FUTURE DECODED

6-7 OTT '16 / MILANO

IN PARTNERSHIP WITH: CommunityDays.it

www.futuredecoded.it

#FutureDecoded

DEV01

Messaging con Rebus.net: da 0 a CQRS in 60 minuti

Andrea Saltarello
Solution Architect @ Managed Designs S.r.l. | Presidente UGIdotNET
Microsoft Regional Director
@andysal74

www.futuredecoded.it
#FutureDecoded

CommunityDays.it

Me.About();

- Solution Architect @ Managed Designs
- Microsoft MVP since 2003
- Microsoft Regional Director
- Author (along with Dino) of .NET: Architecting Applications for the Enterprise, by Microsoft Press
- Basically, a ~~software architect~~ developer eager to write code ☺

Demos.About();

All demos (but one) are based on Merp, a GPL'ed Micro ERP I developed (and still do, to an extent) as the companion project for my book.

It's "free as in speech", so:

1. Download & unzip it
2. Open the .sln file in Visual Studio
3. Run Update-Database (x2)
4. Enjoy time travelling ☺

Newer builds will be published here:
http://www.github.com/mastreeno

The (Relational) Lord of the Rings

It really became clear to me in the last couple of years that we need a new building block and that is the Domain Event.

[Eric Evans]

An event is something that has happened in the past.

[Greg Young]

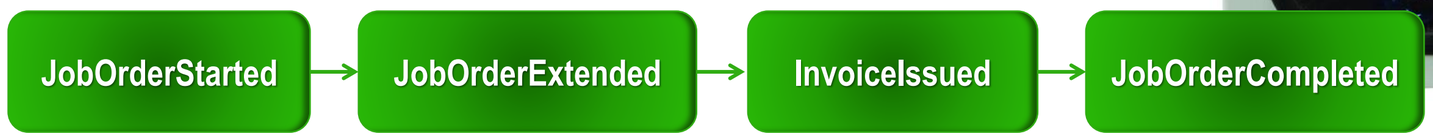
A domain event … captures the memory of something interesting which affects the domain

[Martin Fowler]

Event Sourcing in a nutshell

Instead of focusing on a system's last known state, we might note down every occurring event: this way, we would be able to (re)build the state the system was in at any point in time just replaying those events

To cut a long story short: we'd end up recording an **event stream**



What's an event, anyway?

The (immutable) composition of:

- A (meaningful) name
- (Typed) Attributes

ProjectCompleted
DateOfCompletion
ProjectId

InvoiceIssued
DateOfIssue
Customer
Price

ProjectStarted
DateOfStart
ProjectId

ProjectRegistered
DateOfRegistration
DateOfEstimatedCompletion
ProjectId
CustomerId
Price

DEMO

Event Stream

Event Stream vs. «My application»

Still, my users are more interested in knowing a job order's balance or whether an invoice has been paid. (cit.)

That is, we need a way to produce an entity state

Event Sourcing <3 DDD

DDD's Aggregates provide a convenient way to encapsulate event management

***Aggregate**: A collection of objects that are bound together by a root entity, otherwise known as an aggregate root. The aggregate root guarantees the consistency of changes being made within the aggregate.*

[Wikipedia]

An aggregate is responsible for:

- encapsulating business logic pertaining to an "entity"
- generating events to have them available for saving
- replaying events in order to rebuild a specific state

DEMO

Aggregates

Aggregates vs. Events vs. Repos

```
var aggr = repository.GetById<TAggr>(id);   //Triggers [time travelling] event replay
aggr.DoSomething();                         //Biz logic + events
repository.Save(aggr);                      //Updates the event stream
```

Repository: *Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.* [DDD]

DEMO

Time Travelling

Event Stream vs. «My application»

Still², my users are more interested in knowing a job order's balance or whether an invoice has been paid. Quickly.

Ways to achieve that:

- Snapshots can help
- CQRS to the rescue: let's have a database storing the usual «last known system state» using it as a *read model*

Enter CQRS

Acronym for **C**ommand **Q**uery **R**esponsibility **S**egregation

Basically, ad hoc application stacks for either “writing” or reading:

- “Command” stack writes events and snapshots
- “Read” stack reads from eventually consistent, reading purposes optimized database(s)

CQRS: the “Read” side of the Force

As a business unit manager, I want to collect credits due to unpaid outgoing invoices #ubiquitouslanguage #nuffsaid

CQRS/ES wise, this user story could be implemented by means of the following real world C# code:

```
Database.OutgoingInvoices.
        .PerBusinessUnit(businessUnitId)
        .ExpiredOnly()
        .Select(i => new {InvoiceNumber = i.Number, CustomerId = i.Customer.Id})
        .AsParallel()
        .ForAll(i => bus.Send(new CollectDebtCommand(i.InvoiceNumber, i.CustomerId)));
```

DEMO

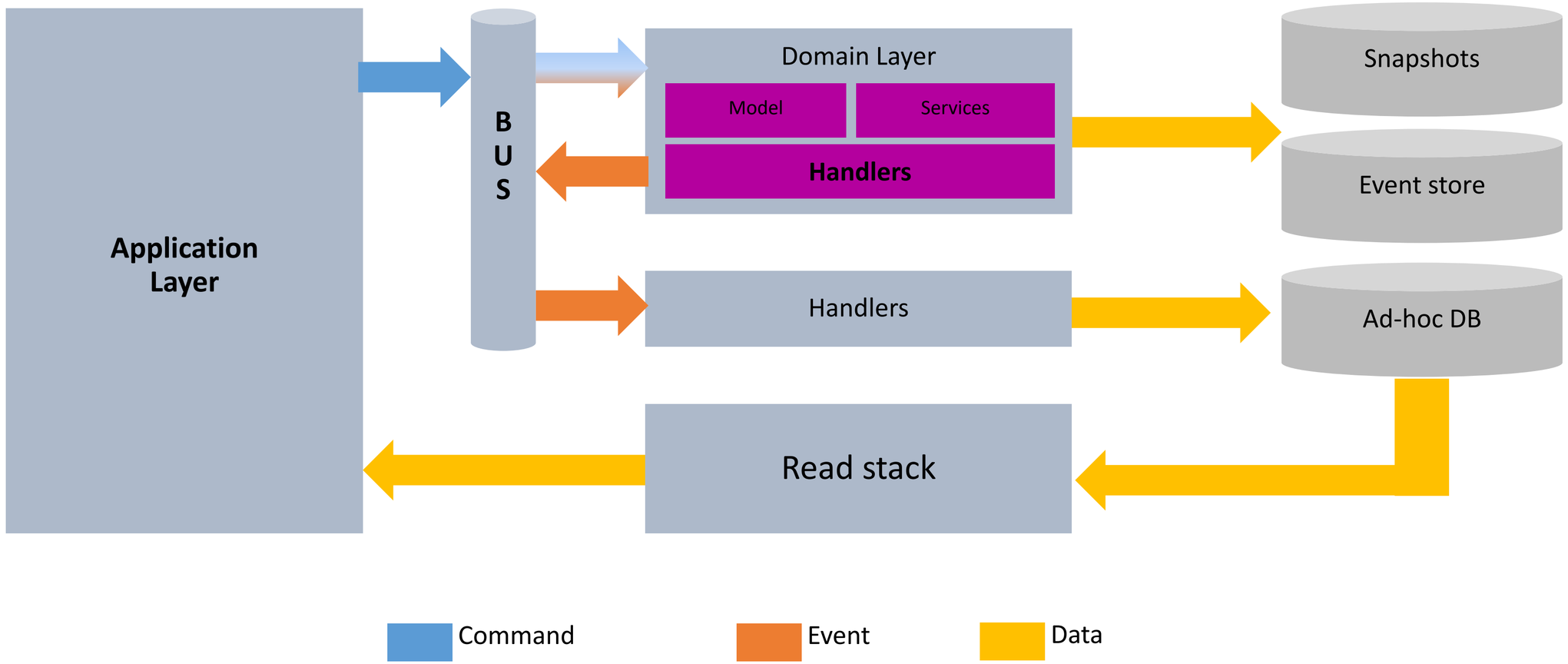
Read Model

CQRS/ES in a nutshell

1. Application sends a command to the system
2. Command execution might alter the system's state and then raise events to state success/failure
3. Events are notified to interested subscribers (a.k.a. handlers), such as:
   - Workflow managers (a.k.a. «Sagas») which could execute more commands
   - Denormalizers, which will update the read model database

Note: command/event dispatch/execution will usually be managed by a Mediator («bus»)

CQRS/ES at a glance



DEMO

Handlers

Enter Rebus.net

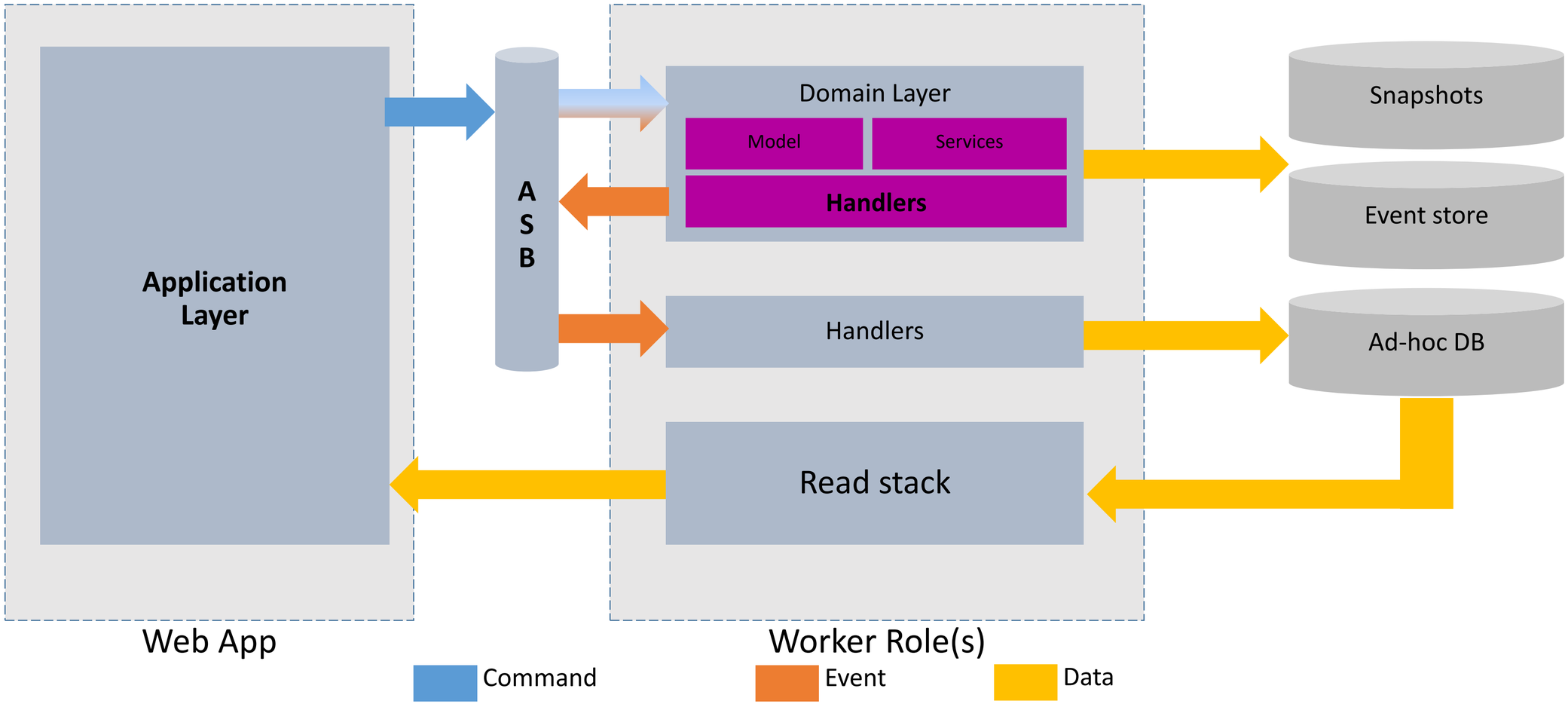
Rebus.net:

- is an open source, production tested bus for .NET. Source code is available on Github, but usually Nuget packages will do.
- provides much-needed features such as:
  - Fault tolerance
  - Scalability
  - Events' scheduling
  - Easy migration from on premises to cloud based
- Requires:
  - a transport protocol (e.g.: MSMQ, RabbitMQ, Azure Service Bus, Amazon Sqs, ...)
  - an IoC container (e.g.: Autofac, Ninject, .NET Core ServiceProvider, StructureMap, Unity, Windsor...)
  - a storage (e.g.: MongoDB, RavenDb, SQL Server, ...)

DEMO

Configuration
Back to the Future

Azure <3 CQRS


Application Layer
A S B
Domain Layer
Model
Services
Handlers
Handlers
Read stack
Snapshots
Event store
Ad-hoc DB
Web App
Worker Role(s)
Command
Event
Data

Look ma, a microservice!

Both web App(s) and worker role(s) can be:

- Evolved
- Deployed
- Configured (e.g.: scaled)

independently

GRAZIE!

www.futuredecoded.it

#FutureDecoded