Setting up the Application Core

Gill Cleeren
CTO XPIRIT BELGIUM

@gillcleeren www.snowball.be

Overview

Understanding the business requirements

Setting up the solution

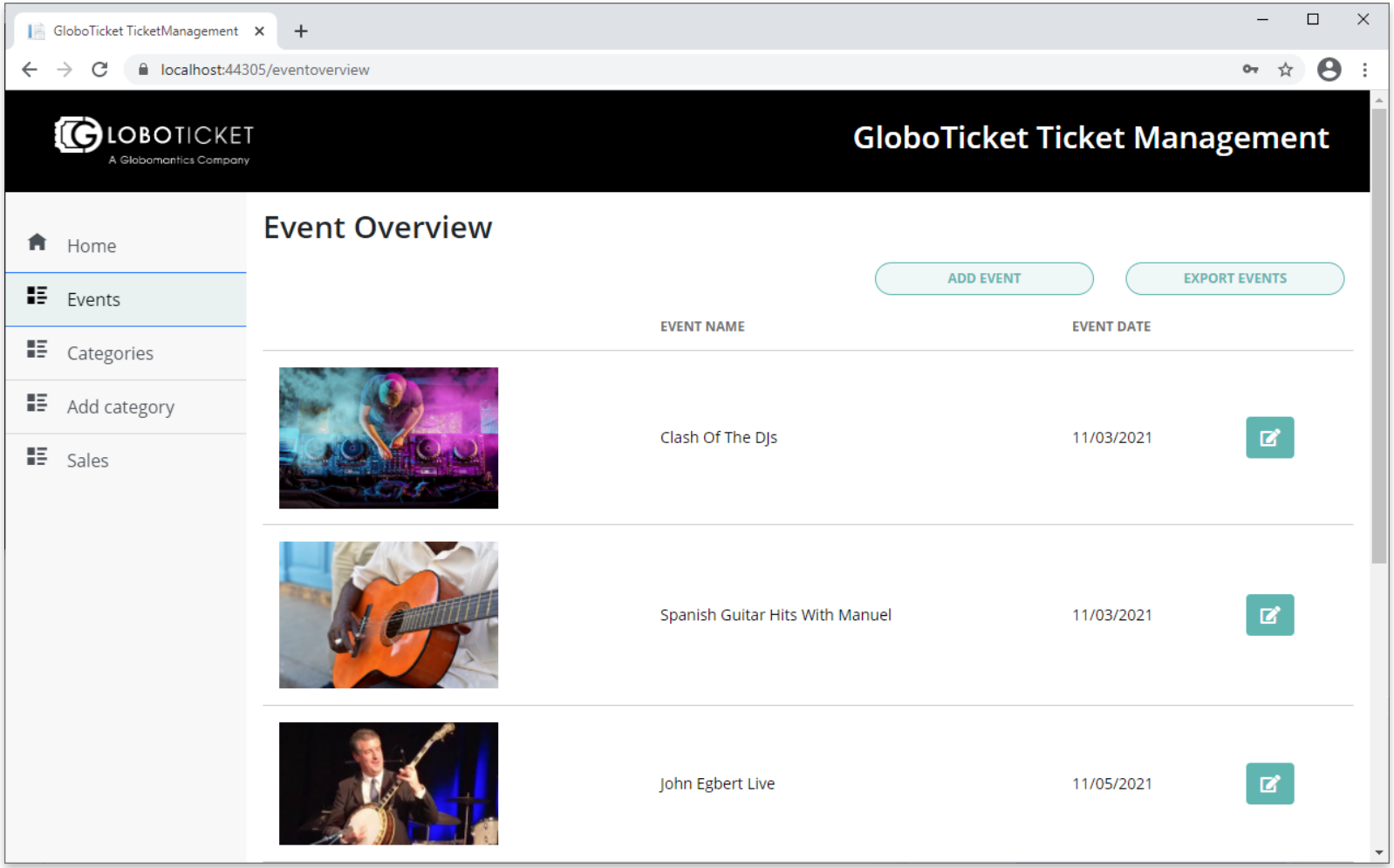
Creating the domain

Designing the application project

- Contracts
- Packages
- Validation
- Exceptions

GloboTicket Ticket Management

You've Seen The Finished Application



Understanding the Business Requirements

“Hello, I’m Mary Goodsale.

Let’s talk about the new system we need to build!”

Requirements for Ticket Management

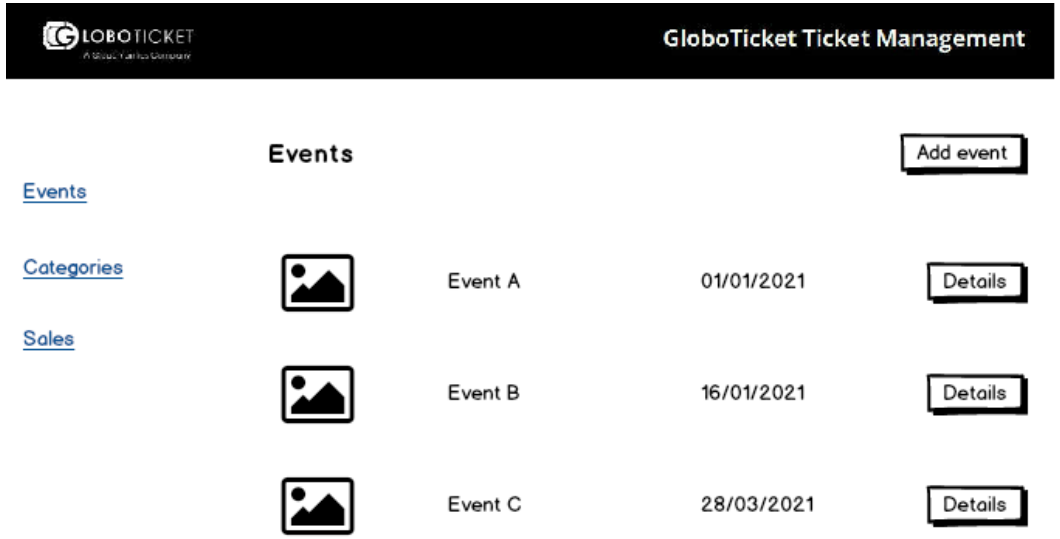
- Manage events
- Overview of events in their categories
- Orders for the different events

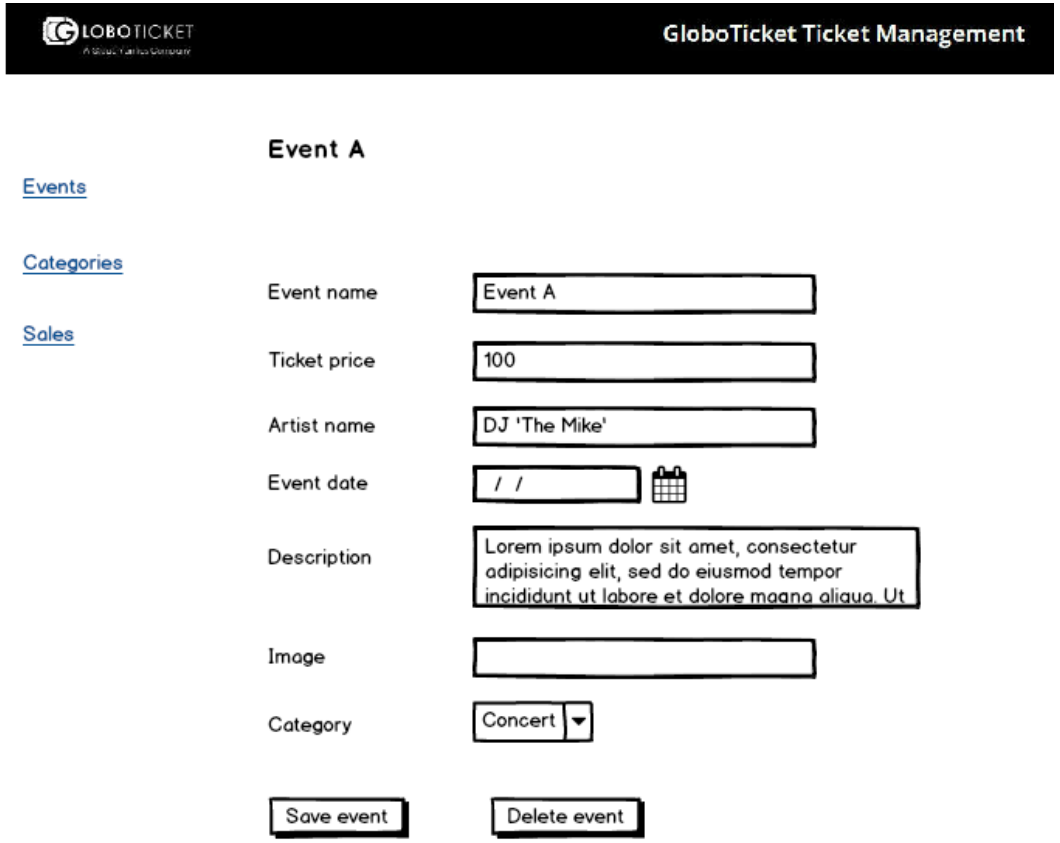
Requirements for Ticket Management

- Manage events
- Overview of events in their categories
- Orders for the different events

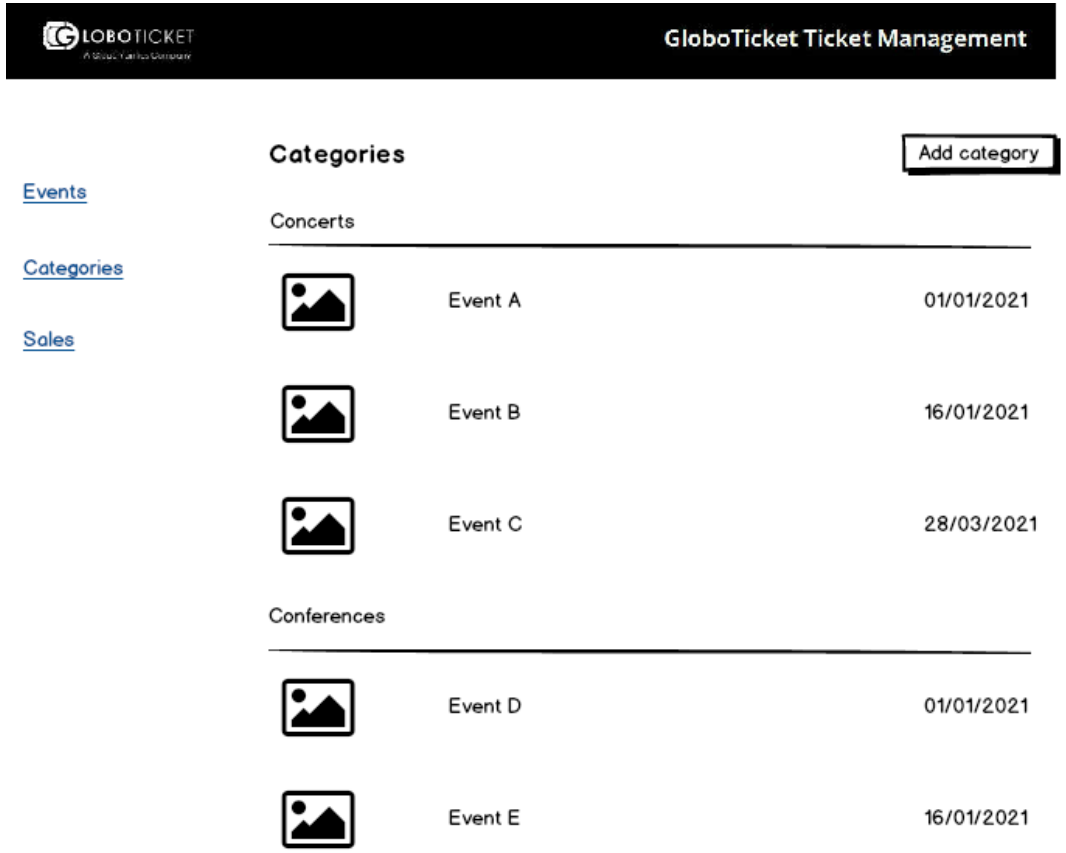
Wireframes for the Application

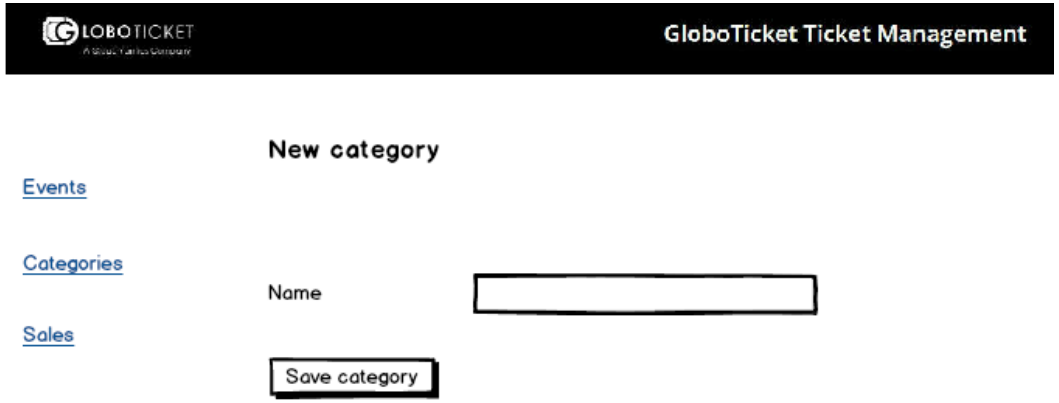
Event Management



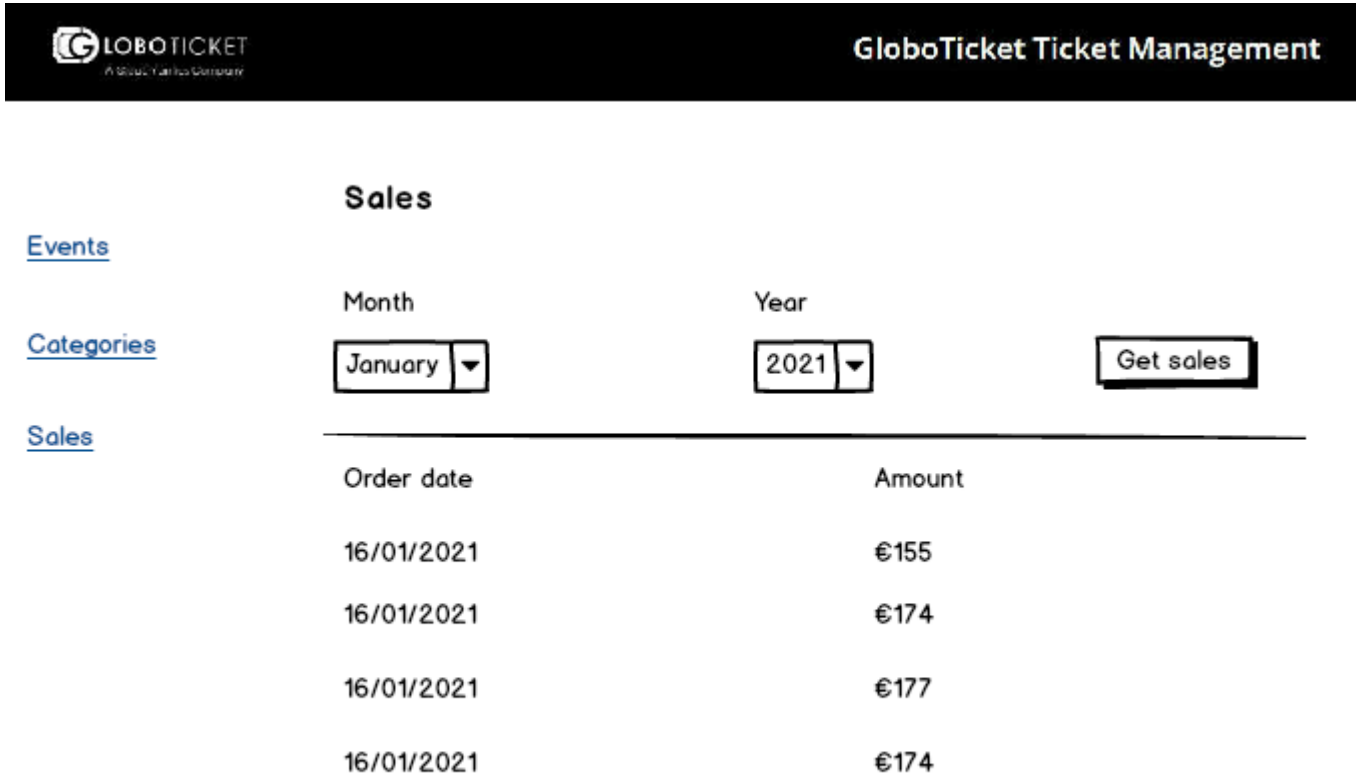


Category Management



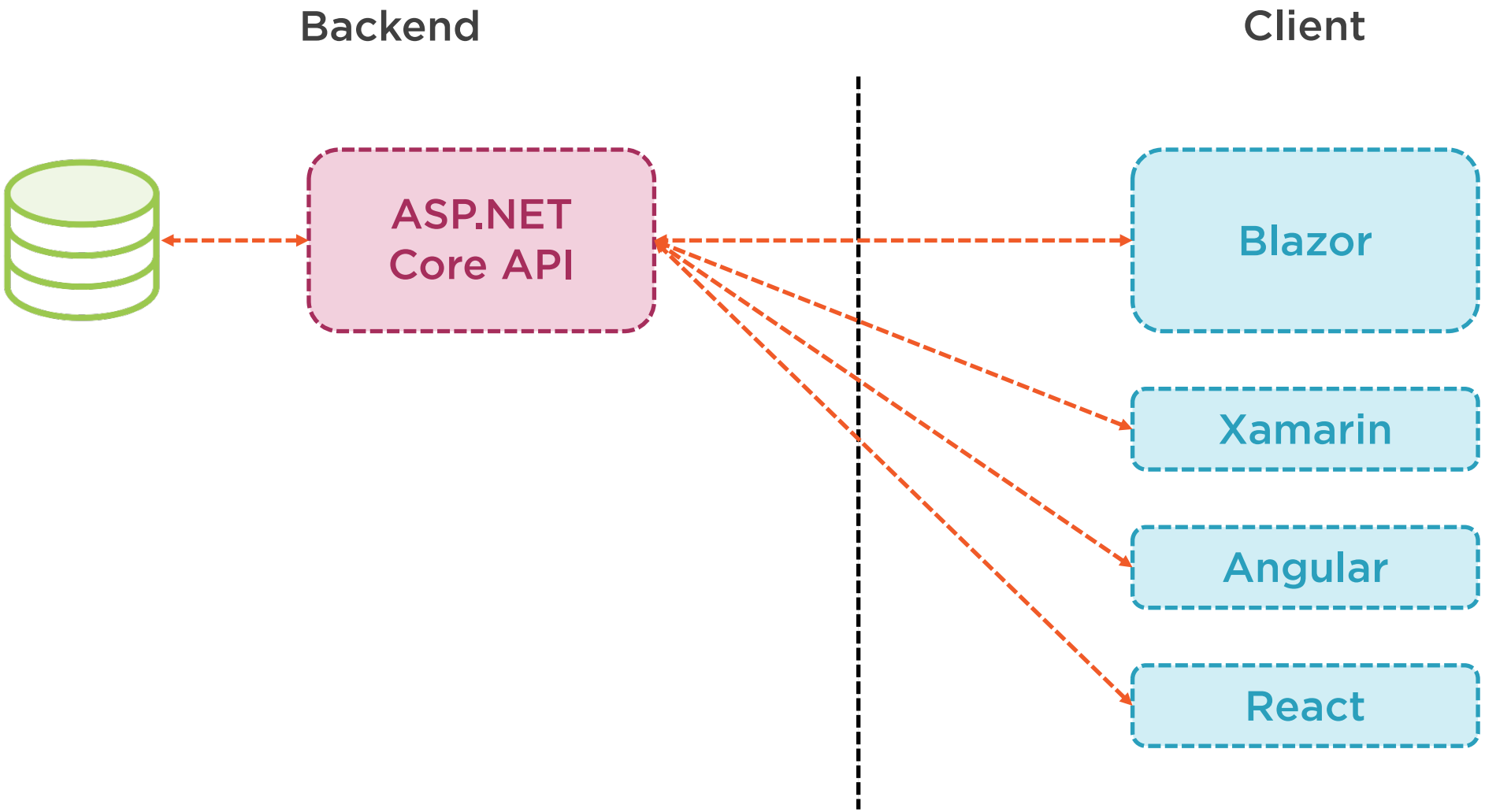


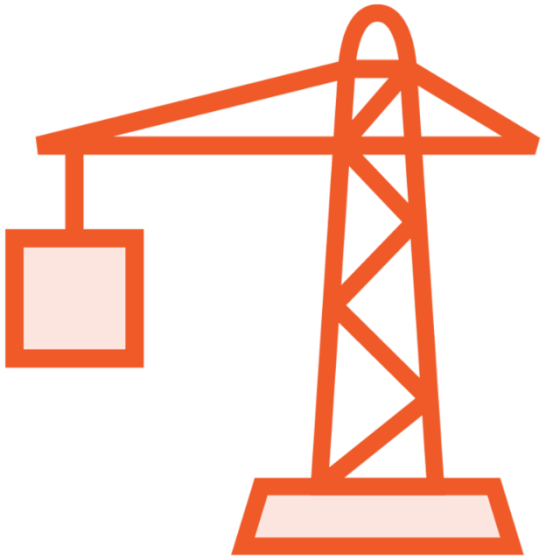
Ticket Sales



Setting up the Application Architecture

Translating to an Application Architecture





REST API

- Built using ASP.NET Core
- Clean architecture principles
- Data access using EF Core

Client

- Built using Blazor WebAssembly

Class libraries

- .NET Standard

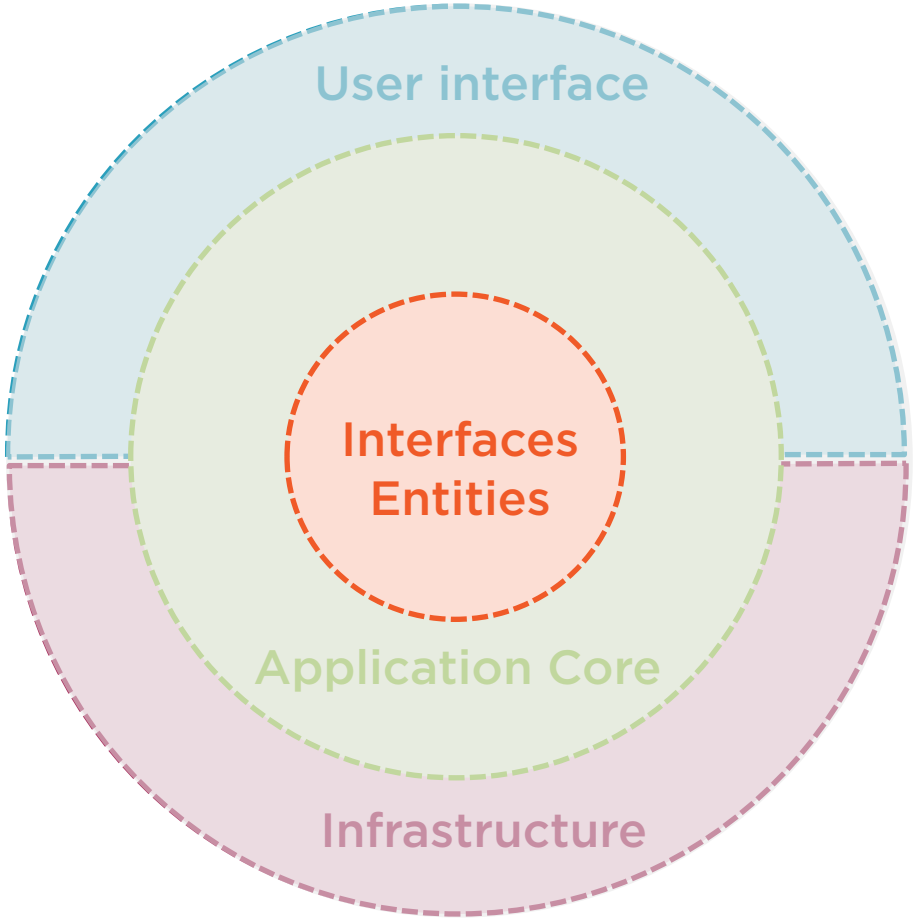
All independent of .NET version

Demo

Creating the Visual Studio solution

Creating the Domain Project

The Domain Project

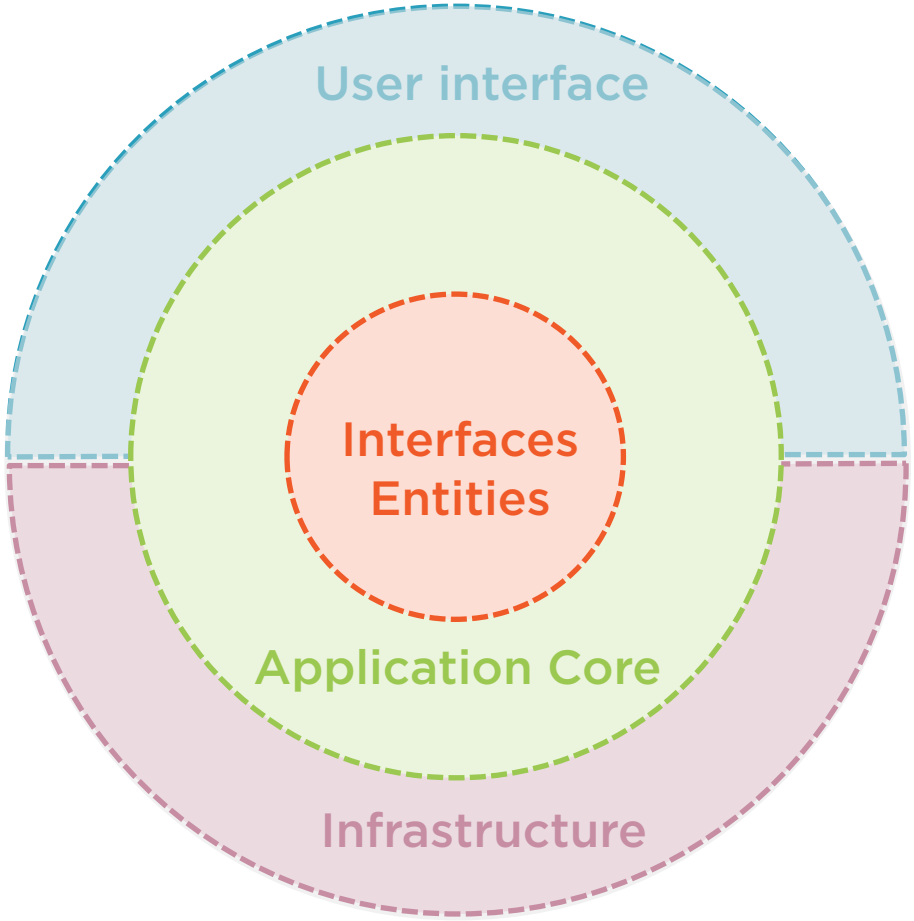


Demo

Creating the domain project

Designing the Application Project

The Application Project



Achieving loose coupling in the application core

- Contracts
- Messaging

Contracts

- Part of application core
- Functionality is described in interfaces
- Implemented in Core or Infrastructure

Using Repositories

Mediates between domain and data-mapping layer

Often used in combination with UOW

Using Repositories

Data access operations

Agnostic for rest of application

Generic methods

Specific repositories

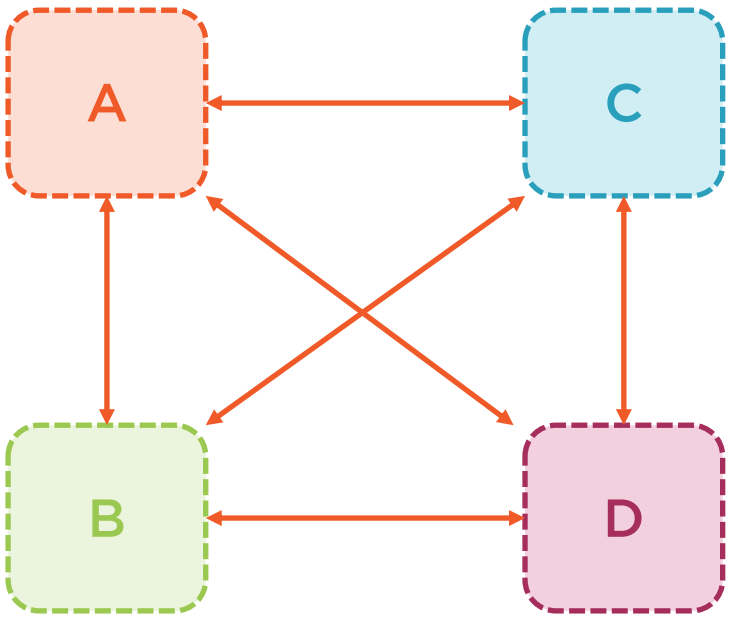
Demo

Creating the Application project

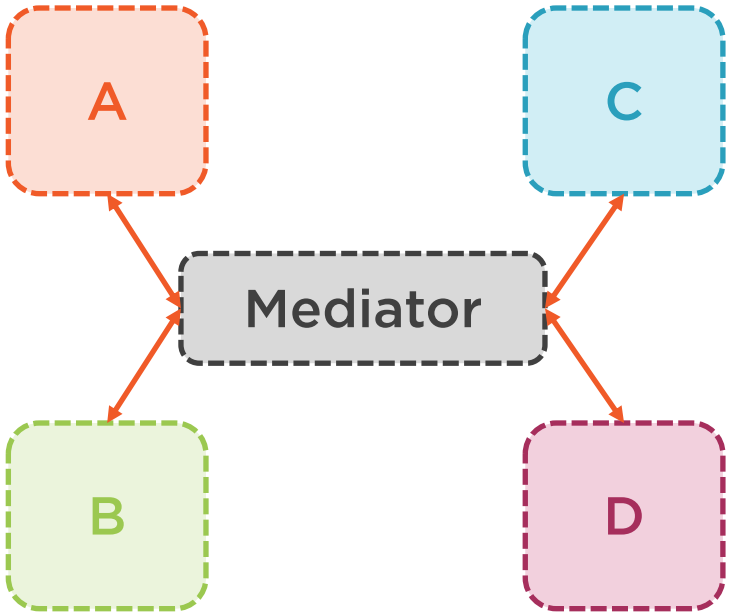
Adding contracts

Introducing the foundation for the repository

Introducing a Mediator



Introducing a Mediator



Object that wraps what how objects need to interact

Avoid hard references from one object to the next

Help with communication from/to Core project objects

Advantages of Using a Mediator

Changes can be handled easily

Easy to test the object

Using MediatR

Simple mediator implementation in .NET

- github.com/jbogard/MediatR

Adding MediatR

- Install-Package MediatR
- services.AddMediatR(Assembly.GetExecutingAssembly());

Using

- IRequest
- IRequestHandler

```
public class GetEventsListQuery: IRequest<List<EventListVm>>
{

}
```

Creating a Request

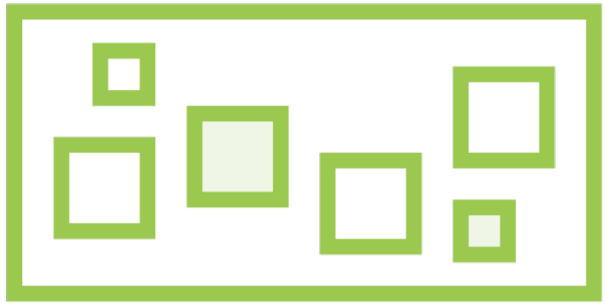
```
public class GetEventsListQueryHandler :
    IRequestHandler<GetEventsListQuery, List<EventListVm>>
{

    public async Task<List<EventListVm>> Handle
        (GetEventsListQuery request, CancellationToken cancellationToken)
    { }

}
```

Defining the Request Handler

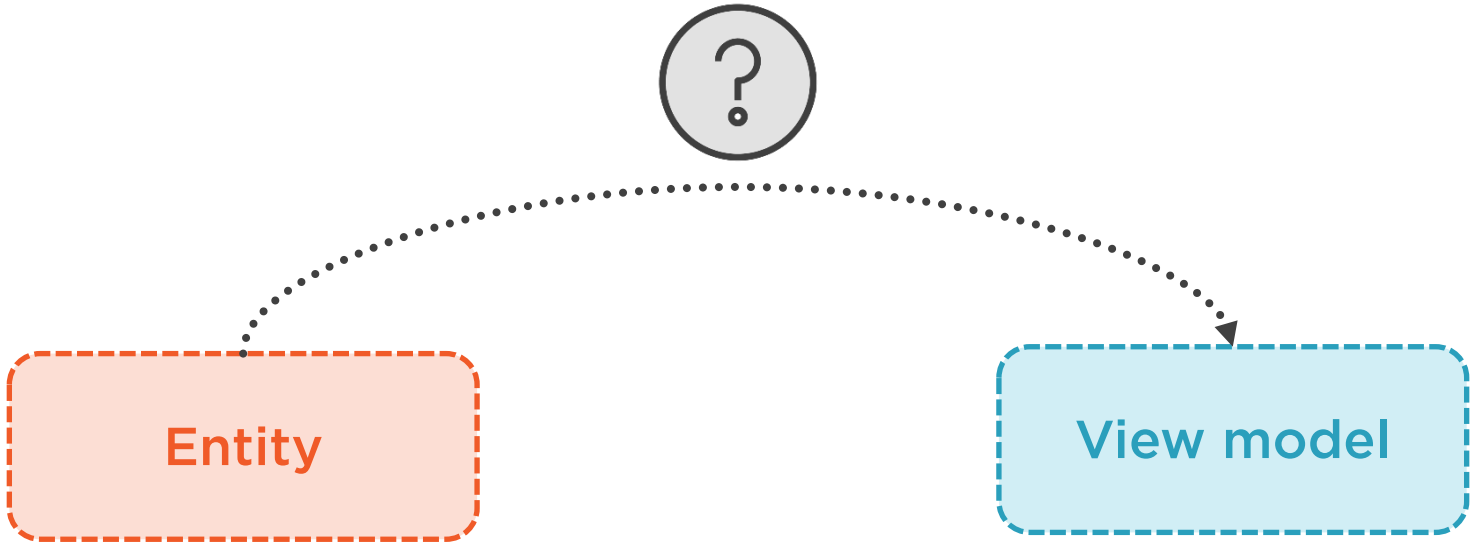
What We Aren't Using of MediatR



Pipeline behaviour

- Logging
- Validation
- Caching

Mapping from an Entity to a View Model



Mapping Between Objects

AutoMapper library

Mapping from one type to another type

Using AutoMapper

NuGet package

Startup registration in DI

Profile

```
var result = _mapper.Map<List<EventListVm>>(allEvents);
```

Mapping with AutoMapper

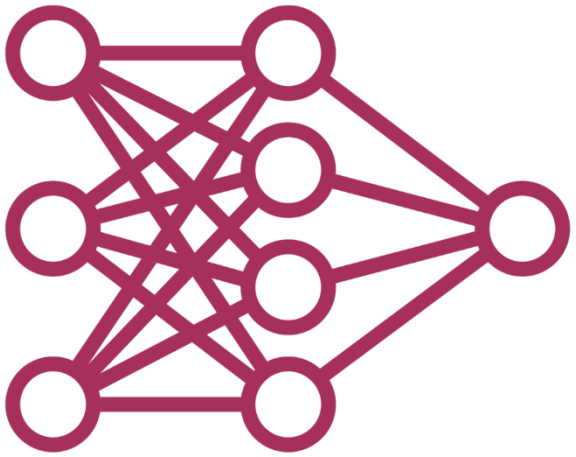
Demo

Introducing MediatR and AutoMapper

Creating a request and request handler

Adding a ServiceCollection extension class

Reading and Writing Data



Same model is used to read and write data

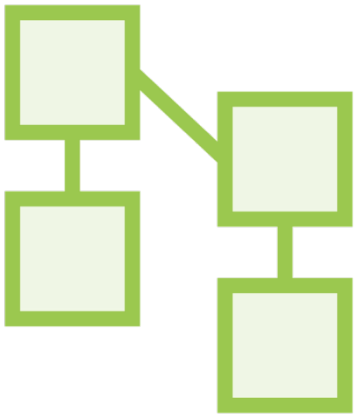
Issues in larger applications

- Different queries
- Different object being returned
- Complex logic for saving entities
- Security may be different

Model may become too heavy

Adding Simple CQRS

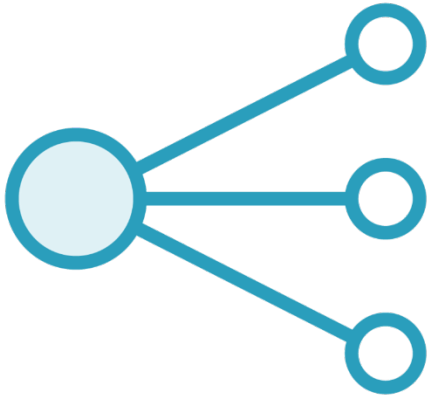
Command-Query Responsibility Segregation



Different models

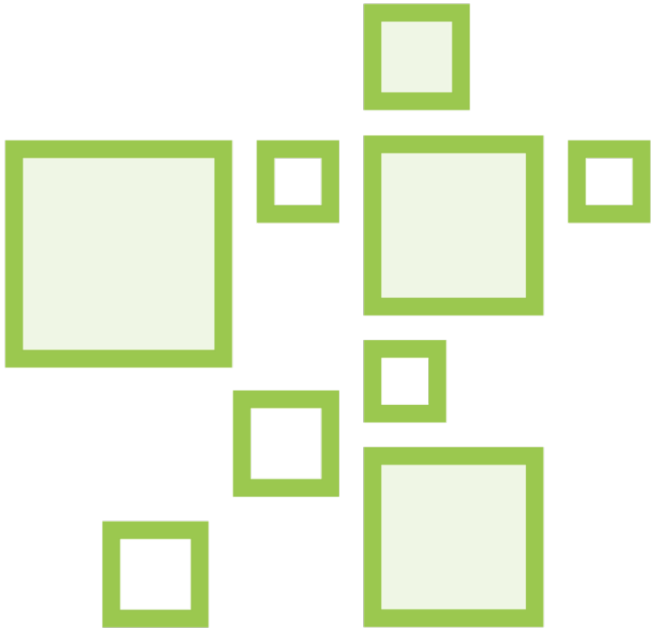
Commands to update data

Queries to read data



Commands are task-based

Can be asynchronous



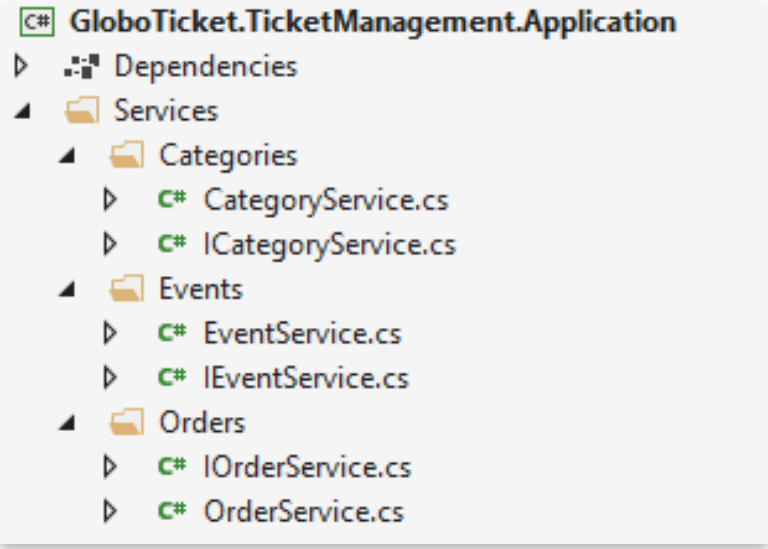
Advantages

- Separation of concerns
- Scaling
- Security
- Easy to make a change, no further impact

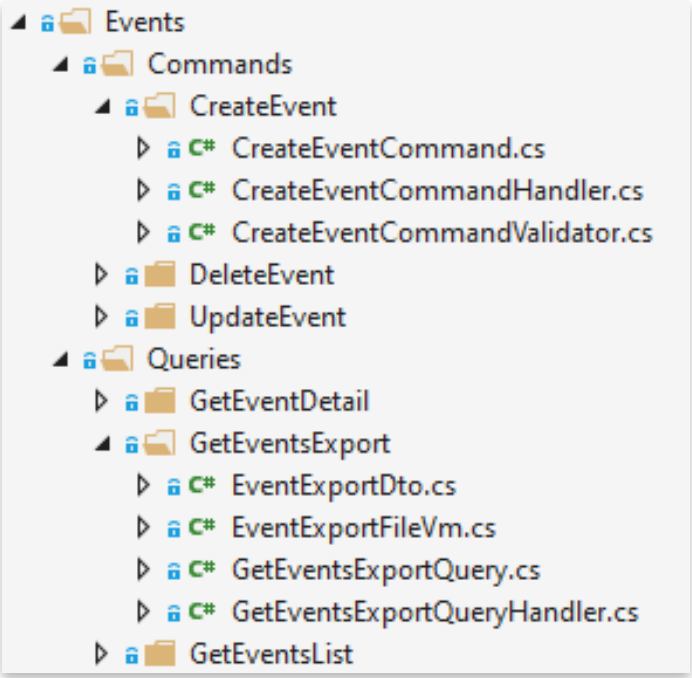
Disadvantages

- Added complexity
- Targeted at more complex applications

What Problem Are We Solving?



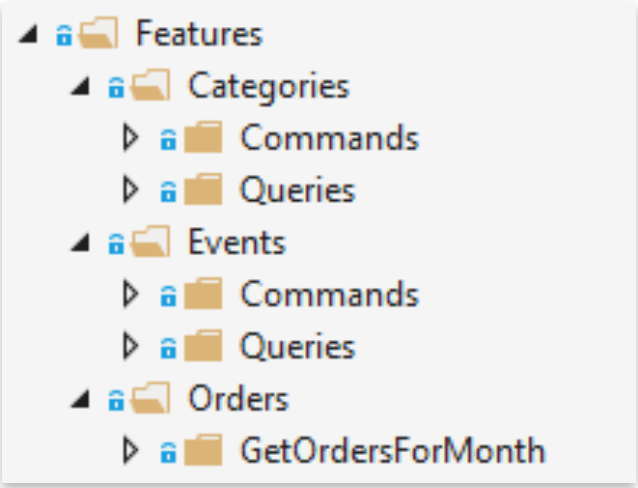
Translating Our Requirements to CQRS



Demo

Adding simple CQRS

Feature-based Approach



Vertical slice

Features folder and subfolders

Own the view models they will use

- Not shared typically, even if identical

Demo

Splitting up in features

Creating a New Entity

Demo

Creating a new Event

Updating and deleting an Event

Validation

Event.cs

```
public class Event: AuditableEntity
{

    public Guid EventId { get; set; }

    [Required]
    [StringLength(50)]
    public string Name { get; set; }

    public int Price { get; set; }

    [Required]
    [StringLength(50)]
    public string Artist { get; set; }

}
```

Adding Fluent Validation

- Nuget package
- Uses lambda expressions for validation rules

Can be used in Core project

- RequestHandler
- Part of the Feature folder

```
public class CreateCategoryCommandValidator:
    AbstractValidator<CreateCategoryCommand>
{
    public CreateCategoryCommandValidator()
    {

        RuleFor(p => p.Name)
            .NotEmpty().WithMessage("{PropertyName} is required.");
    }
}
```

Adding a Custom Validator

```
var validator = new CreateCategoryCommandValidator();

var validationResult = await validator.ValidateAsync(request);
```

Using the Validator

Returning Exceptions

Core should return own set of exceptions

- Can be handled or transformed by consumer

Used exceptions

- NotFoundException
- BadRequestException
- ValidationException

```
public class NotFoundException : ApplicationException
{

    public NotFoundException(string name)
        : base($"{name} is not found")

    {  }

}
```

Adding Custom Exceptions

Demo

Validating data input

Using custom exceptions

Returning a response object

Summary

Core contains the core functionality of the application

CQRS and MediatR help with achieving high-level of loose-coupling

Feature-based helps with organization of code

Validation using Fluent Validation

Implementations not included!

Don’t include concrete implementations for anything infrastructure-related.

Up next:

Bringing in the application infrastructure