# CQS Performance & Simplicity

Sean Rogers

September 2014

## CQS Introduction & overview

- **What** is it?
- **Why** choose it?
- In the context of **SOLID**
- In the context of **DDD**
- Architectural **overview**
- Performance and simplicity **via architecture**
- Performance and simplicity **via complimentary frameworks #1**
- Performance and simplicity **via complimentary frameworks #2**
- Why is performance **so important**?
- Let’s look at **some code**
- Our **query performance** benchmarks
- **Conclusions** and **Questions**

## What is it?

**Command Query Separation**

- A programming **philosophy** and also an architectural **pattern**

“Asking a question should not change the answer

Bertrand Meyer 1990 - Object Oriented Software Construction

**Queries**

- Return a result and do not change the observable state of the system (are free of side effects so are repeatable)

**Commands**

- Change the state of a system but generally do not return a value

**Architecturally**

- One channel for reads and a completely separate channel for writes

## Why choose it?

- **High** Performance and **low** maintenance
- A pattern that acknowledges the true behaviour of the web: **80**% reads **20**% writes
- **Reduces layers** of architecture and **removes inefficiencies** – for example the expensive and unnecessary mapping that is ultimately thrown away
- **Enforces** SRP and Interface Segregation
- Produces **concise** and **self-descriptive** code
- **Simplifies** the domain model as there are no compromises needed to satisfy both read and write

## CQS and CQRS

- CQS a **low level** programming philosophy and also a **high level** architectural pattern
- CQRS an **architectural pattern** that was designed to solve one specific problem : The collaborative domain
- CQS can port CQRS easily due to the **separation** of channels and the one way nature of command handlers

# In the context of SOLID

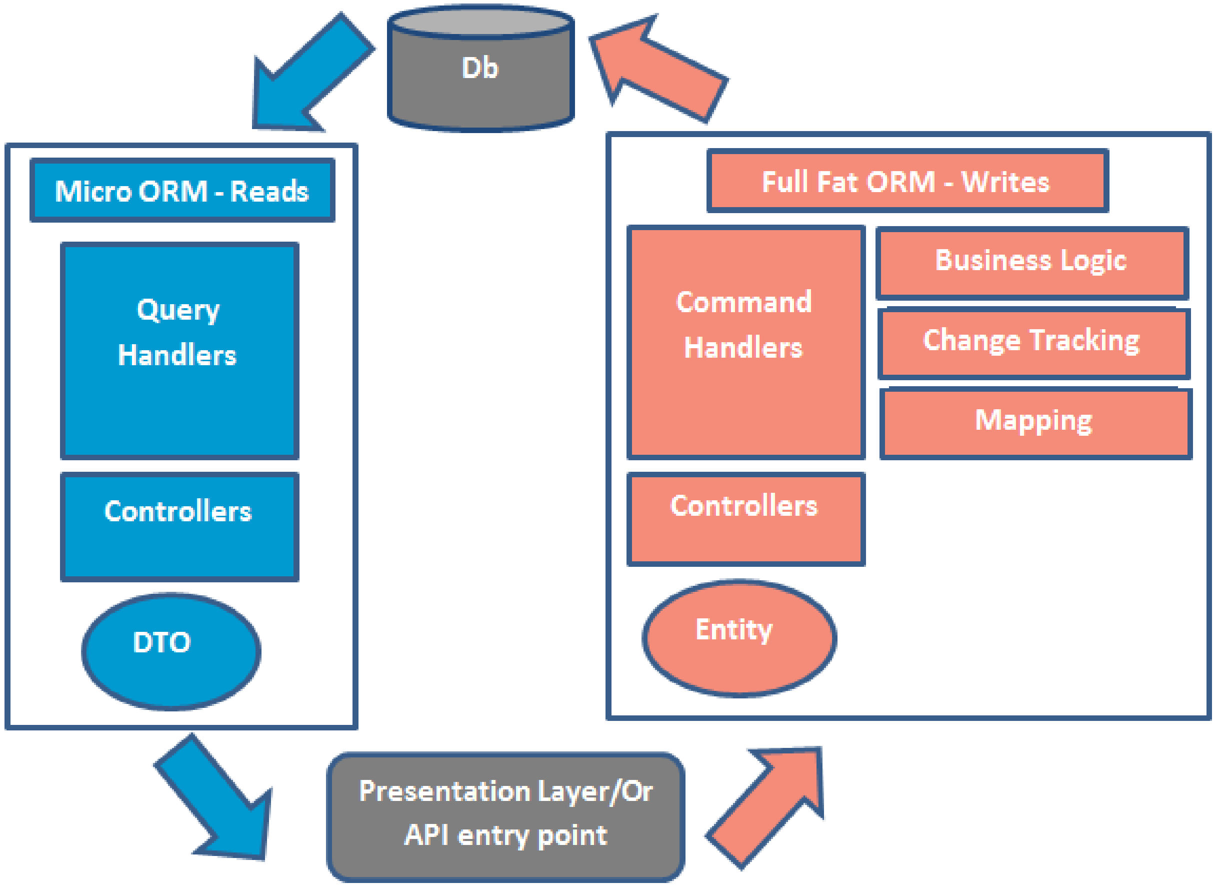
**Enforces the S I D of SOLID**

- **S** - Each command or query handler is responsible for one cohesive unit of work. Each are completely encapsulated
- **I** - Removes *God* classes such as those created by the repository (anti) pattern. Each consumer has only the interface they need
- **D** - Handlers are injected as abstractions with no dependency on implementation. Also, separates the data (commands) from the logic (handlers) removing coupling.

# In the context of DDD

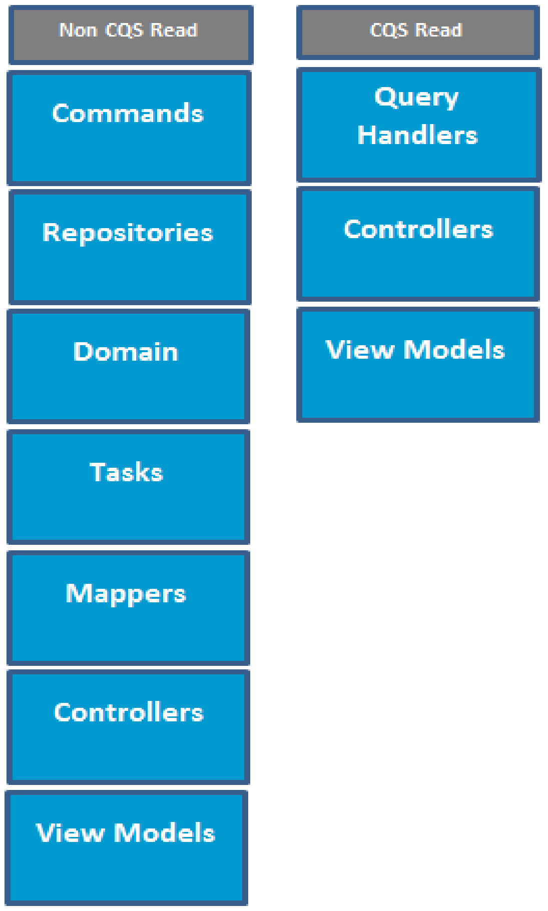
- **Excellent fit** with the DDD layered conventions
  - Presentation - Controllers
  - Application - Commands and Queries
  - Infrastructure - ORM plumbing
  - Domain - ORM entities
- Encourages **small Bounded Contexts** designed to solve one problem
- **Removes the need for complex Aggregates**. You are generally only saving one entity at a time and relationships are via foreign keys not collections
- Infinitely **more efficient and more scalable** than the `customer.Orders.Add(new Order());` (anti) pattern

# Architectural Overview



# Performance and simplicity

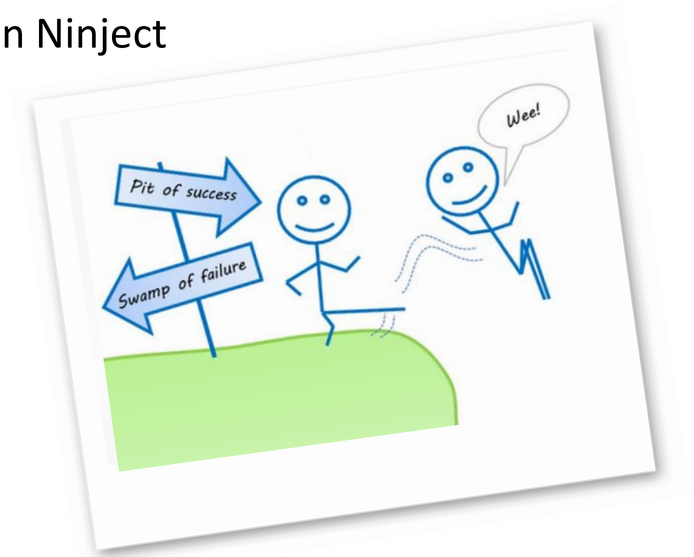
## via architecture



# Performance and simplicity via complimentary frameworks #1

## Simple Injector Container

- Incredibly **fast :** for example - 1500 x faster than Ninject
- **Simple** and **intuitive**
- Excellent **diagnostic feedback**
- Encourages you to fall into the pit of **success!**
  - No implicit property injection
  - Discourages multiple constructors
  - No Interceptors
  - No per thread lifestyle



- **No vendor lock in** no [dependency] attribute which forces your code to know about a container (only the composition root should know about a container)
- **AOP** via Decorators
- **Open generic** registration, cuts registration code in half
- **Container.Verify()** is a wonderful thing!

## Performance and simplicity
## via complimentary frameworks #2

**Dapper**

- *The* **fastest** ORM on the market:

Performance of SELECT mapping over 500 iterations:

| | |
|---|---|
| Dapper ExecuteMapperQuery | 49ms |
| Entity Framework ExecuteStoreQuery | 631ms |

- A 400 line **micro** ORM
- Written **by** Stack Overflow **for** Stack Overflow
- Very **simple** to use
- Projects data straight to **DTO**

Dapper Sam Saffron

## Why is performance so important?

- Software **speed** has been proven to strongly influence a user's overall **perception** of organisational:
  - Reliability
  - Credibility
  - Security
  - Stability
- This is especially important when selling **financial services**
- Ultimately, given two competitive sites that are identical in practically every other way, **the faster site will be more successful**

# Let’s look at some code

- **Commands** and **queries**
- **Project structure mapping to DDD concepts**
- **Simple Injector**
- **Dapper,** Dapper Extensions and Dapper Async
- **Command Unit of Work** decorators for Entity Framework
- **Query Authorisation** decorators
- Claims based authorisation and **Domain specific Claim Transformation**
- **Attribute based** Routing

## Our performance benchmarks

- **First Place** Vanilla Dapper

Dapper

# Our performance benchmarks

- **Second Place** Dapper Extensions with Dynamic SQL

Dapper Extensions

## Our performance benchmarks

- **Third Place** Dapper Async

Dapper Async

## Our performance benchmarks

- **Fourth Place** Entity Framework Projections (with overheads disabled)

Entity Framework Projections

## Our performance benchmarks

- **Wooden Spoon** Entity Framework to Domain Model

Entity Framework to Domain Model

## Conclusions

- **Architecture** choices + **Framework** choices
  = **Performance** and **Simplicity**

Any Questions?