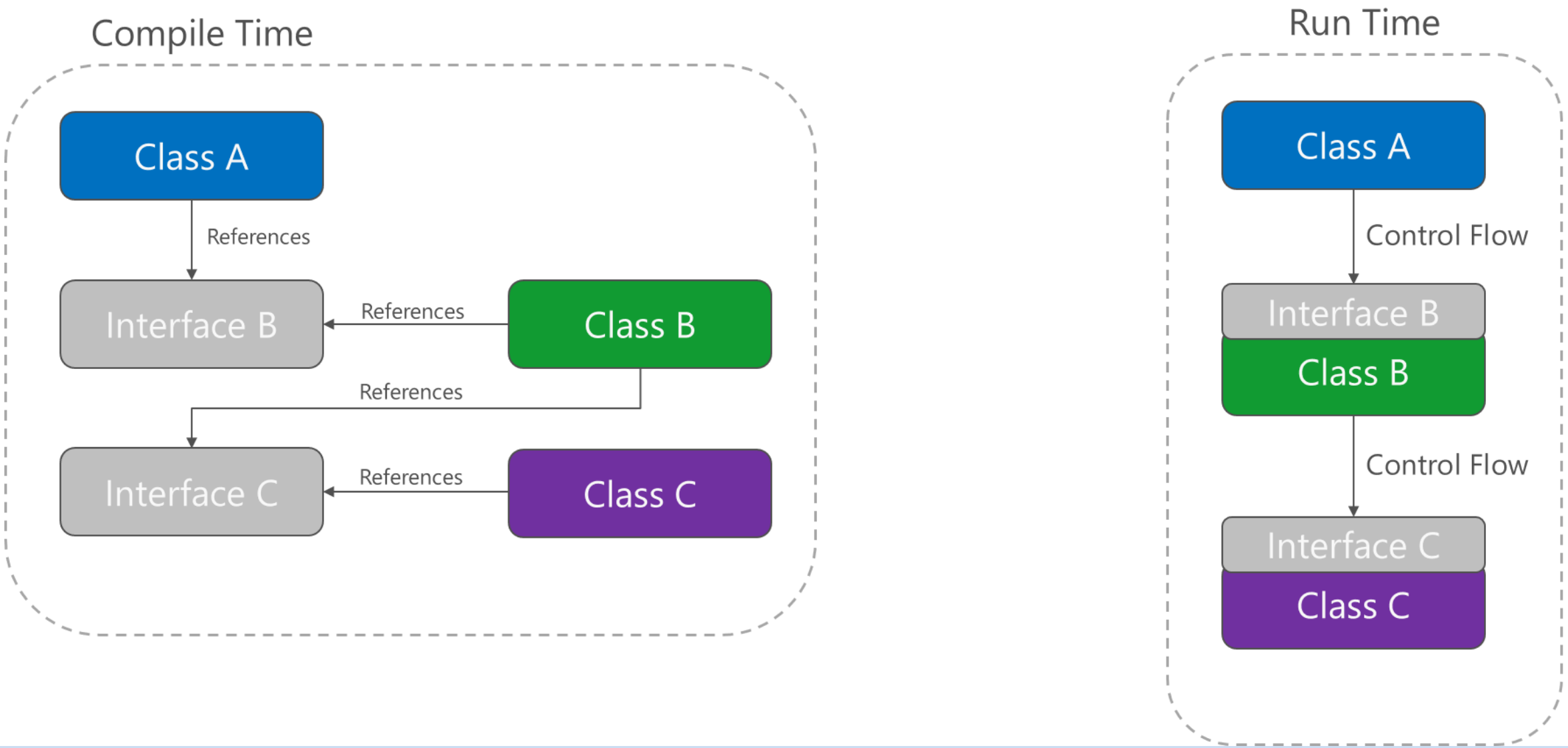
Clean Architecture

Discussed in my earlier session but quick recap for the workshop

Direct Dependency Graph



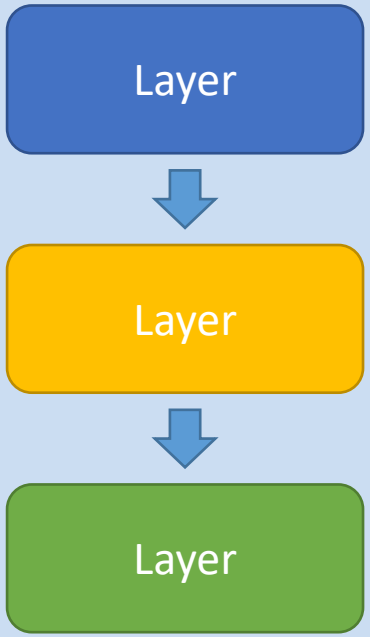
Inverted Dependency Graph



Make the right thing easy and the wrong thing hard

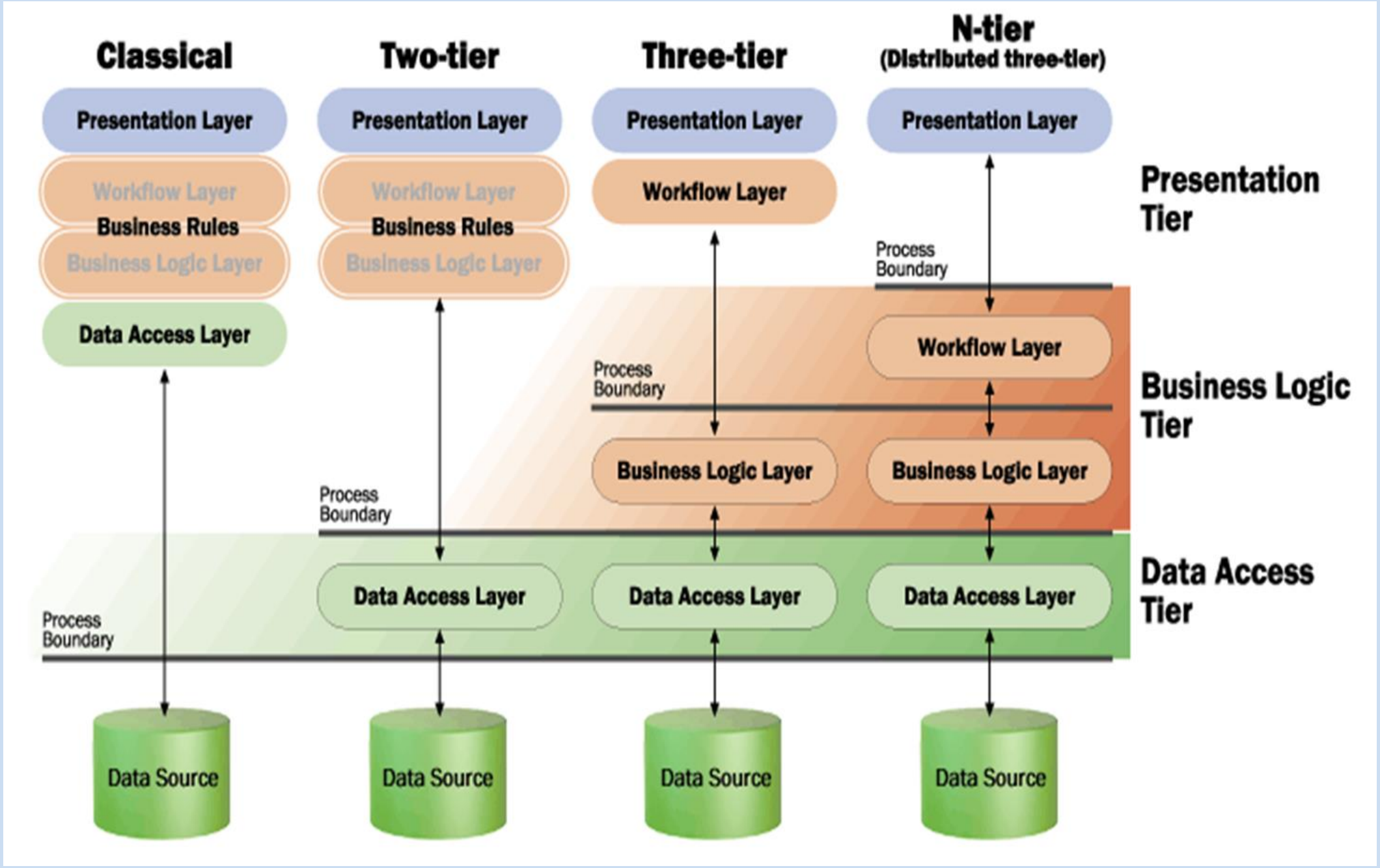
Force developers into the “pit of success!”

Otherwise, they may wind up in…

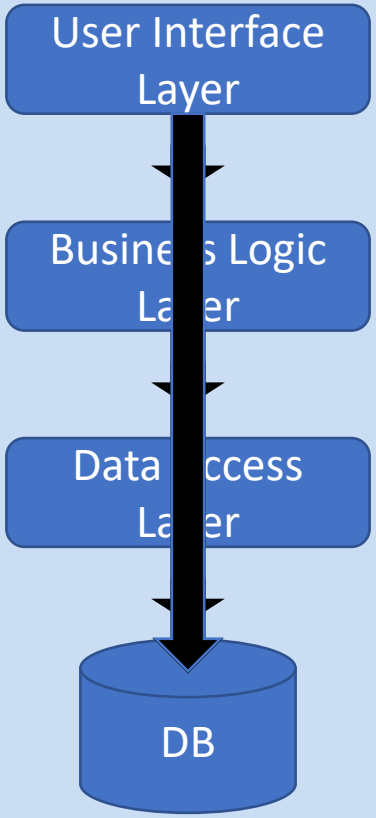


“Classic” N-Tier Architecture

Also referred to as “N-Layer Architecture”

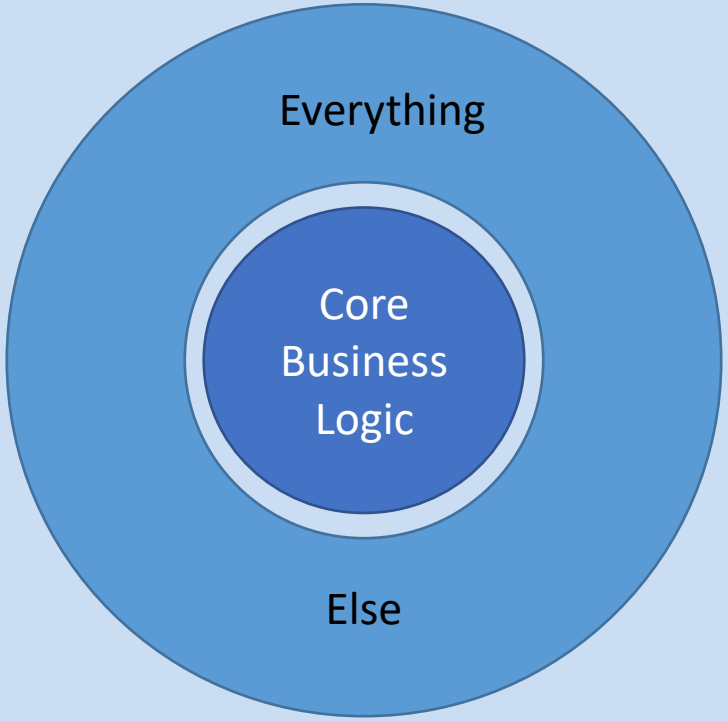

Classical
Two-tier
Three-tier
N-tier
(Distributed three-tier)
Presentation Layer
Workflow Layer
Business Rules
Business Logic Layer
Data Access Layer
Process Boundary
Presentation Tier
Business Logic Tier
Data Access Tier
Data Source

Dependencies are Transitive



Everything

Depends on the *database*



Domain-Centric Design

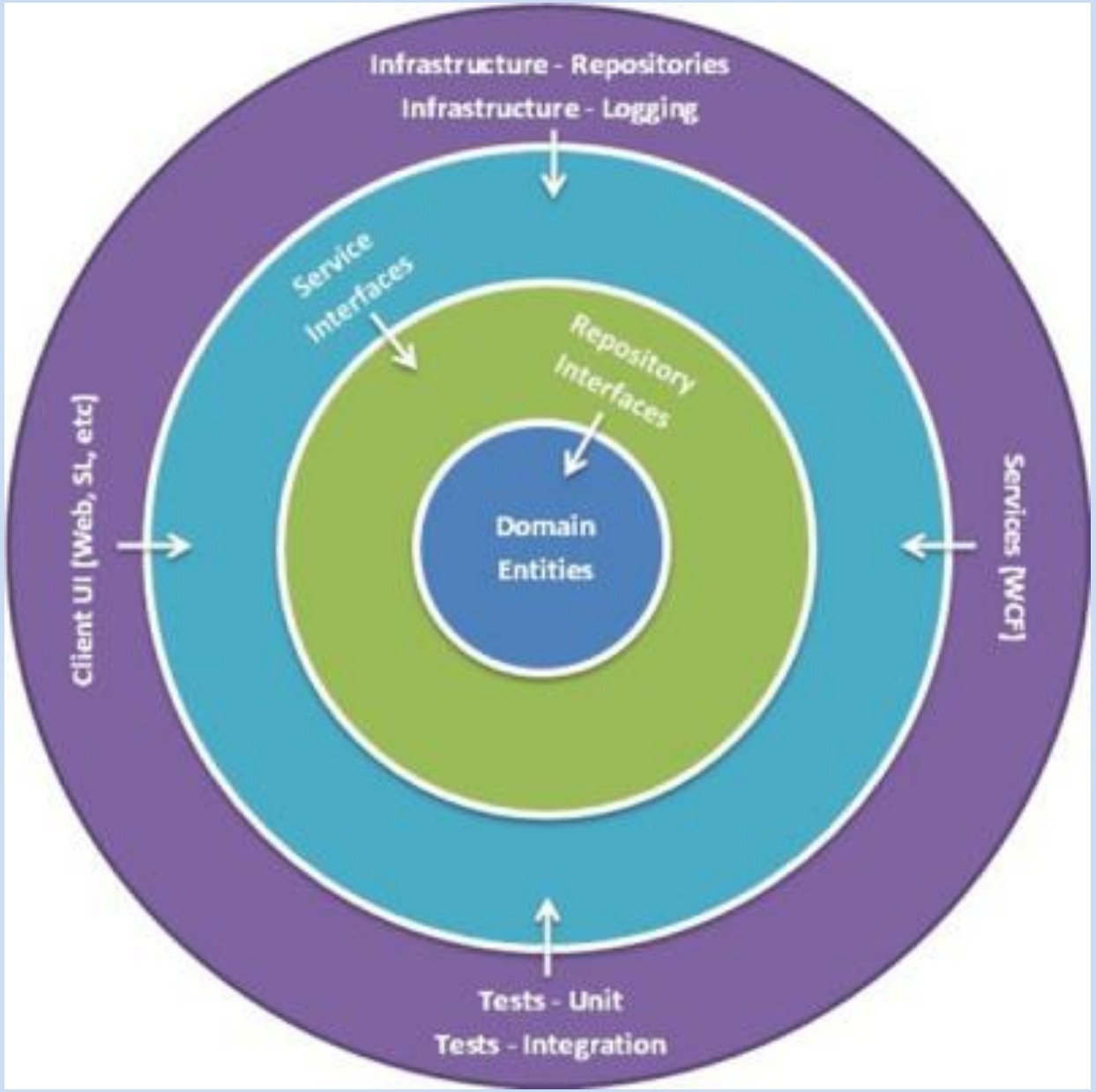
Focus on the domain model and business logic, not infrastructure

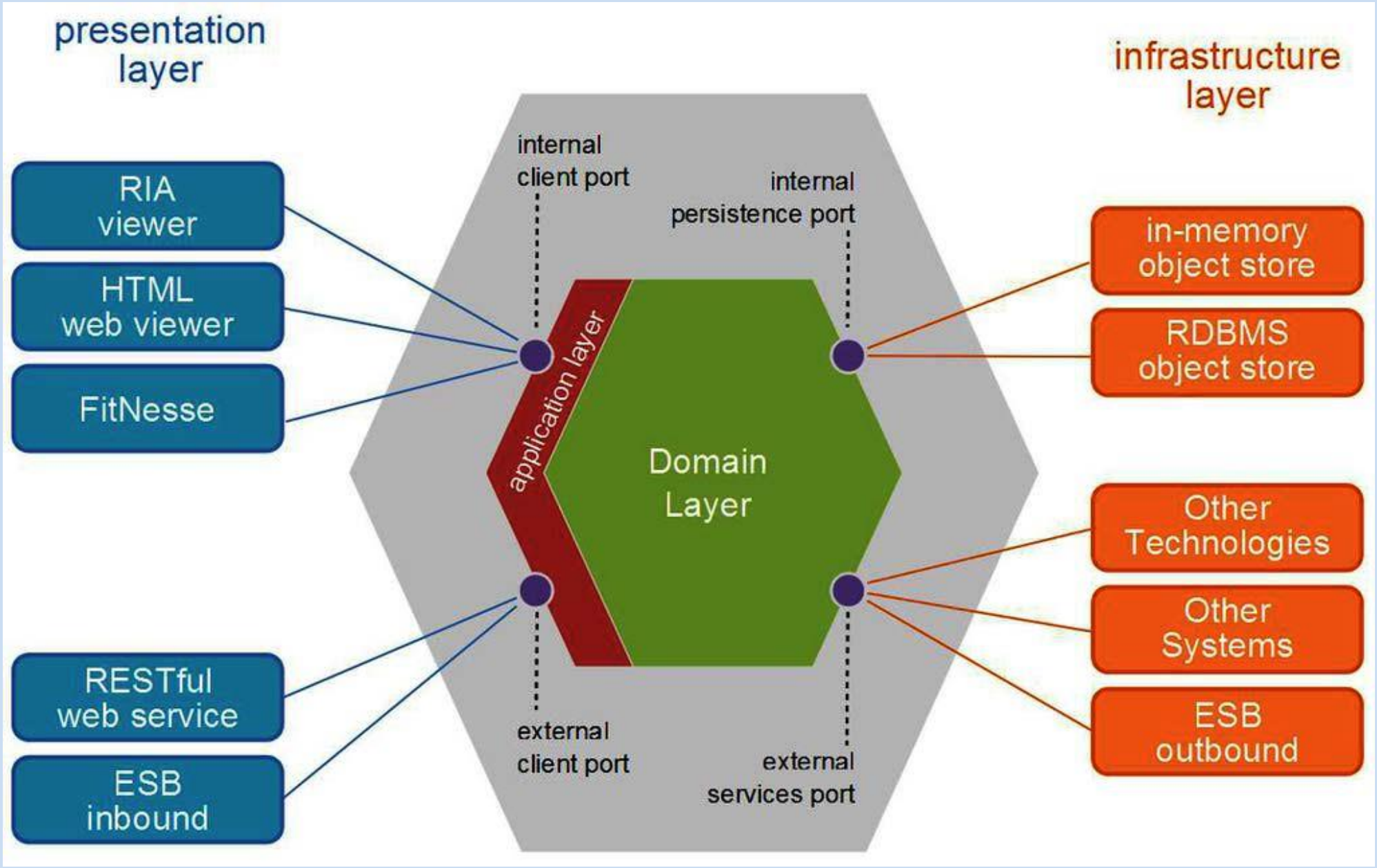
Core Business Logic

- The domain model
  - Entities
  - Aggregates
  - Value Objects
  - Domain Events
  - …and more
- Abstractions and interfaces for all required infrastructure dependencies
- Infrastructure adapters implement these interfaces
- UI constructs consume these interfaces via dependency injection

Clean Architecture

- a.k.a. Onion Architecture
- a.k.a. Hexagonal Architecture
- a.k.a. Ports and Adapters (probably the clearest name)


Infrastructure - Repositories
Infrastructure - Logging
Service Interfaces
Repository Interfaces
Domain Entities
Client UI (Web, SL, etc)
Services (WCF)
Tests - Unit
Tests - Integration


presentation layer
infrastructure layer
RIA viewer
HTML web viewer
FitNesse
RESTful web service
ESB inbound
internal client port
internal persistence port
application layer
Domain Layer
external client port
external services port
in-memory object store
RDBMS object store
Other Technologies
Other Systems
ESB outbound

Rules of Clean Architecture

The Core of the Solution holds the domain model (and all business logic).

Rules of Clean Architecture

The Core of the Solution does not depend on external dependencies.

Rules of Clean Architecture

The Core of the Solution defines abstractions/interfaces, which are implemented Core or Infrastructure

Rules of Clean Architecture

Avoid directly references to Infrastructure project types.

Exception:

- App's Composition Root
- Integration/Functional Tests

App Logic: Commands and Queries

- Application Logic can be organized into
  - **Commands** (!)
  - **Queries** (?)
- **Commands** mutate state
- **Queries** are read-only operations that return state information
- **Command/Query Responsibility Segregation (CQRS)**
- Organize app logic around Use Cases
- Each Use Case should map to a **Command** or **Query**

Organizing ASP.NET Core Apps into Clean Architecture Solutions

The Core Project (domain model)

Interfaces

Entities

Value Objects

Aggregates

Domain Services

Exceptions

Domain Events

Domain Event Handlers

Specifications

The Use Cases Project (app model - optional)

DTOs

Interfaces

Commands

Command Handlers

Queries

Query Handlers

The Infrastructure Project (dependencies)

Repositories

EF (Core) DbContext

Cached Repositories

System Clock

Query Services

Web API Clients

File System Accessors

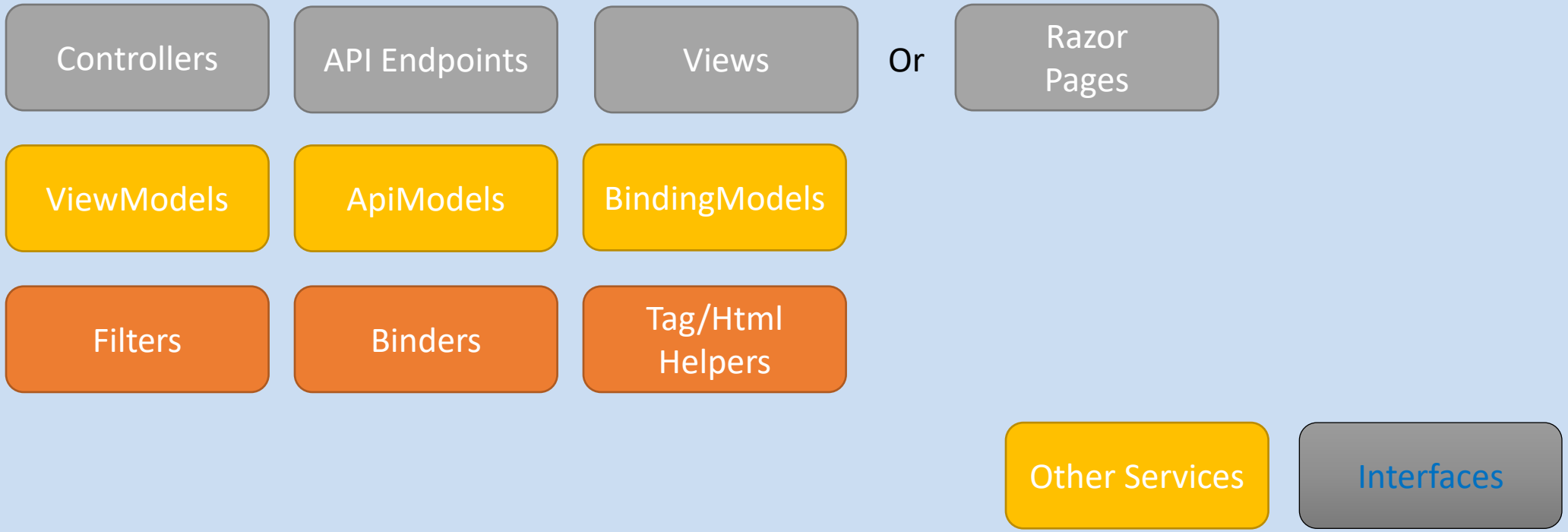
Logging Adapters

Email/SMS Sending

Other Services

Interfaces

The Web Project (dependencies)



Sharing Common Code Between Solutions

- Common types may be shared between solutions.
- Domain-Driven Design refers to this class library as a **Shared Kernel**.
- Shared Kernel is ideally distributed as a NuGet Package.

The Shared Kernel Package

Base Entity

Base Domain Event

Base Specification

Common Exceptions

Common Interfaces

Common Auth e.g. User class

Common DI

Common Logging

Common Guard Clauses

Common Pipeline Behaviors

DDD Patterns

DDD Patterns

Entities

Value Objects

Aggregates

Repositories

Domain Events

Specifications

Entities

Entities

- Have an identity
- Are long-lived and typically persisted
- Should be designed to support encapsulation
- Avoid anemic entities which have properties but no behavior

Entities

- Avoid Dependency Injection as well as static calls or direct instantiation of services that have dependencies
- But also avoid creating Domain Services to deal with this when you find you need a dependency to perform some behavior triggered within an entity method
- Instead leverage domain events if possible

Entities - Example

```
public class Customer : BaseEntity
{
    public string Name { get; private set; }
    public Customer(string name)
    {
        Name = name;
    {
    private Customer()
    {
        // required by EF
    }
}
```

Value Objects

Value Objects

- Immutable types that are compared based on the state of their properties
- Use factories or helpers to create complex value objects
- Modifying a value object should not actually modify it, but should instead return a new instance with the modification applied

- DateTime is an example of a Value Object in .NET
  - You cannot create it in an invalid state
  - You cannot change it – methods like AddDays() return a new instance

Value Objects vs. Primitive Obsession

- Primitive Obsession is a code smell recognizable by the overuse of primitive types rather than higher level classes

- Common examples include groups of related properties comprised of primitives
  - Address (street1, street2, city, state, zip, country, etc)
  - Name (firstname, lastname, middlename, prefix, suffix, etc)
    Even just a single string “ProjectName” can be a value object!
  - DateTimeRange (startdatetime, enddatetime or timespan)
  - Money
  - Id types (orderId, customerId as ints rather than custom types)
    - See: https://github.com/andrewlock/StronglyTypedId

Value Object Example

```
// NuGet package CSharpFunctionalExtensions/
public class SomeValue : ValueObject
{
  public int Value1 { get; private set; }
  public string Value2 { get; private set; }

  public SomeValue(int value1, string value2)
  {
    Value1 = value1;
    Value2 = value2;
  }
  protected override IEnumerable<object> GetEqualityComponents()
  {
    yield return Value1;
    yield return Value2;
  }
}
```

Value Objects in EF Core

- Configure them as Owned Types (typically)

```
modelBuilder.Entity<Order>()
  .OwnsOne(o => o.ShippingAddress);
```

- They’re persisted in the same table as their parent entity
  - Property name (of VO) is prefixed before each VO property’s column
  - Example: ShippingAddress_City, ShippingAddress_State, etc.
- Always returned with the entity
  - No need to .Include(o => o.ShippingAddress)

Vogen (and other libraries)

- VO-jen: Value Object Generator
- https://github.com/SteveDunn/Vogen
- Strongly-Typed IDs:

```
[ValueObject<int>]
public partial struct CustomerId;
```

- Simple value objects for strings, etc.:

```
[ValueObject<string>]
public partial class LegalEntityName;
```

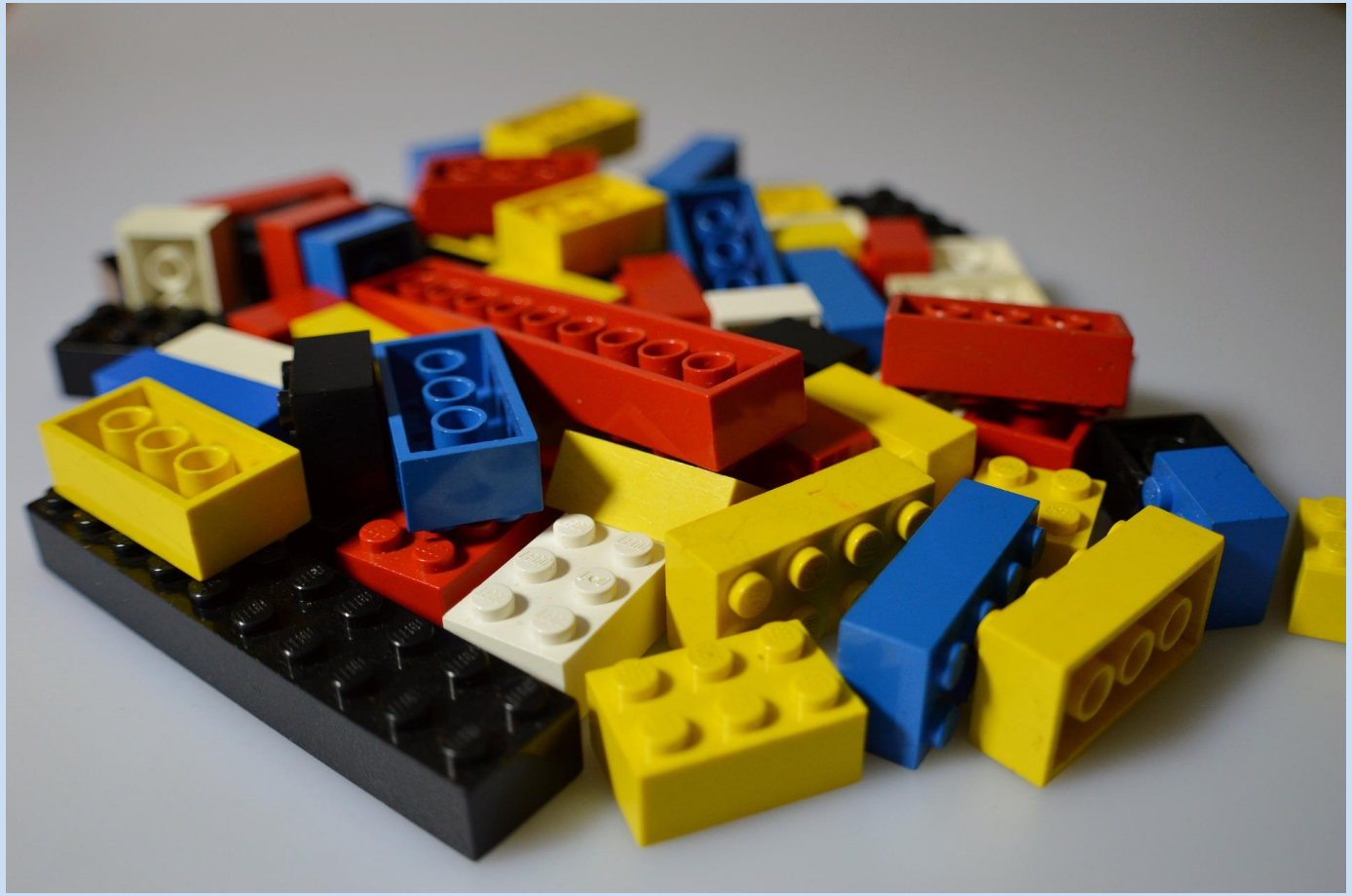
Aggregates

Aggregates

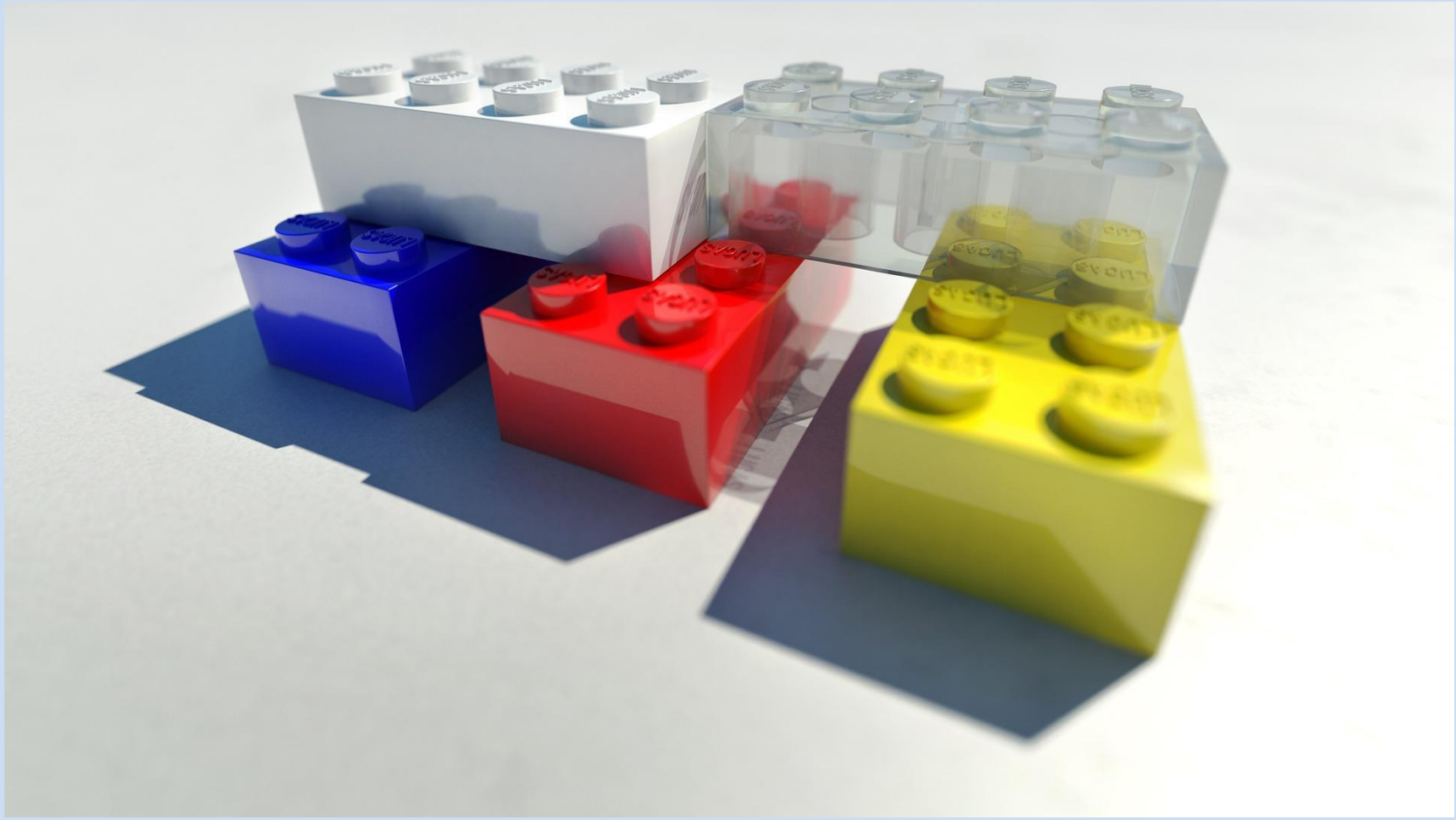
"An aggregate is a cluster of associated objects that we treat as a unit for the purpose of data changes."

Eric Evans, *Domain-Driven Design*

Aggregates – Instead of this…



Aggregates – do this



Aggregates

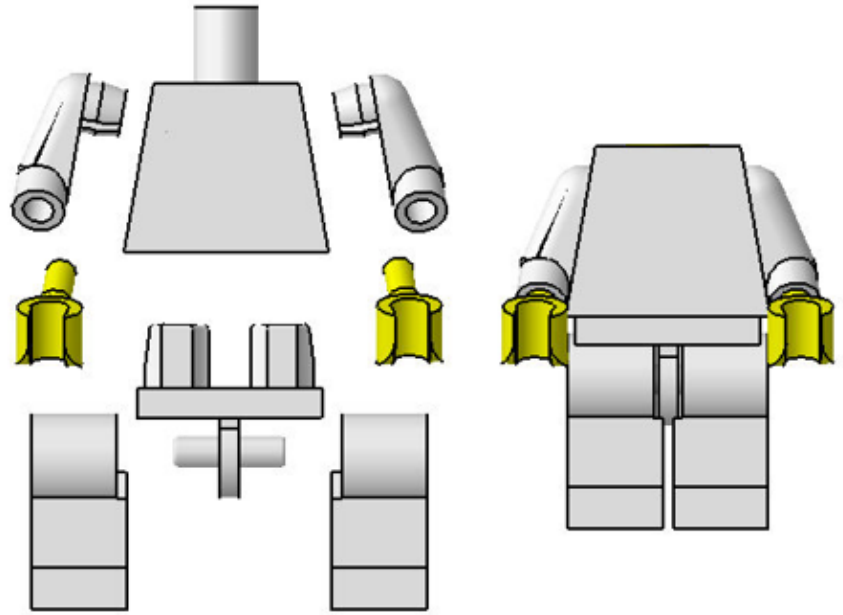
- Provide another encapsulation boundary
- Must be persisted and retrieved as a unit
  - Child entities cannot be persisted directly
- Can enforce rules on collections that are otherwise difficult to perform (and usually can’t be performed on the entities themselves)
- Example:
  - A Purchase Order class contains a collection of LineItems and a Limit
  - The sum of the LineItems should never be allowed to exceed the Limit
  - LineItems don’t know about each other and can’t enforce this rule, but Purchase Order can

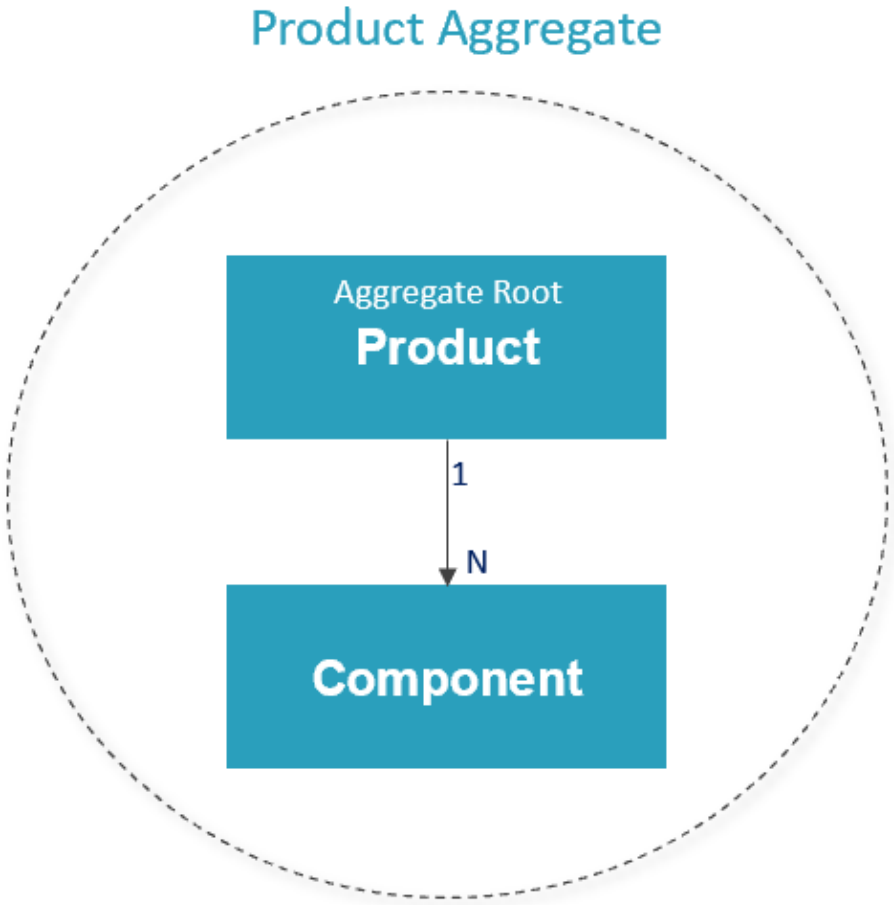
Aggregates

- Use an interface or base class to identify which entities are aggregate roots
- Enforce data access policy using generic constraints

- Aggregates can reference other aggregates, but only by ID, not navigation properties

Aggregates Enforce Business Rules





Identifying Aggregates

- Look for master-detail relationships
- Use the “cascading deletes” test: if I delete the root of the aggregate, should all of its children be deleted as well?

Enforcing Aggregates

```
// A Buyer has collection of payment methods.
// They shouldn't be modified independently – Avoid:
var payMethod = _db.PaymentMethods.Find(123);

// make some changes

_db.SaveChanges();
```

Enforcing Aggregates

```
// designate Buyer as an aggregate root
public class Buyer : BaseEntity, IAggregateRoot
{
…
}
// Enforce operations only on aggregate roots in Repositories
```

Lab 1: App and List Endpoint

Repositories

Repository

- A persistence abstraction
- Promotes separation of concerns
- Communicates design intent
- Enables better testability
- Improves maintainability

"But we don't need a repository because EF Core is a repository!"

-- people on the Internet

A repository is an **abstraction**.

The entire value of the pattern is tied to this fact.

Abstractions should not depend on implementations.

EF Core is an implementation.

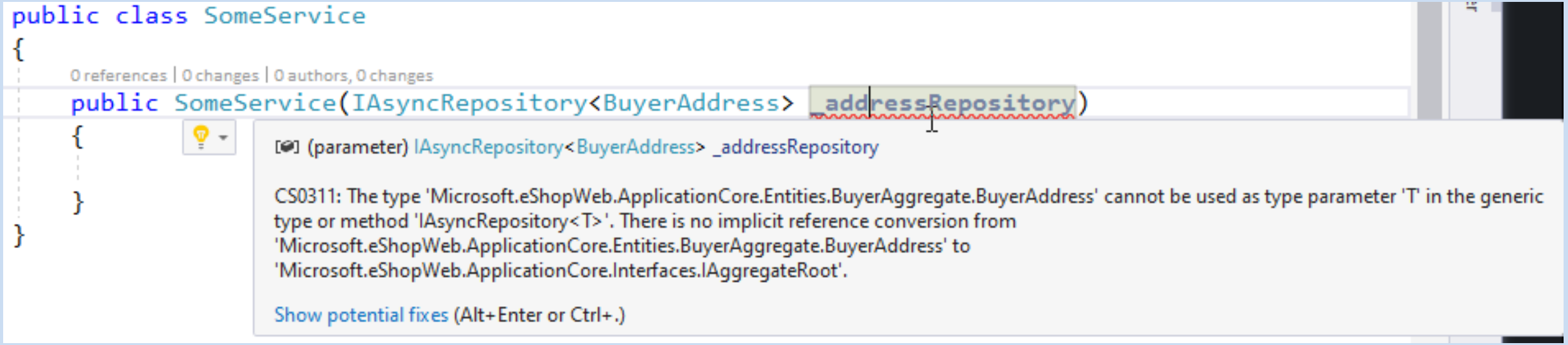
Don’t

- Return `IQueryable`
- Return DTOs or other types
- Use lazy loading (in web apps)

Do

- Leverage caching via CachedRepository pattern
- Enforce Aggregate data access policy using generic constraint
- Combine with Specification to follow Open/Closed Principle

Enforcement of Data Access Policy



Lab 2: Persisting Entries

Domain Events

Domain Events

- Something of interest that happened in the domain
  - Appointment Confirmed
  - Checkout Completed
  - Profile Updated

- Same concept as UI and other events, applied to your domain model

Domain Event Design

- Past Tense – something happened in the past
- Immutable – we can’t change the past
- Data Only – Events are messages; think of them as immutable DTOs

Requirements Analysis and Events

- Listen for: “when this happens, then this should happen” in your requirements

- Examples
  - When a customer schedules an appointment, send them an email confirmation
  - When the library book becomes available, notify the next person on the waitlist
  - When the order is completed, send a request to fulfillment

Consider Design with and without events

Given a customer has created an order

When the customer completes their purchase

Then the system sends the customer an email confirmation

One Solution

An OrderService

- With a Method ‘Checkout(Order order)’
- Which performs these steps:
  - Save pending order in database
  - Send confirmation email

One Solution

Simple! Ship it!

More Requirements

- Given a customer has created an order
- When the customer completes their purchase
  - **Then the customer's card is charged**
    - **If it fails, send a message to the customer**
  - **Then inventory is checked**
    - **If insufficient, send a message to the customer**
- Then the system sends the customer an email confirmation

Extend the simple solution

An OrderService

With a Method ‘Checkout(Order order)’

Which performs these steps:

- Save pending order in database
- Attempt to process payment
  - If payment fails, send an email
- Confirm available inventory
  - If insufficient inventory, send an email
- Send confirmation email

One Solution

Much less simple…

Analysis

- Complexity is growing rapidly
- Method changes with every requirement change
  - Open/Closed Principle violation
- Checkout/OrderService Responsibilities keep growing
  - Single Responsibility Principle violation
- Potentially different levels of abstraction
  - Email sending details combined with business rules

The Hollywood Principle

"Don't call us; we'll call you."

Reduce coupling with inversion of control

Refactor to use Domain Events

- Events allow other code to respond when something happens
- The code initiating the change doesn't need to know about all of the follow-on code the change may create
- You're probably already familiar with this programming model from UI programming

New Solution with Event Handlers

- **OrderPaymentHandler**
  - Charge card
  - Send command to notify customer on failure
- **OrderInventoryHandler**
  - Verify inventory
  - Send command to notify customer if insufficient
- **OrderSuccessNotificationhandler**
  - Send command to notify customer order has been placed successfully

Domain Event Qualities

- Model something that happens of interest to a domain expert
- Listen for **“when this happens, then this happens”** in conversations
- Part of the domain model; live in Core; may be grouped with an Aggregate
- Are **usually handled synchronously** within the application (not persisted or sent over a bus or queue)

Event Processing

- Events are raised or queued and then later dispatched
- Handlers **within the current process** handle the event
- If external apps need to be notified, specific handlers can adapt and enrich domain events into **integration events** which are sent outside of the domain
- If unknown or future apps will need to respond to events, a **service bus** or **topic** can be used
  - Otherwise integration events typically write to specific **queues**

Lab 3: Notifying Users of New Entries

Specifications

Specifications

- Define a set of criteria that other objects match or don’t match
- Encapsulate within a well-named class as part of your domain model
- Support discoverability and reuse

Simple Implementation

```
using System;
using System.Linq.Expressions;

namespace Core.Interfaces
{
    public interface ISpecification<T>
    {
        Expression<Func<T, bool>> Criteria { get; }
    }
}
```

Ardalis.Specification

Ardalis.Specification by: ardalis

3,469,925 total downloads last updated 4 months ago Latest version: 7.0.0

spec specification repository ddd

A simple package with a base Specification class, for use in creating queries that work with Repository types.

Ardalis.Specification.EntityFrameworkCore by: ardalis

2,705,955 total downloads last updated 4 months ago Latest version: 7.0.0

spec specification repository ddd ef core entity framework

EF Core plugin package to Ardalis.Specification containing EF Core evaluator and abstract repository.

Ardalis.Specification.EntityFramework6 by: ardalis

25,531 total downloads last updated 4 months ago Latest version: 7.0.0

spec specification repository ddd ef ef6 entity framework

EF6 plugin package to Ardalis.Specification containing EF6 evaluator and abstract repository.

Specification Benefits

- Provides blocks of LINQ with a **Name**
- Improves **Separation of Concerns**
- Improves **Single Responsibility Principle**
- Lets Repositories follow **Open/Closed Principle**
- Keeps data access and business logic from spreading uncontrollably
- Lives as part of domain model
- Can be tested independently

Replace IQueryable

```
// replace
public interface IFooRepository
{
    IQueryable<Foo> List();
}

// with
public interface IFooRepository
{
    IEnumerable<Foo> List(ISpec<Foo> spec);
}
```

Specifications: Not Just About Filtering

- .Include() related collections
- .Select() to transform results
- Paging support with Skip() and Take()
- Cache Support

- Can also be used to represent rules in rule engine implementations

Query Services

- Don’t use Specifications for non-domain results
- Specifications, Repositories are part of domain model which is focused primarily on changes, **not** ad hoc reads!

Examples:

- Custom reports
- Ad hoc queries
- Search results

If you just need to query the database and get back a custom result type, use a Query Service instead of Specification/Repository!

Query Services

- Not part of domain model
- Usually interface is defined in Use Cases project
- Implementation belongs in Infrastructure
- Can use whatever querying approach makes sense
  - Custom SQL (or other query language)
  - Stored Procedures
  - Dapper (even if you use EF elsewhere)
- Each service can use different dependencies if that makes sense!

Lab 4: Using Specifications for Data Access

Mission: Add Email Notifications

(when a new Entry is added)

Applying Architecture Principles and Considerations

- How might we design a solution to this requirement?
- What architectural principles and capabilities should we apply?
- What are the trade-offs involved?
- How do we build each option?
- How do we deploy each option?
- How do we support each option in production?

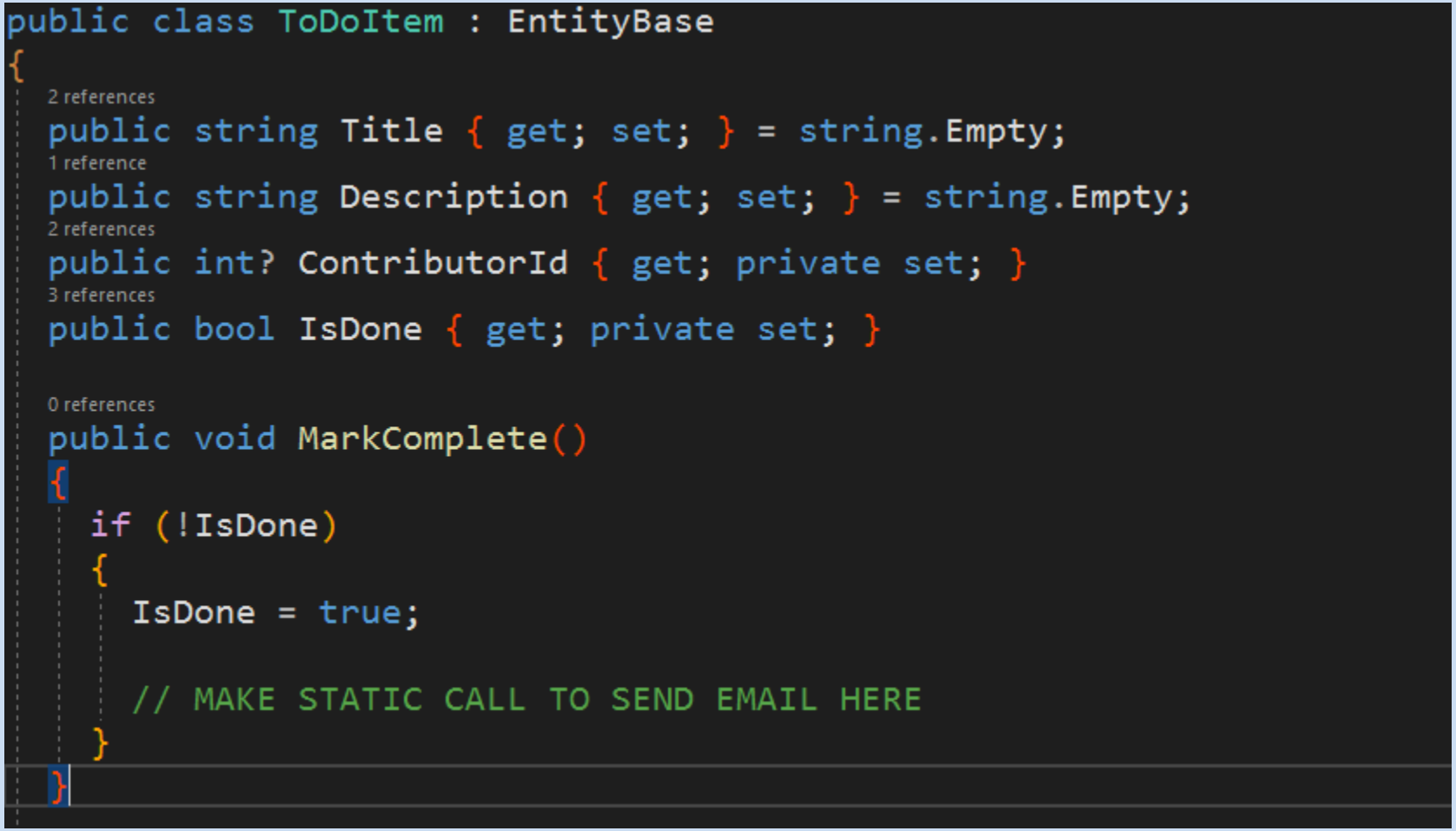
Design Option 1: Just Hardcode It!

- In the API endpoint, just send the email
- Or in an application service / use case handler
- Or in a domain service
- Or in the GuestbookEntry entity
- Or in an in process event handler (in response to a domain event)

- All variations on the same approach – send the email immediately and in the existing app process

Hardcode it in the Endpoint

Hardcode it in the entity



```
public class ToDoItem : EntityBase
{
    2 references
    public string Title { get; set; } = string.Empty;
    1 reference
    public string Description { get; set; } = string.Empty;
    2 references
    public int? ContributorId { get; private set; }
    3 references
    public bool IsDone { get; private set; }

    0 references
    public void MarkComplete()
    {
        if (!IsDone)
        {
            IsDone = true;

            // MAKE STATIC CALL TO SEND EMAIL HERE
        }
    }
```

Hardcode it in an event handler

```
using Ardalis.GuardClauses;
using DevInt.ToDoApp.Core.Interfaces;
using DevInt.ToDoApp.Core.ProjectAggregate.Events;
using MediatR;

namespace DevInt.ToDoApp.Core.ProjectAggregate.Handlers;

1 reference
public class ItemCompletedEmailNotificationHandler : INotificationHandler<ToDoItemCompletedEvent>
{
  private readonly IEmailSender _emailSender;

  // In a REAL app you might want to use the Outbox pattern and a command/queue here...
  0 references
  public ItemCompletedEmailNotificationHandler(IEmailSender emailSender)
  {
    _emailSender = emailSender;
  }

  // configure a test email server to demo this works
  // https://ardalis.com/configuring-a-local-test-email-server
  0 references
  public Task Handle(ToDoItemCompletedEvent domainEvent, CancellationToken cancellationToken)
  {
    Guard.Against.Null(domainEvent, nameof(domainEvent));

    return _emailSender.SendEmailAsync("test@test.com", "test@test.com",
      $"{domainEvent.CompletedItem.Title} was completed.", domainEvent.CompletedItem.ToString());
  }
}
```

Analysis

- All of these options happen synchronously as part of a web request
- All of these options keep the logic and implementation in the same monolithic app – nothing is distributed
- This is a great place to start many features of new features!
  - Evolve if or when needed to a more complex architecture

Demo: Initial App

Show email sending working with MediatR event handlers

Design Option 2: Separate Worker in same App

- Dispatch a command to do the work in a separate process
- Host the separate process as a WorkerService / BackgroundWorker collocated with the existing app
- Analysis
  - Same logic.
  - Still in the same application.
  - But now it's out of process and requests can complete without waiting for the operation to complete.
  - A good interim step before moving to a full distributed/microservices approach (if needed)

Design Option 3: Separate Service for Sending Emails

- Dispatch a command to do the work to a persistent queue/bus
- Host a separate service to perform email-sending work
- Connect the service to the persistent queue/bus

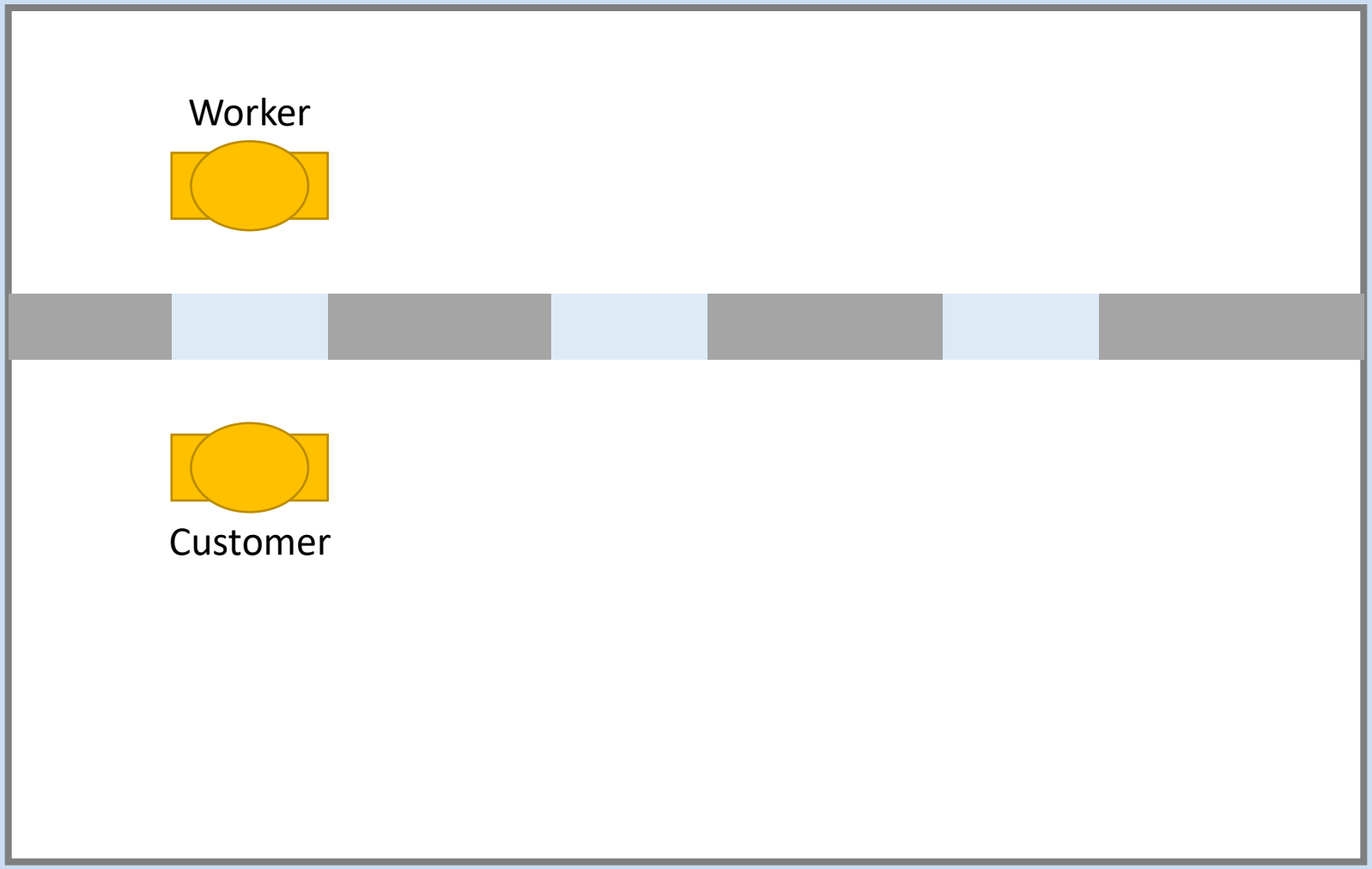
Web API Requests as Office Interactions

A useful metaphor to help understand REST, resources, and HTTP

Web Processing as a Government Office

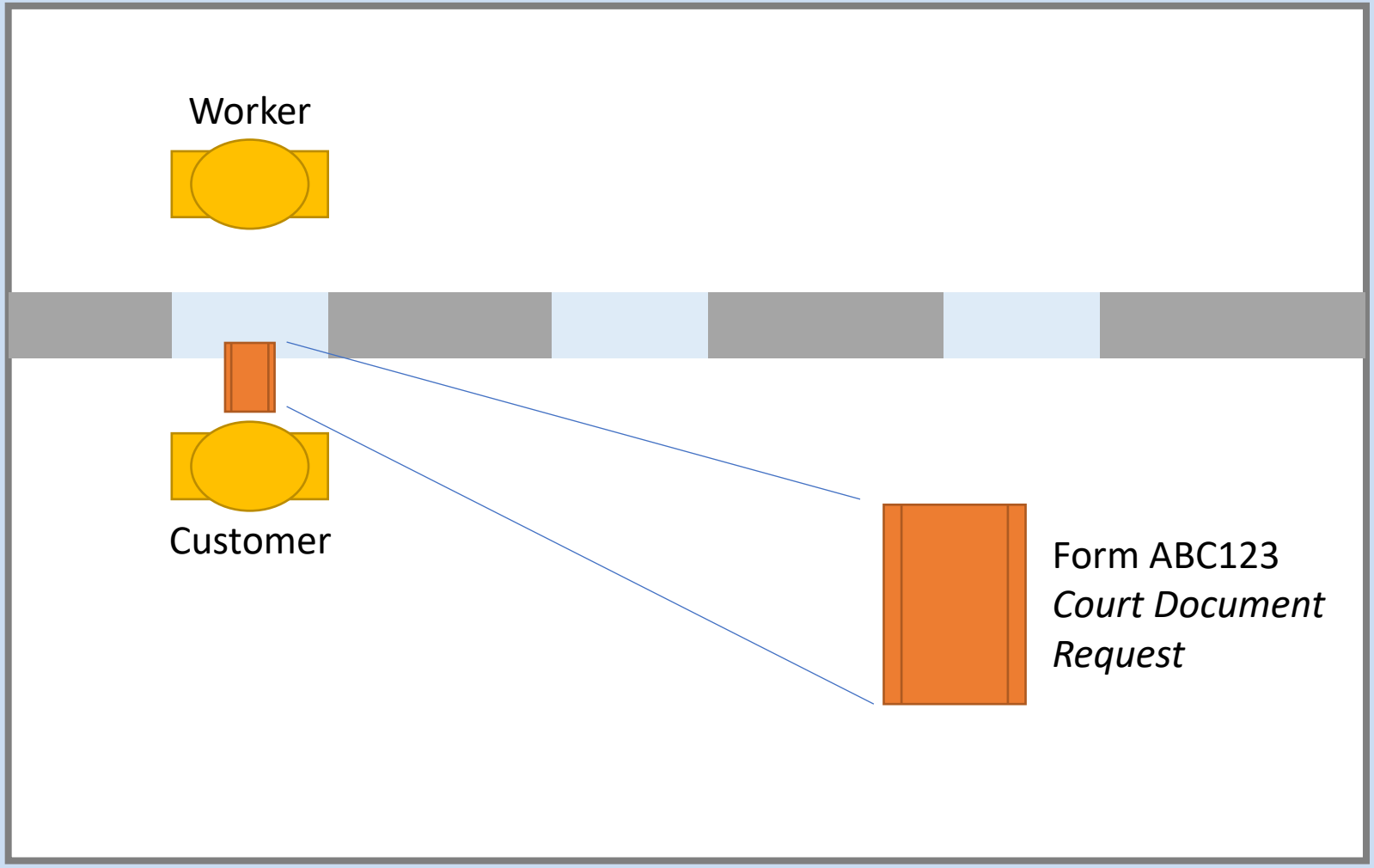
Country records office, bank, post office, etc.

Workers and Customers



A customer wants a copy of a document

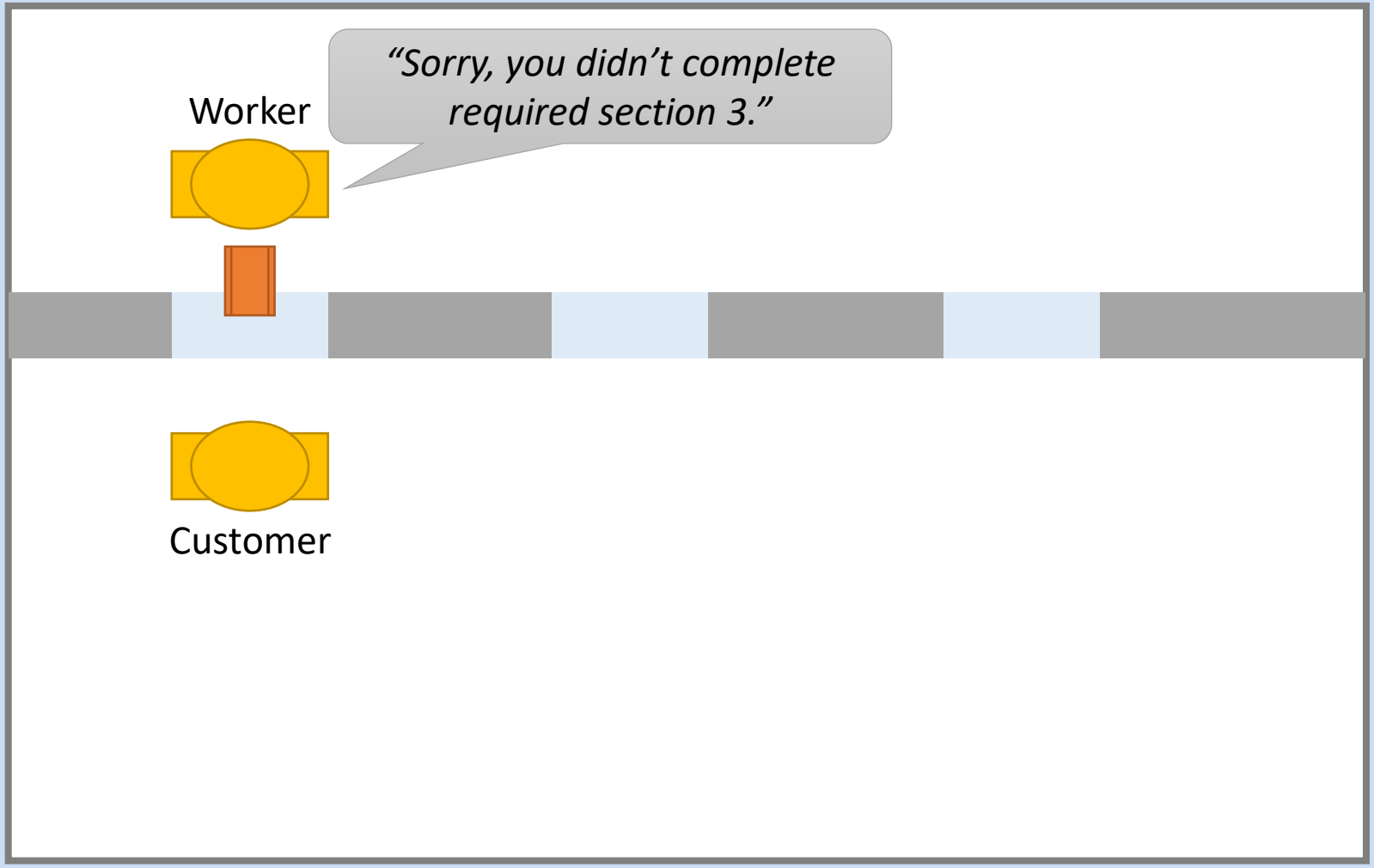
Initiating a request



- Worker = Web API
- Customer = Web Client
- Form = Request (DTO)

Note, you don't send the resource, you send a document that represents a query or state change request

Request validation

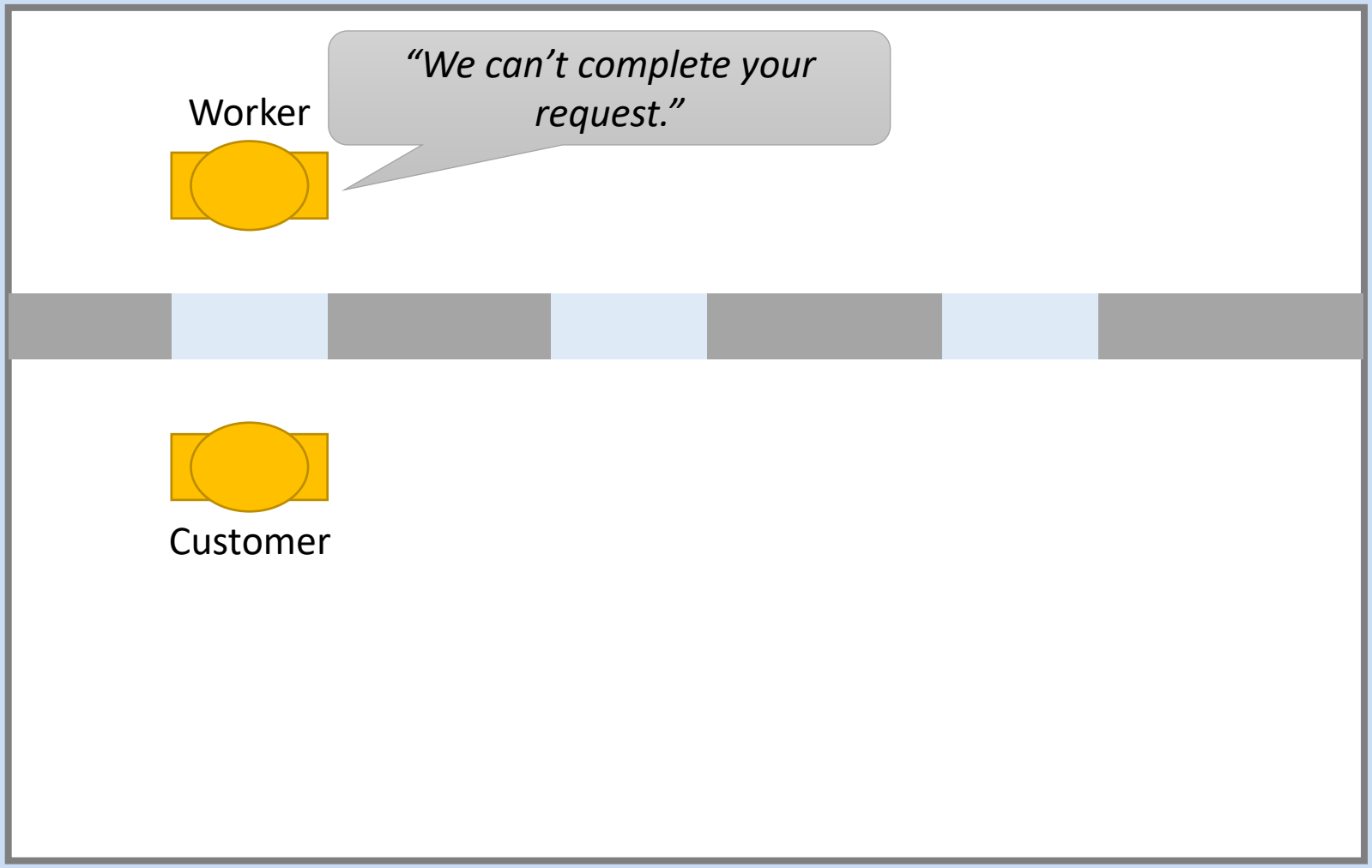


Response: 400 Bad Request

The form – the message – is validated.

This check is done up front by the worker, before any back office worker applies domain logic.

Domain validation errors

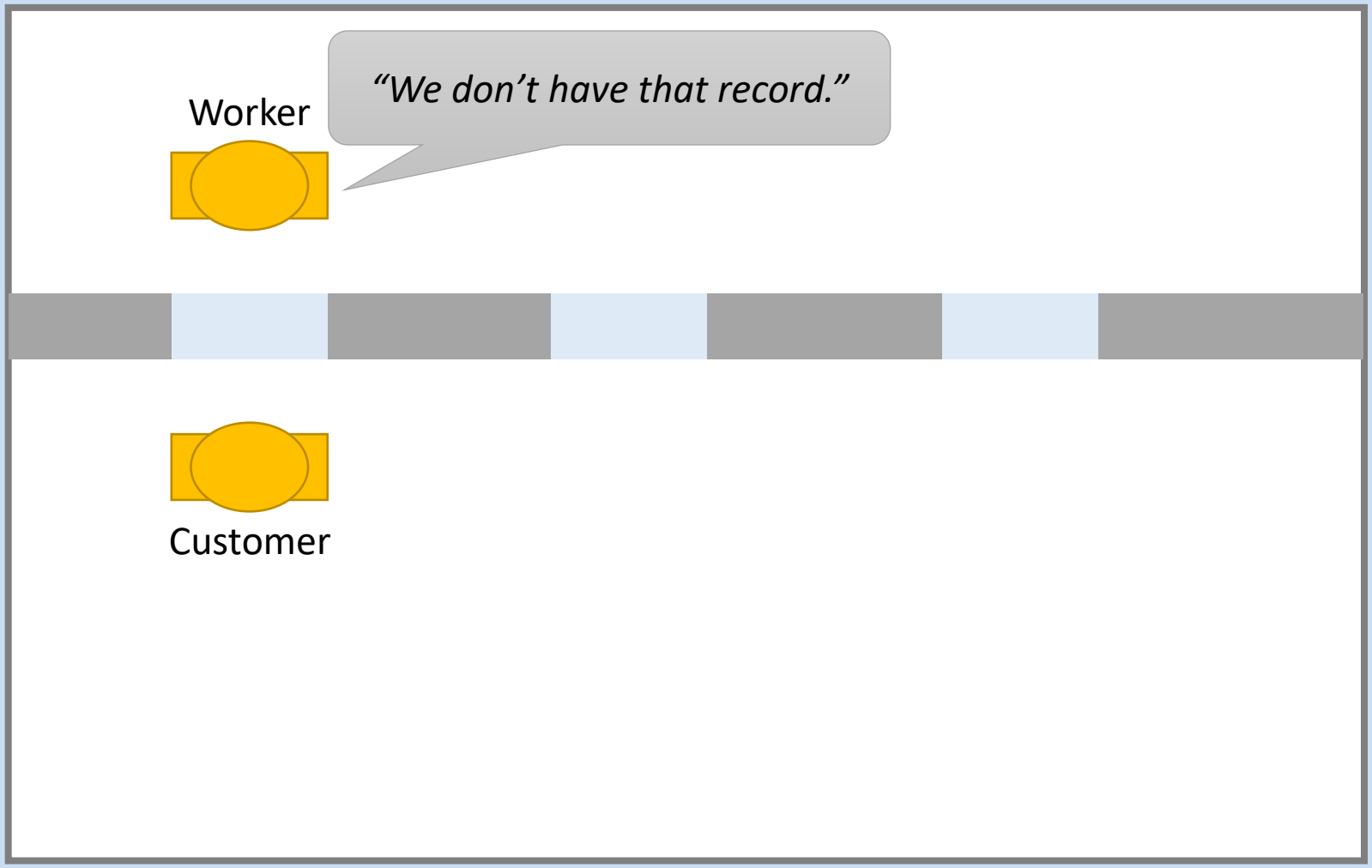


Response: 400 Bad Request

Despite a valid form, the worker determines that the request cannot be completed because of some business rule.

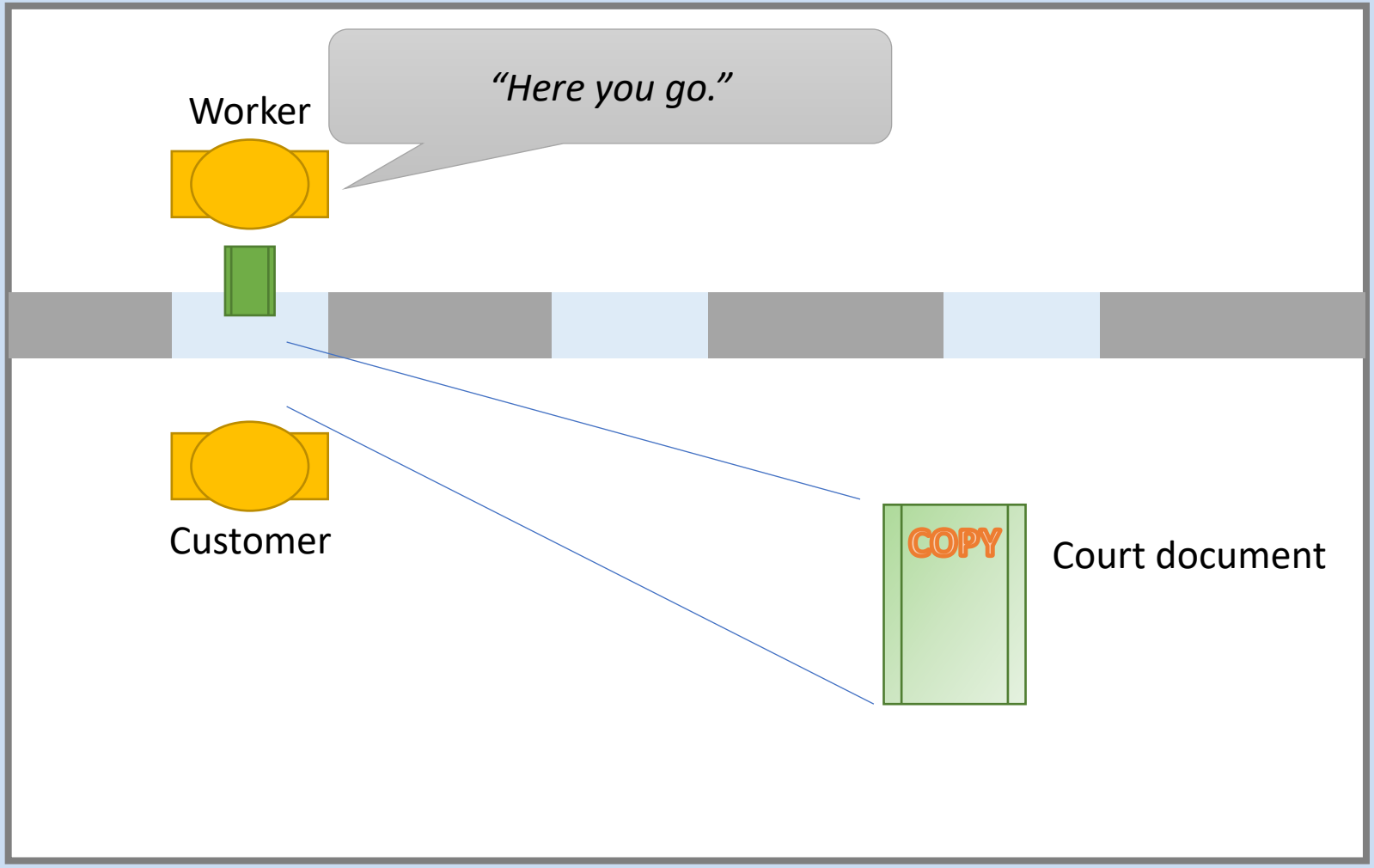
This isn't exceptional behavior, so typically should not be represented by an exception or a 500 response.

Resource not found



Response: 404 Not Found

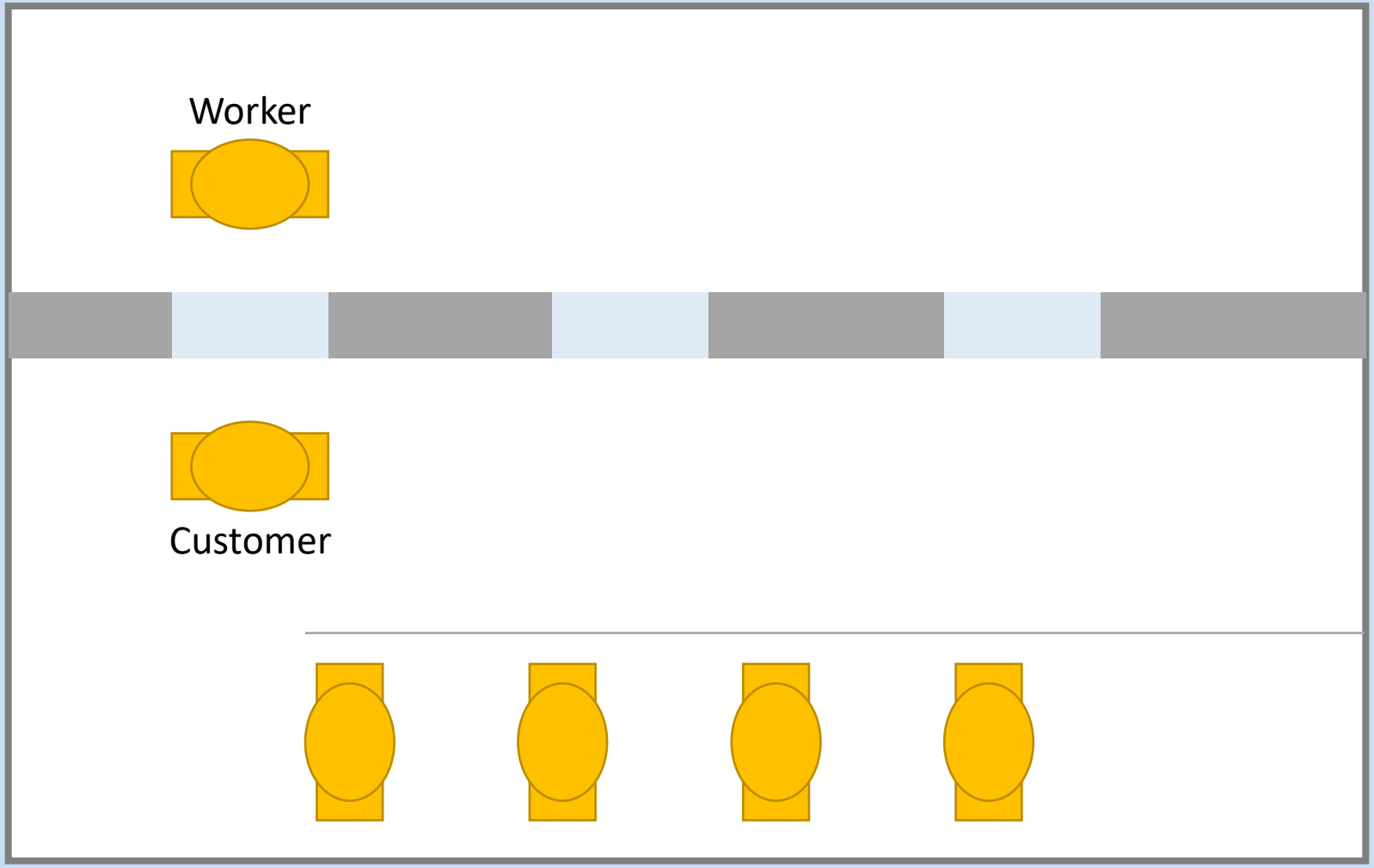
Success



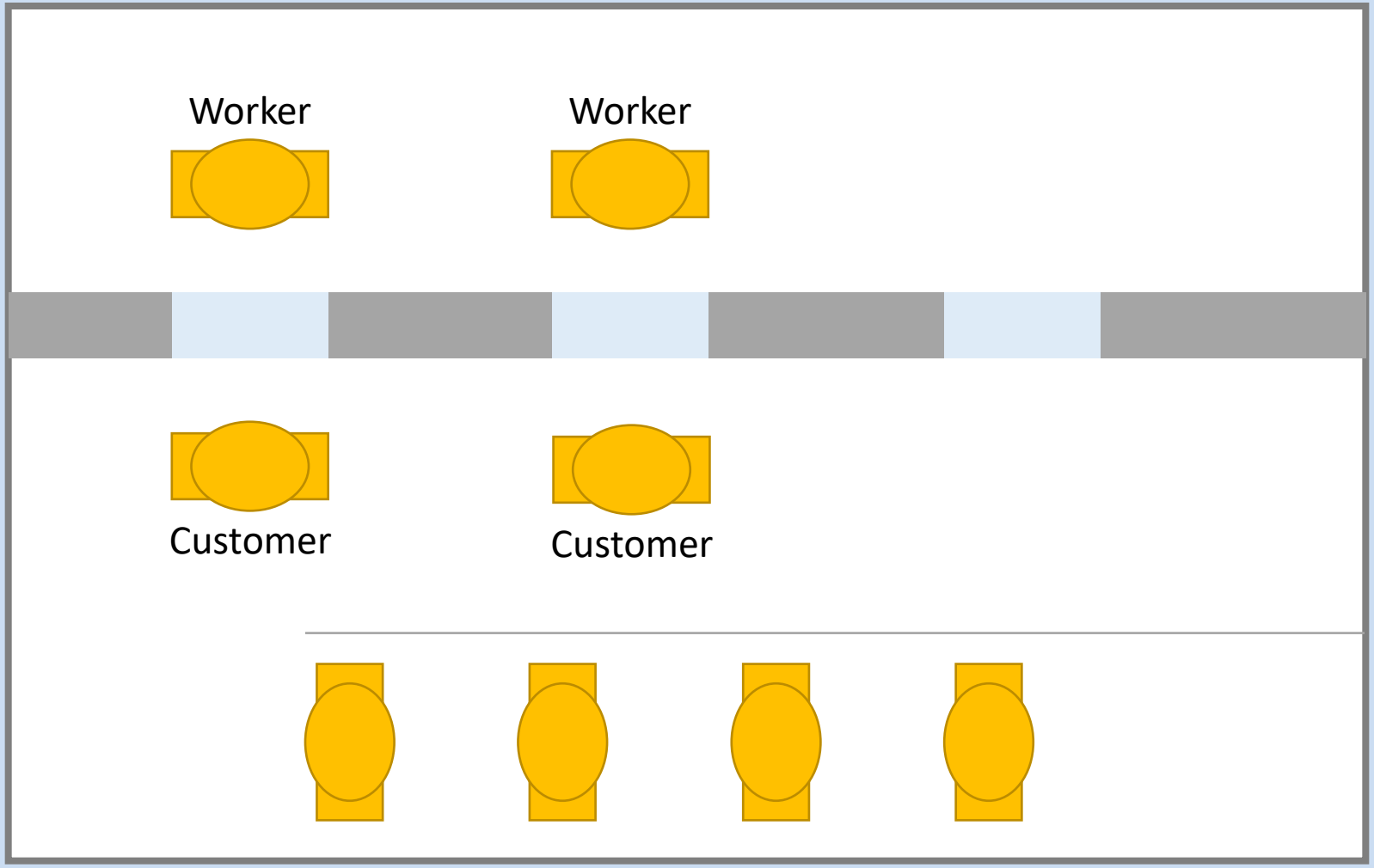
Response: 200 Success

Note: You don't get the requested resource, you get a copy of it. You may also get other data, such as a receipt or a datestamp showing when the request was completed, etc.

Request queueing



Worker threads



Async Worker threads