Implementing Microservice Domain Logic

Mark Heath
CLOUD ARCHITECT

@mark_heath www.markheath.net

Overview

Domain logic patterns

- Transaction script
- Domain model
- Table module
- Serverless approach

Backend for frontend

Data access patterns

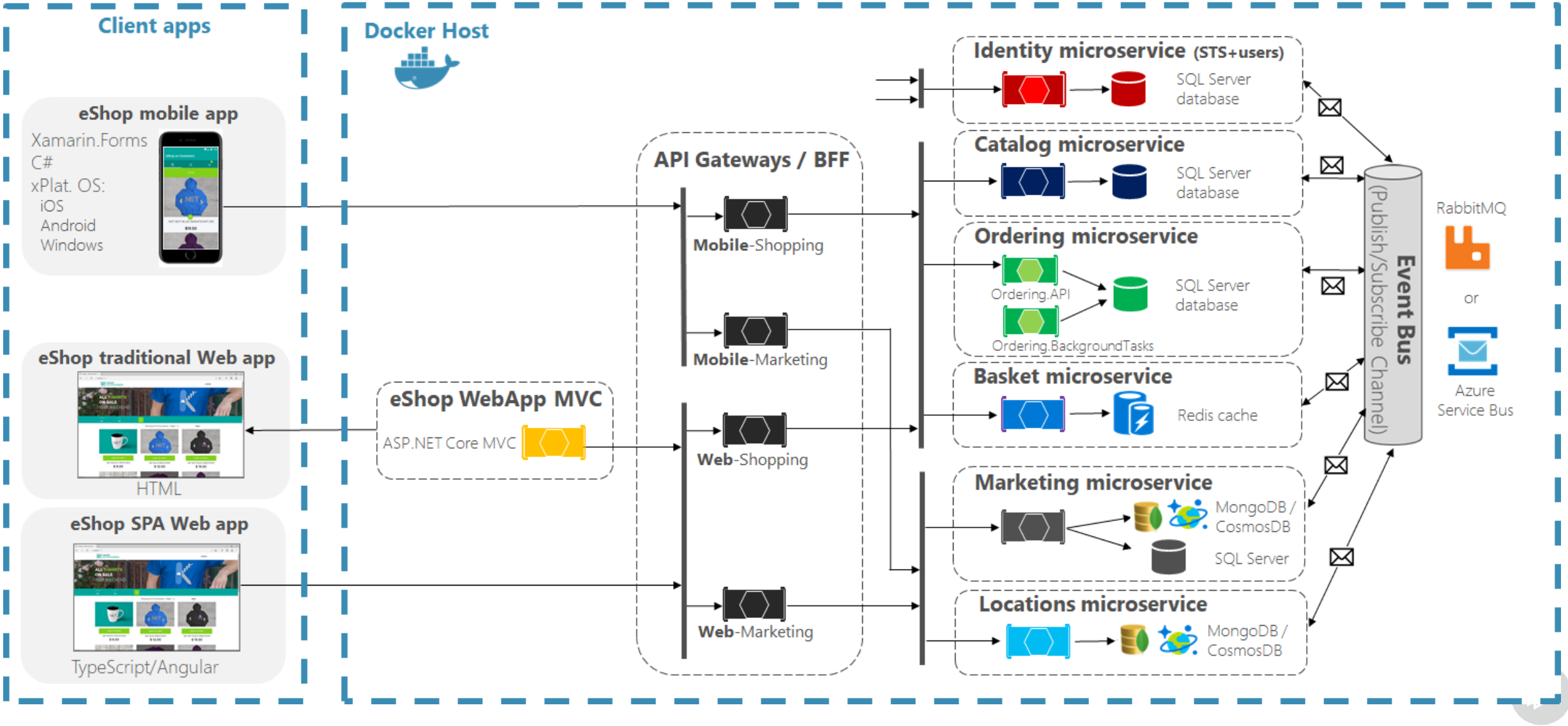
- ORMs
- Document databases
- CQRS

A microservices architecture can make use of multiple different domain logic and data access patterns

Each microservice should have a clearly-defined responsibility

Domain-Driven Design (DDD) can help us identify "bounded contexts"

eShopOnContainers Architecture



Microservice Code Responsibilities

Incoming Requests

HTTP requests

Messages

Domain Logic

Business rules

e.g. Sales tax

Data Access

Queries

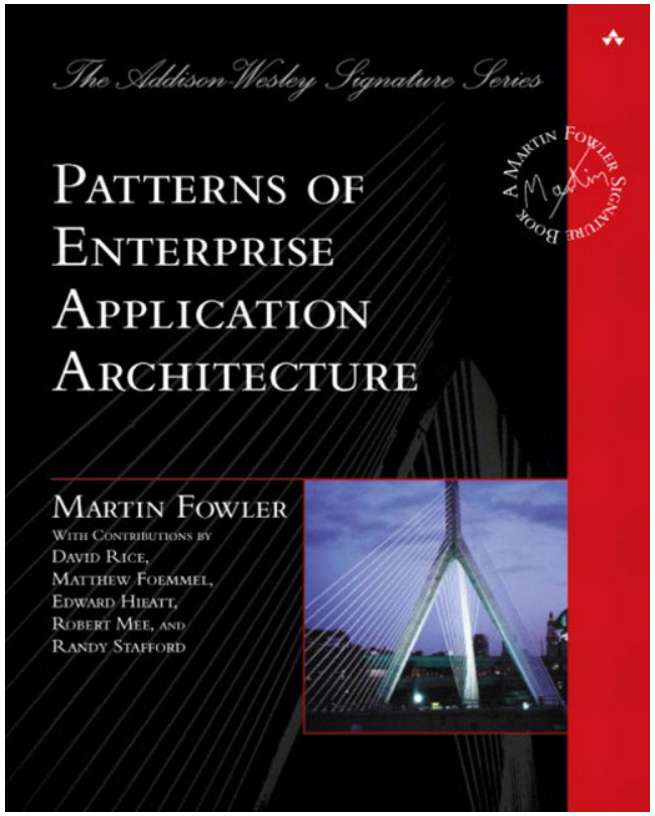
Updates

Integrate

Publishing messages

Third party services

Domain Logic Patterns



Patterns of Enterprise Application Architecture

Martin Fowler et al

Transaction Script

Domain Model

Table Module

Data Access Patterns

Relational Databases

ORM (Object relational mappers)

Hand-crafted SQL

Document Databases

Whole document operations

Limited support for joins

CQRS

Command query responsibility segregation

Separate read and update models

Event Sourcing

Store state changes as events

“Materialized views”

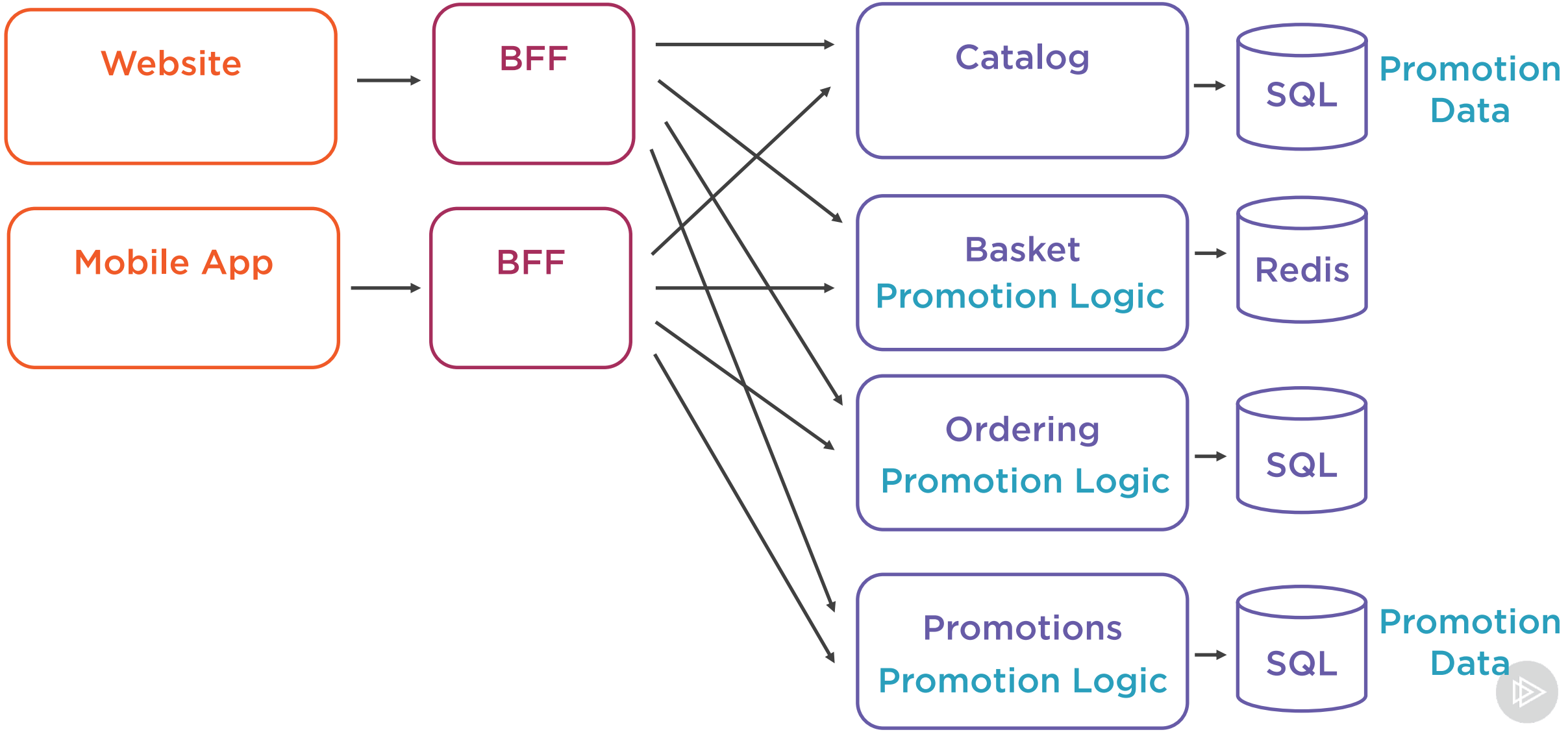
Eventual consistency

Apply the domain logic and data access patterns that make most sense for each individual microservice

Which microservice does my domain logic belong in?

Example: Promotions

Whose Responsibility Is It?



Choosing where to put the domain logic for a new feature is an important decision.

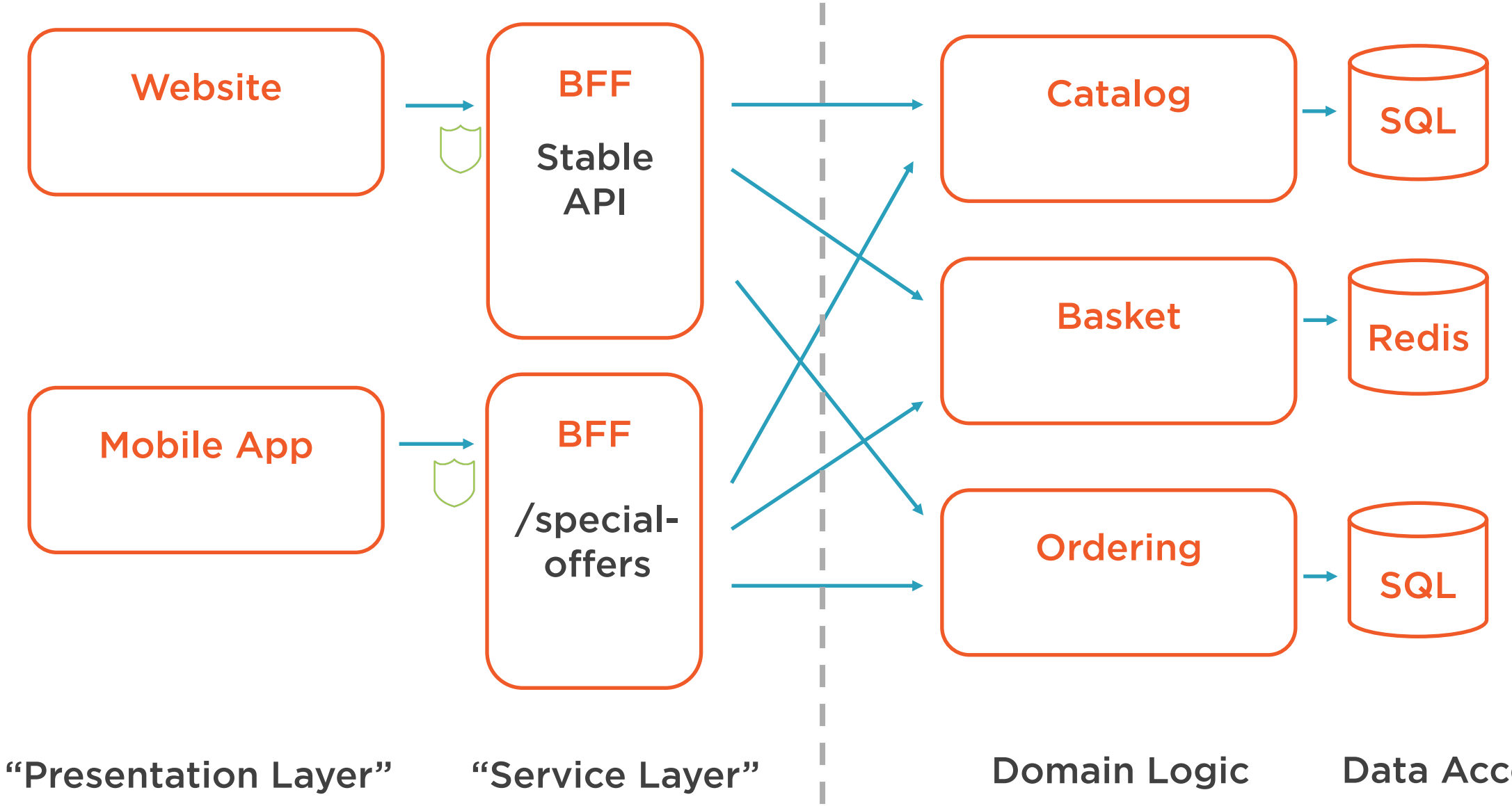
Dangers to Avoid

Duplicating business logic across multiple microservices

Giving too many responsibilities to a single microservice

Too much chatty communication between microservices

Service Layer



Domain Logic Patterns – Transaction Script

Transaction script

A single procedure handles each request from the presentation layer

Often calls directly to the database through a thin wrapper

Example: Add New Item to the Basket

1. **Fetch contents of user basket from data store** — Data Access
2. **Implement basket validation rules** — Domain Logic
   e.g. Basket cannot contain > 100 of the same item
3. **Persist basket contents to data store** — Data Access
4. **Publish BasketUpdated message to event bus**

 **Simple**

 **Easy to see the big picture**

Problems with Transaction Script

Growing complexity

Difficult to unit test

Duplication of domain logic

- Leads to divergent copies

Only suitable for simple microservices

- Otherwise leads to unmaintainable code

Demo

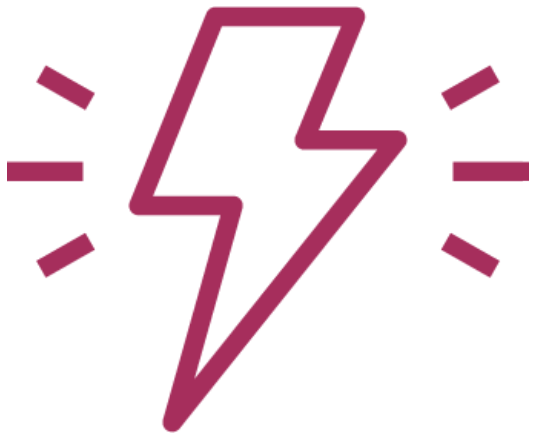
Transaction script approach to domain logic

- Basket microservice
- ASP.NET Core and C#

Repository pattern

Places a thin layer of abstraction around database access, simplifying code and unit testing

Serverless Domain Logic Patterns



Functions as a Service (FaaS)

- "Nano-services"
- Event-driven functions

Built-in integrations

- Reduced boilerplate

You only need to provide the domain logic

- Very few lines of code needed

Example Serverless Function

```
[FunctionName("GetTodoById")]
public static IActionResult GetTodoById(
    [HttpTrigger(AuthorizationLevel.Anonymous, "GET",
       Route = "/{id}")] HttpRequest req,
    [CosmosDB(DatabaseName, CollectionName,
        ConnectionStringSetting = "MyDB",
        Id = "{id}")] Todo todo, string id)
{
    return (todo == null) ?
     new NotFoundResult() : new OkObjectResult(todo);
}
```

https://github.com/markheath/funcs-todo-csharp

Domain Model

An object model of the domain that incorporates both behavior and data

Domain Model Versus DTOs

Data Transfer Objects (DTOs)

- Exposes public getters and setters
- Contain no business logic

Domain Model

- Expose public methods containing business logic
- Can't get into an invalid state

Example: Order Status

Domain models protect us from getting into an invalid state and from making invalid transitions between states

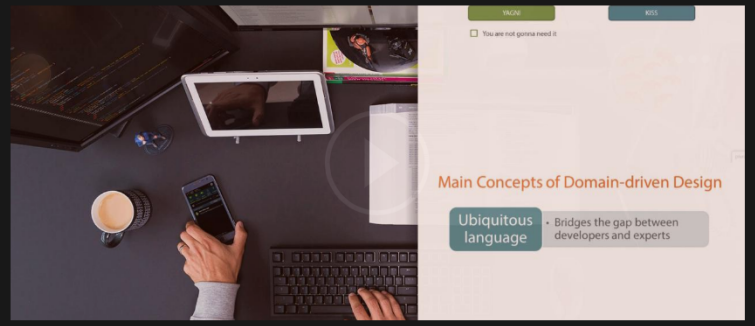
Demo

Domain Model pattern

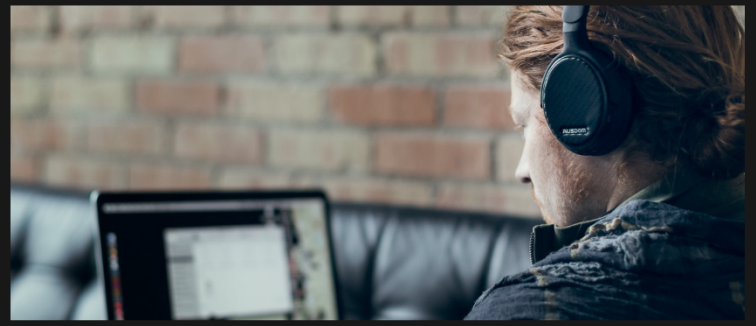
Ordering microservice

Domain models should use the language of the business

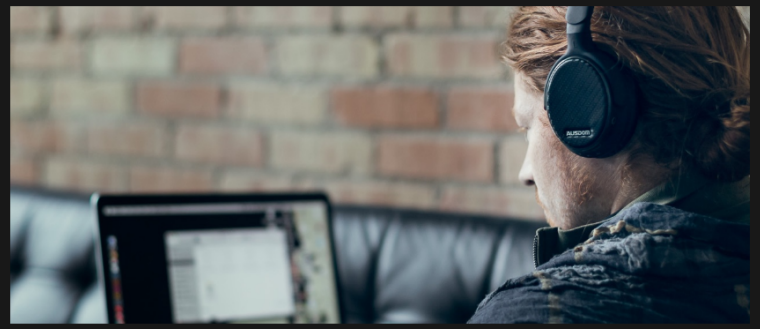
Learn More About Domain Driven Design



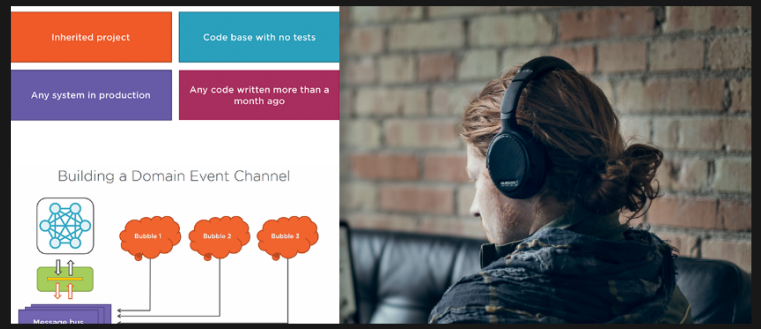
Domain-Driven Design in Practice
Vladimir Khorikov
Intermediate · 4h 19m · Sept 16, 2019



Refactoring from Anemic Domain Model Towards a Rich One
Vladimir Khorikov
Intermediate · 3h 36m · Nov 13, 2017



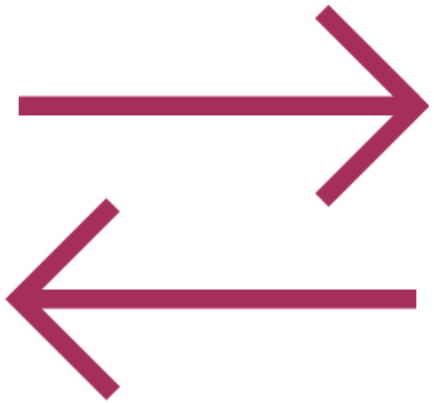
Specification Pattern in C#
Vladimir Khorikov
Intermediate · 1h 27m · June 27, 2017



Domain-Driven Design: Working with Legacy Projects
Vladimir Khorikov
Intermediate · 3h 51m · Mar 27, 2018

Domain model classes do not contain any data access code

Data Mapping Layer



Retrieves data from the database

Converts it into objects used by domain logic

- Domain models (e.g. Order class)
- Data transfer objects (DTOs)

Approaches to Data Mapping

Object Relational Mappers

Auto-generate SQL

Navigate relationships

Track modifications

e.g. Entity Framework Core

Simplifies data access

Potentially inefficient

ORMs can auto-load order items when we retrieve an order

Or fetch order items on demand when we need them

Approaches to Data Mapping

Object Relational Mappers

Auto-generate SQL

Navigate relationships

Track modifications

e.g. Entity Framework Core

Simplifies data access

Potentially inefficient

Micro ORMs

More control over SQL

Dynamic typing

More efficient

Document Databases

Rarely need to perform “joins”

e.g. store an order including order items in a single document

Simplifies reading and updating

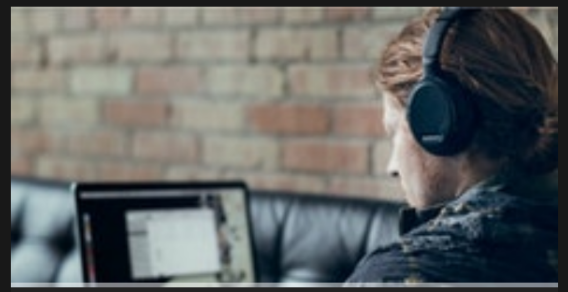
Serverless Frameworks

Data access via “bindings”

Less boilerplate code

Limited support for advanced scenarios

Learn More About Data Mapping



Entity Framework Core 2: Getting Started

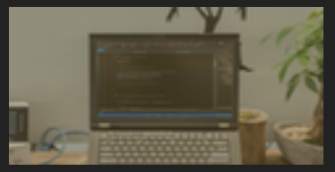
by Julie Lerman

Feb 6, 2018 / 2h 42m

Entity Framework Core 2: Mappings

by Julie Lerman

May 16, 2018 / 1h 59m

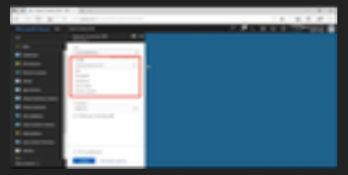


Dapper: Getting Started

by Steve Michelotti Beginner

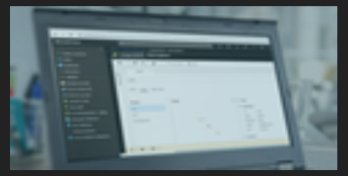
.NET Micro ORMs

by Steve Michelotti Intermediate



Learning Azure Cosmos DB

by Leonard Lobel Intermediate Sep 25 2019 6h 4m



Implementing NoSQL Databases in Microsoft Azure

by Reza Salehi Intermediate Sep 10 2019 3h 19m

Demo

Data mapping

Ordering microservice

Command Query Responsibility Segregation (CQRS)

Uses separate models for reads (queries) and updates (commands)

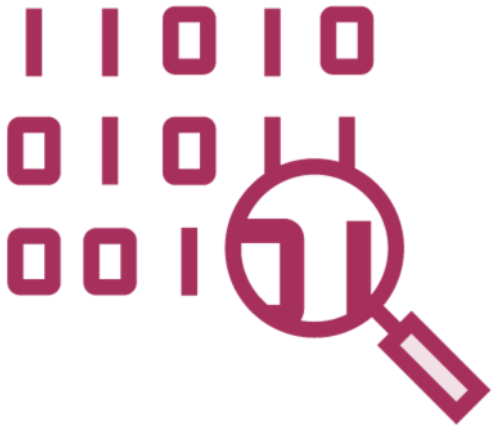
Commands should only update data and not return any data

Queries should only return data and not update any data

Avoid using the same models for database access and data transfer (DTOs)

The database schema and public API of your microservice should be able to change independently

Ordering Microservice Data Mapping



Keeps domain logic separate from data access

CQRS pattern

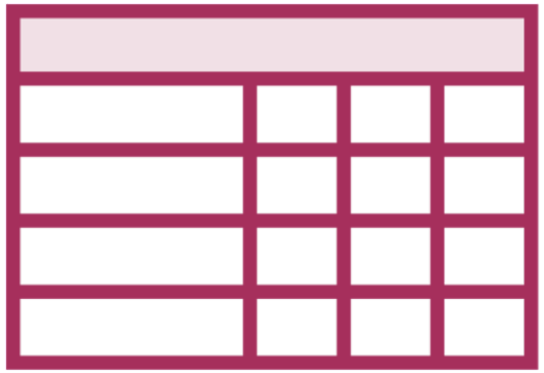
Commands

- ORM (Entity Framework Core)
- Repository pattern
- Unit of work pattern

Queries

- Raw SQL
- Used separate models

Table Module Domain Logic Pattern



Domain Model used a class per entity (database row)

- e.g. Order

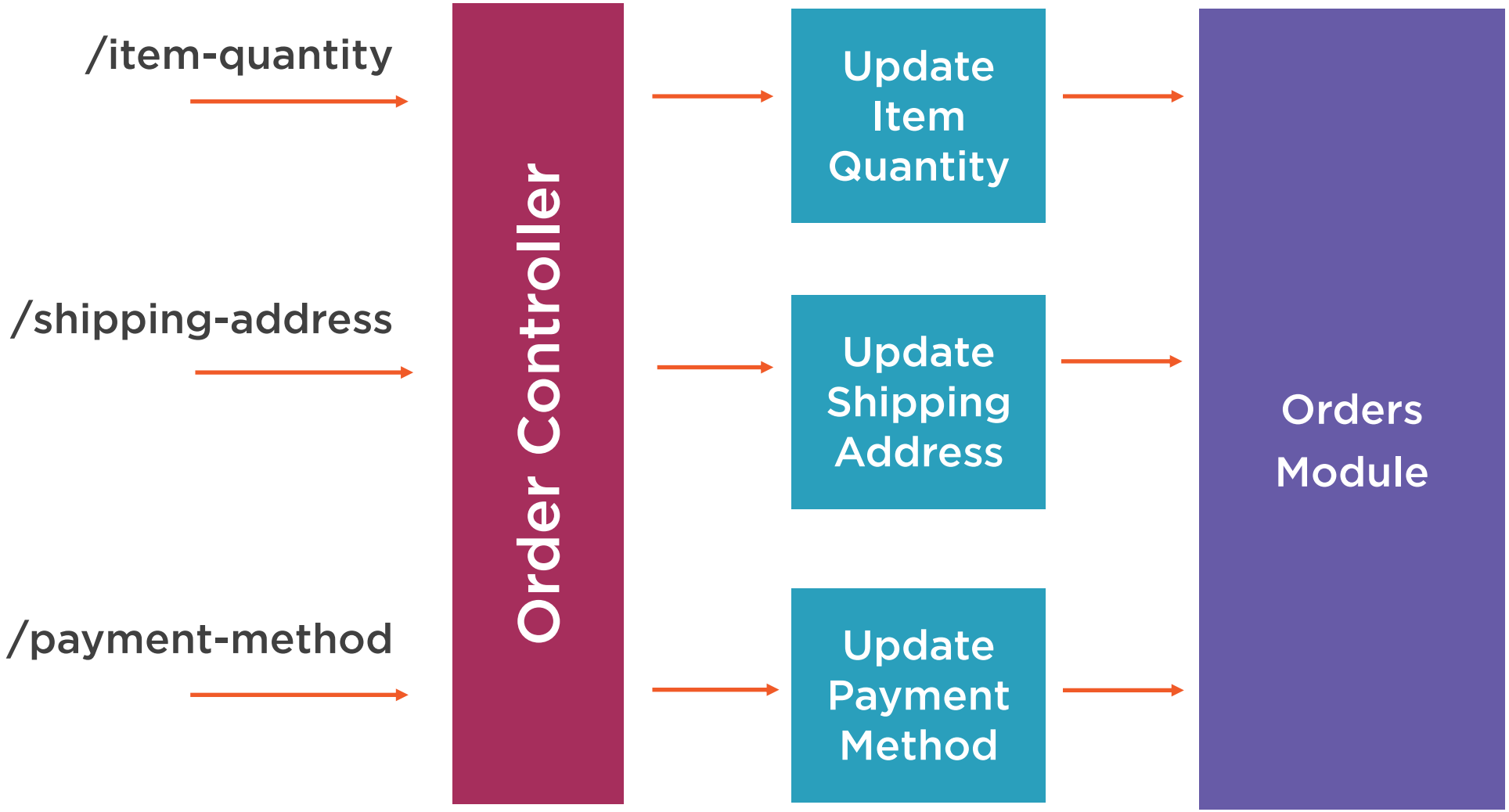
Table Module uses a class per table

- e.g. Orders
- Useful for batch operations
- Less commonly used

Transaction Script for simple domains

Domain Model for complex domains

Refactoring to Table Module



Summary

Choosing which microservice to own the domain logic

- Avoid duplication
- Avoid too many responsibilities
- A new microservice may be needed

Summary

Transaction Script

- Simple
- Potential for duplication

Domain Model

- Object-oriented
- Domain-Driven Design (DDD)

Table Module

- One class per table
- Helps eliminate duplication

Summary

Data mapping layer

- Keeps domain logic testable

Object relational mappers (ORMs)

- Good for relational databases

Micro-ORMs or raw SQL

- More control over performance

Document databases

- Easier to map to business objects

Serverless / Functions as a Service

- Built-in data access bindings

Summary

Command Query Responsibility Segregation (CQRS)

- Separate models for reads (queries) and updates (commands)

Up next...

Testing microservices