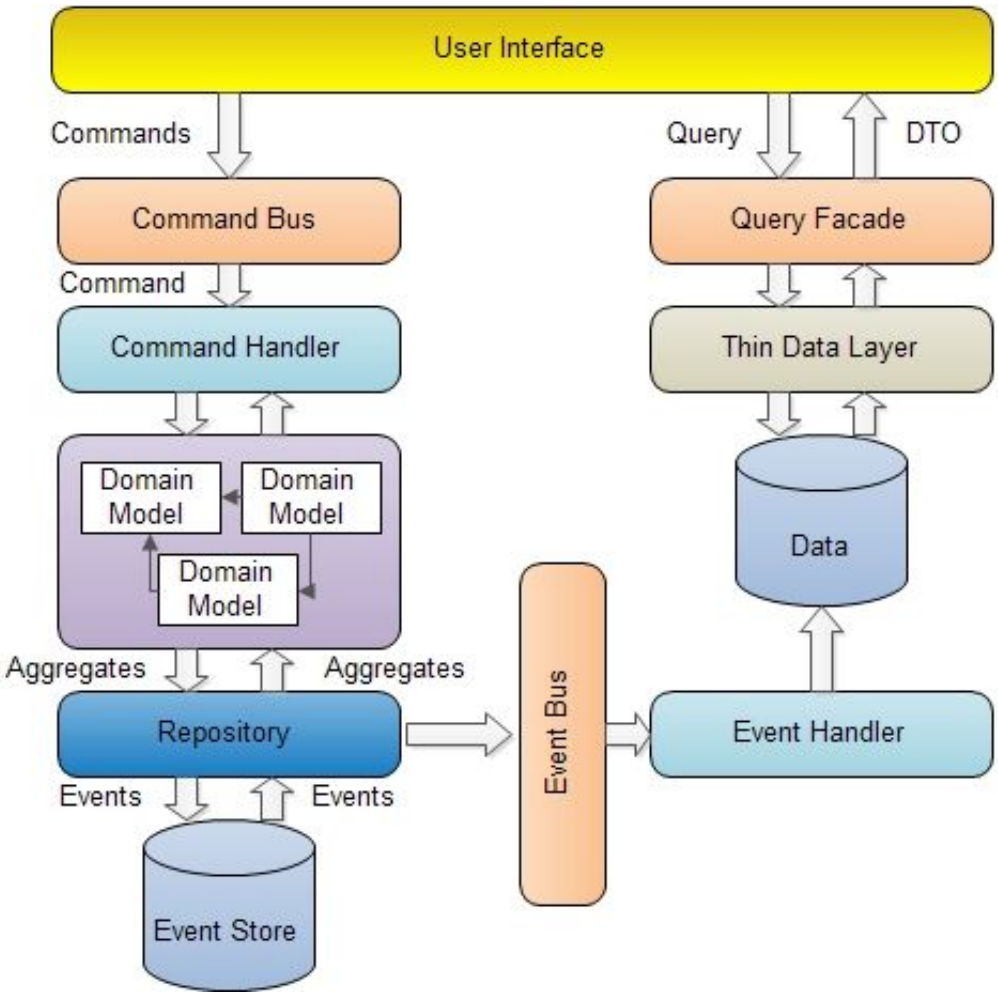
Introduction to CQRS

codeproject.com Kanasz Robert March 21st, 2013 view original

What is CQRS



CQRS means Command Query Responsibility Segregation. Many people think that CQRS is an entire architecture, but they are wrong. CQRS is just a small pattern. This pattern was first introduced by Greg Young and Udi Dahan. They took inspiration from a pattern called Command Query Separation which was

defined by Bertrand Meyer in his book “Object Oriented Software Construction”. The main idea behind CQS is: “A method should either change state of an object, or return a result, but not both. In other words, asking the question should not change the answer. More formally, methods should return a value only if they are referentially transparent and hence possess no side effects.” (Wikipedia) Because of this we can divide a methods into two sets:

- **Commands** - change the state of an object or entire system (sometimes called as modifiers or mutators).
- **Queries** - return results and do not change the state of an object.

In a real situation it is pretty simple to tell which is which. The queries will declare return type, and commands will return void. This pattern is broadly applicable and it makes reasoning about objects easier. On the other hand, CQRS is applicable only on specific problems.

Many applications that use mainstream approaches consists of models which are common for read and write side. Having the same model for read and write side leads to a more complex model that could be very difficult to be maintained and optimized.

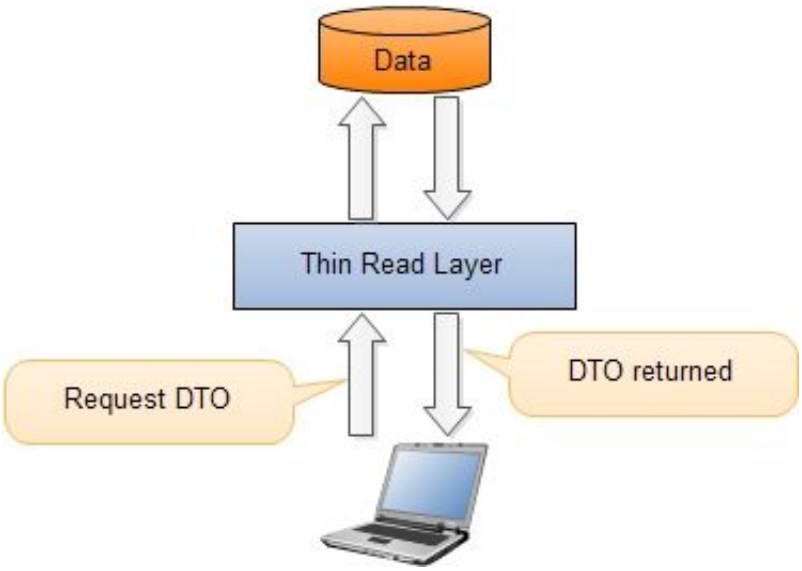
The real strength of these two patterns is that you can separate methods that change state from those that don’t. This separation could be very handy in situations when you are dealing with performance and tuning. You can optimize the read side of the system separately from the write side. The write side is known as the domain. The domain contains all the behavior. The read side is specialized for reporting needs.

Another benefit of this pattern is in the case of large applications. You can split developers into smaller teams working on different sides of the system (read or write) without knowledge of the other side. For example developers working on read side do not need to understand the domain model.

Query side

The queries will only contain the methods for getting data. From an architectural point of view these would be all methods that return DTOs that the client consumes to show on the screen. The DTOs are usually projections of domain objects. In some cases it could be a very painful process, especially when complex DTOs are requested.

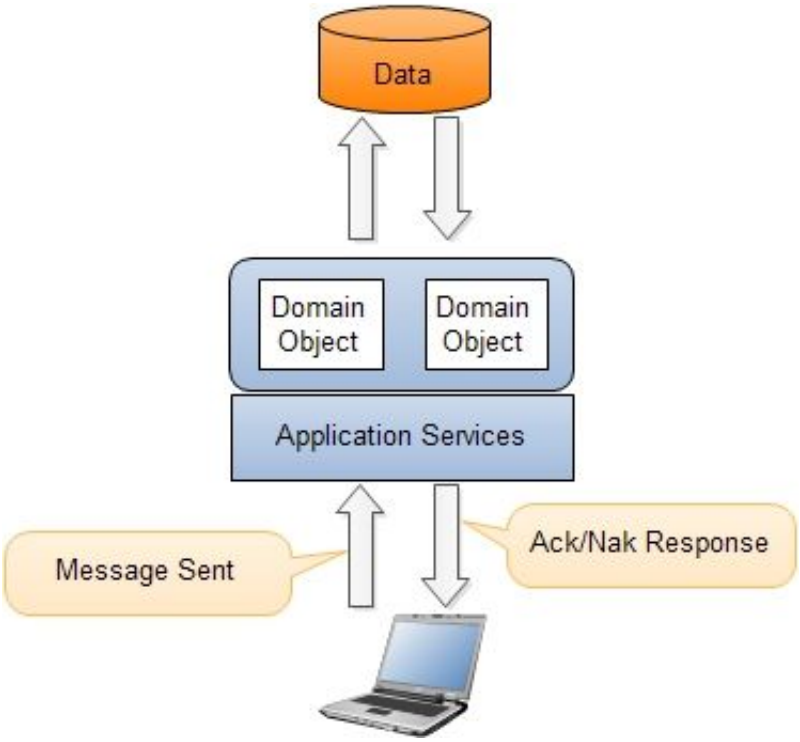
Using CQRS you can avoid these projections. Instead it is possible to introduce a new way of projecting DTOs. You can bypass the domain model and get DTOs directly from the data storage using a read layer. When an application is requesting data, this could be done by a single call to the read layer which returns a single DTO containing all the needed data.



The read layer can be directly connected to the database (data model) and it is not a bad idea to use stored procedures for reading data. A direct connection to the data source makes queries very easy to by maintained and optimized. It makes sense to denormalize data. The reason for this is that data is normally queried many times more than the domain behavior is executed. This denormalization could increase the performance of the application.

Command side

Since the read side has been separated the domain is only focused on processing of commands. Now the domain objects no longer need to expose the internal state. Repositories have only a few query methods aside from `GetById`.



Commands are created by the client application and then sent to the domain layer. Commands are messages that instruct a specific entity to perform a certain action. Commands are named like DoSomething (for example, ChangeName, DeleteOrder ...). They instruct the target entity to do something that might result in different outcomes or fail. Commands are handled by command handlers.

All commands will be sent to the Command Bus which will delegate each command to the command handler. This demonstrates that there is only one entry point into the domain. The responsibility of the command handlers is to execute the appropriate domain behavior on the domain. Command handlers should have a connection to the repository to provide the ability to load the needed entity (in this context called Aggregate Root) on which behavior will be executed.

```
public interface ICommandHandler<TCommand> where TCommand
: Command
{
    void Execute(TCommand command);
}

public class CreateItemCommandHandler :
ICommandHandler<CreateItemCommand>
```

```
{
    private IRepository<DiaryItem> _repository;

    public CreateItemCommandHandler(IRepository<DiaryItem>
repository)
    {
        _repository = repository;
    }

    public void Execute(CreateItemCommand command)
    {
        if (command == null)
        {
            throw new ArgumentNullException("command");
        }
        if (_repository == null)
        {
            throw new
InvalidOperationException("Repository is not
initialized.");
        }
        var aggregate = new DiaryItem(command.Id,
command.Title, command.Description,
                                      command.From,
command.To);
        aggregate.Version = -1;
        _repository.Save(aggregate, aggregate.Version);
    }
}
```

The command handler performs the following tasks:

- It receives the Command instance from the messaging infrastructure (Command Bus)
- It validates that the Command is a valid Command

- It locates the aggregate instance that is the target of the Command.
- It invokes the appropriate method on the aggregate instance passing in any parameter from the command.
- It persists the new state of the aggregate to storage.

Internal Events

The first question we should ask is what is the domain event. The domain event is something that has happened in the system in the past. The event is typically the result of a command. For example the client has requested a DTO and has made some changes which resulted in a command being published. The appropriate command handler has then loaded the correct Aggregate Root and executed the appropriate behavior. This behavior raises an event. This event is handled by specific subscribers. Aggregate publishes the event to an event bus which delivers the event to the appropriate event handlers. The event which is handled inside the aggregate root is called an internal event. The event handler should not be doing any logic instead of setting the state.

Domain Behavior

```
public void ChangeTitle(string title)
{
    ApplyChange(new ItemRenamedEvent(Id, title));
}
```

Domain Event

Internal Domain Event Handler

```
public void Handle(ItemRenamedEvent e)
{
    Title = e.Title;
}
```

Events are usually connected to another pattern called Event Sourcing (ES). ES is an approach to persisting the state of an aggregate by saving the stream of events in order to record changes in the state of the aggregate.

As I mentioned earlier, every state change of an Aggregate Root is triggered by an event and the internal event handler of the Aggregate Root has no other role than setting the correct state. To get the state of an Aggregate Root we have to replay all the events internally. Here I must mention that events are write only. You cannot alter or delete an existing event. If you find that some logic in your system is generating the wrong events, you must generate a new compensating event correcting the results of the previous bug events.

External Events

External events are usually used for bringing the reporting database in sync with the current state of the domain. This is done by publishing the internal event to outside the domain. When an event is published then the appropriate Event Handler handles the event. External events can be published to multiple event handlers. The Event handlers perform the following tasks:

- It receives an Event instance from the messaging infrastructure (Event Bus).
- It locates the process manager instance that is the target of the Event.
- It invokes the appropriate method of the process manager instance passing in any parameters from the event.
- It persists the new state of the process manager to storage.

But who can publish the events? Usually the domain repository is responsible for publishing external events.

Using the Code

I have created a very simple example that demonstrates how to implement the CQRS pattern. This simple example allows you to create diary items and modify them. The solution consists of three projects:

- Diary.CQRS
- Diary.CQRS.Configuration
- Diary.CQRS.Web

The first one is the base project that contains all domain and messaging objects. The Configuration project is consumed by Web which is the UI for this example. Now let's take a closer look at the main project.

Diary.CQRS

As I mentioned earlier, this project contains all the domain and messaging objects for this example. The only entry point for the CQRS example is the Command Bus into which commands are sent. This class has only one generic method `Send(T command)`. This method is responsible for creating the appropriate command handler using `CommandHandlerFactory`. If no command handler is associated with a command, an exception is thrown. In other case, the `Execute` method is called in which a behavior is executed. The Behavior creates an internal event and this event is stored into an internal field called `_changes`. This field is declared in the `AggregateRoot` base class. Next, this event is handled by the internal event handler which changes the state of an Aggregate. After this behavior is processed, all the aggregate's changes are stored into the repository. The repository checks whether there are some inconsistencies by comparison of the expected version of the aggregate and the version of the aggregate stored in the storage. If those versions are different, it means that the object has been modified by someone else and a `ConcurrencyException` is thrown. In other case the changes are stored in the Event Storage.

Repository

InMemoryEventStorage

In this simple example I have created an `InMemoryEventStorage` which stores all events into memory. This class implements the `IEventStorage` interface with four methods:

```
public IEnumerable<Event> GetEvents(Guid aggregateId)
{
```

```
        var events = _events.Where(p => p.AggregateId ==
    aggregateId).Select(p => p);
        if (events.Count() == 0)
        {
            throw new
    AggregateNotFoundException(string.Format(
              "Aggregate with Id: {0} was not found",
    aggregateId));
        }
        return events;
    }
```

This method returns all events for the aggregate and throws an error when there aren't events for an aggregate which means that aggregate doesn't exist.

This method stores events into memory and creates every three events memento for the aggregate. This memento holds all state information for the aggregate and the version. Using mementos increases the performance of the application because it is not important to load all the events but just the last three of them.

When all events are stored, they are published by the Event Bus and consumed by the external Event Handlers.

```
    public T GetMemento<T>(Guid aggregateId) where T :
    BaseMemento
    {
        var memento = _mementos.Where(m => m.Id ==
    aggregateId).Select(m=>m).LastOrDefault();
        if (memento != null)
            return (T) memento;
        return null;
    }
```

Returns memento for aggregate.

```
public void SaveMemento(BaseMemento memento)
{
    _mementos.Add(memento);
}
```

Stores memento for aggregate.

Aggregate Root

The `AggregateRoot` class is the base class for all aggregates. This class implements the `IEventProvider` interface. It holds information about all uncommitted changes in the `_changes` list. This class also has an `ApplyChange` method which executes the appropriate internal event handler. The `LoadFromHistory` method loads and applies the internal events.

```
public abstract class AggregateRoot:IEventProvider
{
    private readonly List<Event> _changes;

    public Guid Id { get; internal set; }
    public int Version { get; internal set; }
    public int EventVersion { get; protected set; }

    protected AggregateRoot()
    {
        _changes = new List<Event>();
    }

    public IEnumerable<Event> GetUncommittedChanges()
    {
        return _changes;
```

```
        }

        public void MarkChangesAsCommitted()
        {
            _changes.Clear();
        }

        public void LoadsFromHistory(IEnumerable<Event>
    history)
        {
            foreach (var e in history) ApplyChange(e, false);
            Version = history.Last().Version;
            EventVersion = Version;
        }

        protected void ApplyChange(Event @event)
        {
            ApplyChange(@event, true);
        }

        private void ApplyChange(Event @event, bool isNew)
        {
            dynamic d = this;


    d.Handle(Converter.ChangeTo(@event,@event.GetType()));
            if (isNew)
            {
                _changes.Add(@event);
            }
        }
    }
```

EventBus

Events describe changes in the system's state. The primary purpose of the events is to update the read model. For this purpose I have created the `EventBus` class. The only behavior of the `EventBus` class is publishing events to subscribers. One event can be published to more than one subscriber. In this example there is no need for a manual subscription. The event handler factory returns a list of all `EventHandler`s that can process the current event.

Event Handlers

The primary purpose of event handlers is to take the events and update the read model. In the example below you can see the `ItemCreatedEventHandler`. It handles the `ItemCreatedEvent`. Using information from the event it creates a new object and stores it in the reporting database.

```
public class ItemCreatedEventHandler :
IEventHandler<ItemCreatedEvent>
{
    private readonly IReportDatabase _reportDatabase;
    public ItemCreatedEventHandler(IReportDatabase
reportDatabase)
    {
        _reportDatabase = reportDatabase;
    }
    public void Handle(ItemCreatedEvent handle)
    {
        DiaryItemDto item = new DiaryItemDto()
            {
                Id = handle.AggregateId,
                Description =  handle.Description,
                From = handle.From,
                Title = handle.Title,
                To=handle.To,
                Version =  handle.Version
            };

        _reportDatabase.Add(item);
```

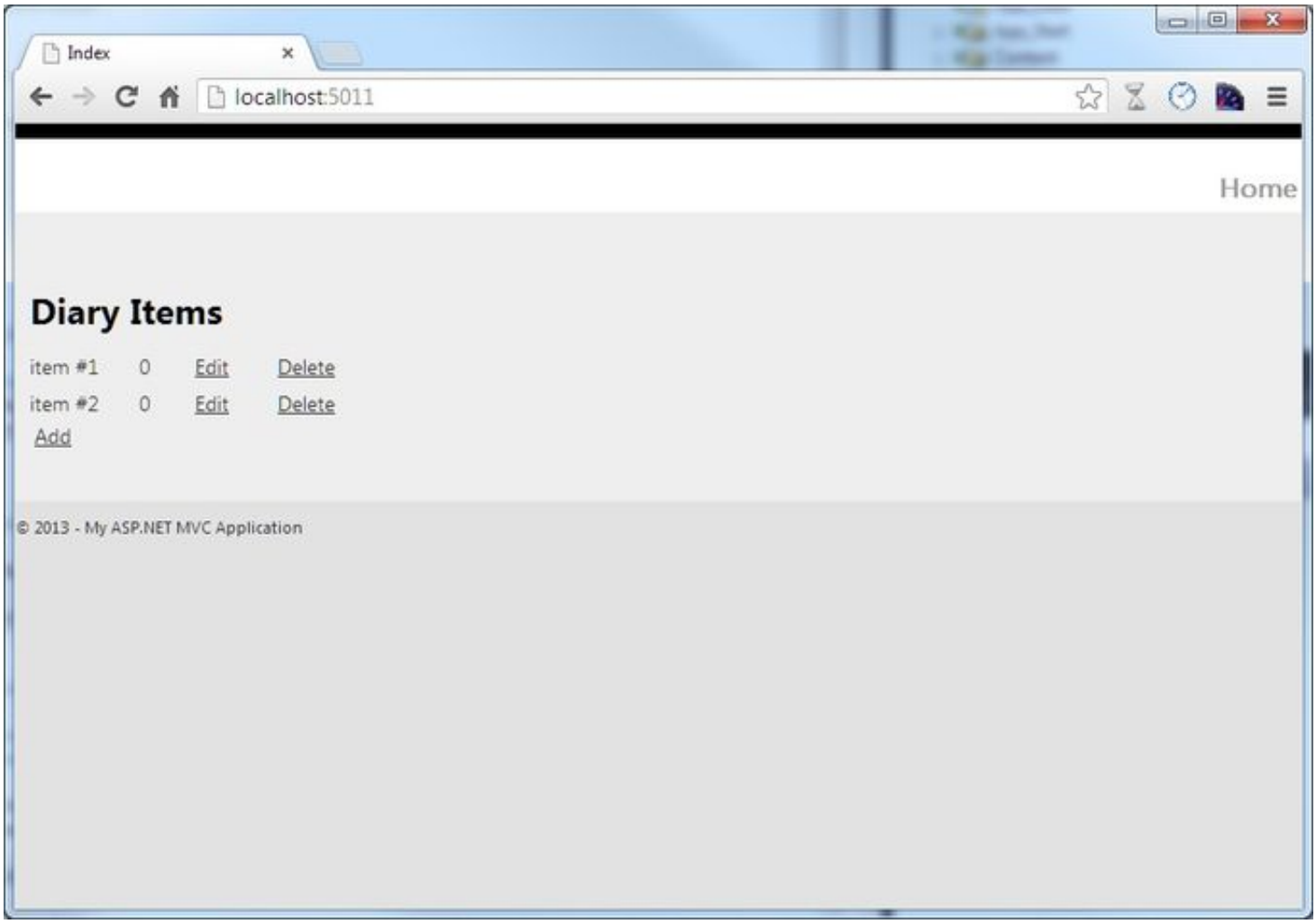
```
        }
    }
```

Diary.CQRS.Web

This project serves as UI for the CQRS example. The Web UI project is a simple ASP.NET MVC 4 application with only one `HomeController` with six `ActionResult` methods:

- `ActionResult Index()` - this method returns the Index view which is the main view of this application where you can see the list of all diary items.
- `ActionResult Delete(Guid id)` - this method creates a new `DeleteItemCommand` and sends it to `CommandBus`. When a command is sent, the method will return Index view.
- `ActionResult Add()` - returns Add view where you can input data for a new diary item.
- `ActionResult Add(DiaryItemDto item)` - this method creates a new `CreateItemCommand` and sends it to the `CommandBus`. When a new item is created, the Index view is returned.
- `ActionResult Edit(Guid id)` - returns the Edit view for a selected diary item.
- `ActionResult Edit(DiaryItemDto item)` - this method creates a new `ChangeItemCommand` and sends it to the `CommandBus`. When an item is successfully updated, the Index screen is returned. In the case of `ConcurrencyError`, the edit view is returned and an exception is displayed on screen.

In the picture below you can see the main screen with a list of diary items.



When to use CQRS

In general, the CQRS pattern could be very valuable in situations when you have highly collaborative data and large, multi-user systems, complex, include ever-changing business rules, and delivers a significant competitive advantage of business. It can be very helpful when you need to track and log historical changes.

With CQRS you can achieve great read and write performance. The system intrinsically supports scaling out. By separating read and write operations, each can be optimized.

CQRS can by very helpful when you have difficult business logic. CQRS forces you to not mix domain logic and infrastructural operations.

With CQRS you can split development tasks between different teams with defined interfaces.

When not to use CQRS

If you are not developing a highly collaborative system where you don't have multiple writers to the same logical set of data you shouldn't use CQRS.

History

- 4 March - Original version posted.