Blu-Softwave : Layer

Command Query Segregation Responsibility (CQRS) with Event Sourcing

Bill Anderson |
bill.anderson@blusoftwave.com

Apex

Table of Contents

Overview ***2***

Our Path to CQRS? ***3***

About the Design ***6***

Advantages and Disadvantages *6*

Resolver ***7***

HandlerResolverBase *8*

Command Dispatcher Resolver *8*

Command - CQRS ***8***

Command *9*

Command Dispatcher *9*

Command Handlers *10*

Query - CQRS ***10***

Query Handlers *11*

Query Dispatcher *12*

Sample Code ***12***

Caveat-Preemptor *12*

Sample Command *13*

Sample Query *14*

Sample Service *15*

Summary ***17***

Appendix: Improvements ***18***

Overview

Through the course of looking at many custom designs and implementations we understand why, as Consultants, we are brought into the fray. The grains of sand became a mountain of pain.

In an initial Salesforce customization, thought was to create functionality to appease the Business as quickly as possible. Timelines, cost, resources cause an adoption of "*quick-and-dirty*" or general lack of design knowledge when it came to customization. And, as this adoption continues it is discovered what was a CRM, ***Customer-Resource-Management***, system is now, ***Custom-Recurring-Mess***. Blunt as that may sound, it is easier said than remediated.

This document outlines steps that can be taken to alleviate the customization pain as well as how one provides event monitoring and instrumentation. The latter **NOT** being on the top of many Salesforce customers. Let's revisit a proven design pattern, **CQRS with Event Sourcing**, which may have been forgotten in many customized Salesforce environments. In a high-level N-tiered architecture, CQRS lies in the *Service* Layer. The document provides a glimpse into this layer and how one can expand it to alleviate the mess.

This work stands on the shoulders of others. We take those common design patterns to provide a solid solution.

Finally, below is the layered architecture used with a current focus on Service Layer (i.e. CQRS). The layered architecture provides a way to manage, segragate and maintain functionality without being overwhelmed. Our ultimate goal is to decompose these components into layers, packages, and the use of Design Patterns. Why?

Due to complexity, we

- Use layers to segment and manage,
- Use Design Patterns for Guidance/Best Practice
- Use Packages for Reuse and Support

We do not go into low-level details about **Design Patterns**, except:

They isolate the variability and make systems easier to understand, maintain and communicate.

Domain Driven Design (DDD) also comes into the picture but that is a topic for another day.

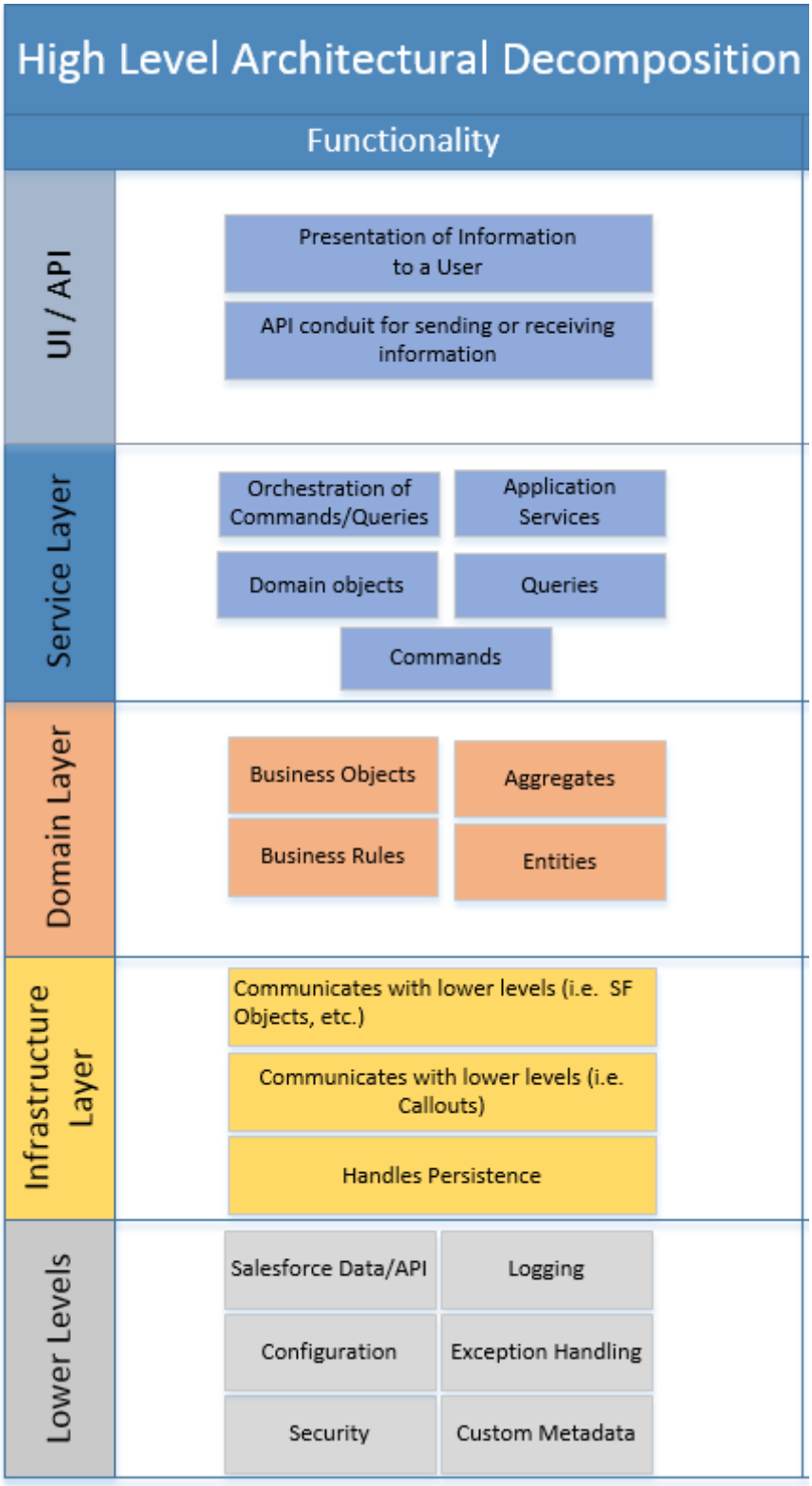


Figure 1 N-tiered Architecture

Our Path to CQRS?

When customers start down the path of customization, i.e., *Apex*, they need to first consider *Clicks, Not Code*. But, if there is a need for customization, then they should think about design patterns that are well proven, because a road well-traveled provides guidance for a smoother surface of enjoyment. Let's jump into the main course.

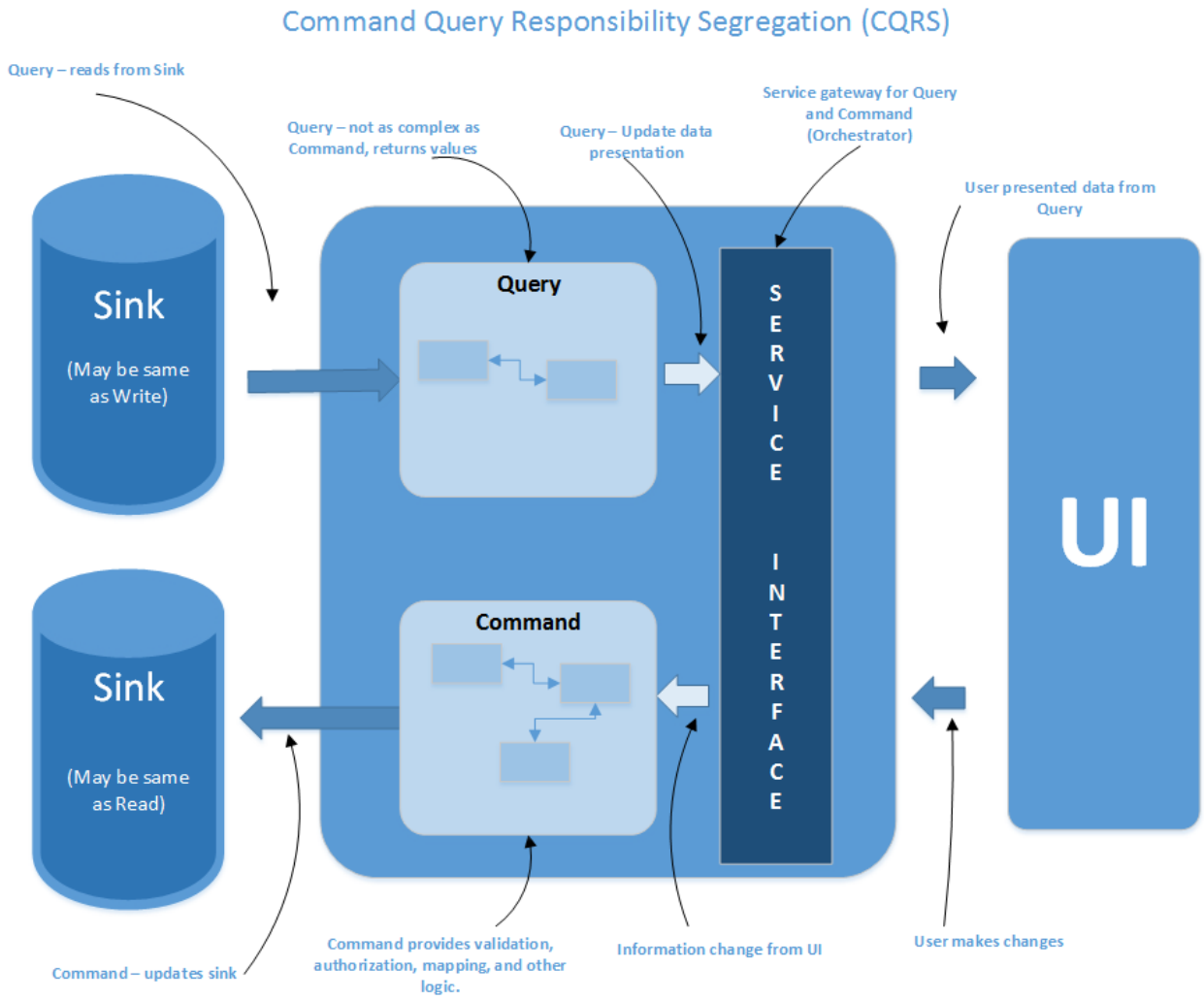
Often in a Salesforce environment there is no distinction between a command (*mutable* state) and a query (*non-mutable* state). Nor is there a distinction between a Domain Object (i.e. Account, Contact, etc.) and a Data Transfer Object (DTO).

The lines become blurred and difficult to manage; in addition, there is no distinction between data consumed by different personas. For example, a financial customer is often given the same Salesforce object (i.e., Account) as an internal employee, i.e., Financial Planner. This leakage of information gets further complicated when fields and business logic are added and updated (constantly). The lack of signal responsibility, abstraction, and design patterns leaves reuse out until one sort out the H* Soup! This blurred state makes for a rather tangled mess and a simple change ripples breaking functionality throughout a brittle design. What can be done to address these concerns? *CQRS to the Rescue*!

Command-**Q**uery **R**esponsibility **S**egregation, or **CQRS**, provides the ability to separate Query from Commands responsibilities using SOLID Principles. Queries (*non-mutable*) retrieve information from a sink (data store) for the user. While a Command mutates information from a sink (data store).

A command performs a task, such as update a sink (data store). Commands mutate state, while a Query does not. Technically, a Command does not return a value; however, the example which follows will return status. Each provides a single responsibility (**S**ingle Responsibility and **L**iskov Substitutability principle in ***SoLid)***.

The diagram below shows two sinks, but these could be the same sink; they do not have to be separate. For example, in Heroku, one can setup a Postgres *follower*. The follower is a latent read-only data from a transaction database (i.e. Salesforce).



Definitions in a CQRS Design

The above information had quite a bit of domain specific context and as you read further there will be additional terms used regarding CQRS. Below are high-level definitions.

- **Domain Objects**. Objects from the business specific area that represent something meaningful to the domain expert. Domain objects are mostly represented by entities and value objects. Most objects that live in domain layer contribute to the model and are domain objects. A Command and Query will deal with Domain objects and DTOs.
- **Data Transfer Object (DTO)**. Provides a singular focus of data needed for a specific persona. These Objects usually have just public exposed data properties. They provide the view-model on a Domain Object; tailored to a specific persona's need or view.
- The **Query**, **QueryResult**, **Command** and **CommandResult** objects: These are the actual classes used to make a request from/to a data store in an application. For example,

GetBookByIdQuery, *GetAllBooksQuery*, *UpdateBookCommand* and *DeleteBookCommand* could all be actions you'd find in a CQRS architecture.

- **Dispatcher**: This is where the chain starts. You tell the Dispatcher to dispatch a Query, Command or Service object. The Dispatcher then passes your request to the correct Handler.
- **Resolver** – Resolves the right handler for a Command, Query or Service.
- **Abstract base class** for *Command*, *Query and Service Handlers*:
    - These objects allow you to manage code that is common in all handlers. Functionalities like logging, authorization, exception handling, etc. can be consumed here.
- **Handlers**: The classes that handle your request and either retrieves data or mutates data. There **should always be one Handler mapping to one action**. So, you would have a *GetAccountIdQueryHandler*, *GetAllOrdersQueryHandler*, *UpdateAccountCommandHandler* and *DeleteContactCommandHandler* that handle the Queries and Commands above.
- **IoC**: Inversion of Control Container will be a Mapping Engine, to glue DTOs, Domain and SObjects together.

About the Design

The design applies various Design Patterns and **SOLID** Principles. It borrows from many references as the topic is well-written and covered. Much of the initial design is around Interfaces and Abstractions. Why? Understanding the WHAT we are doing before understanding HOW we are doing it provides the following advantages.

- *Easier to write and use* **Test Driven Design (TDD**). Defining the abstraction allows me to create classes and process without concreteness,
- *Easier to Test*. Abstraction allows one to mock functionality to verify and validate flow and execution (in a scratch org or sandbox),
- *Easier to Change*. There is no concreteness to inhibit (or slow-down) progress, and if there is, it is minor as we move from red to yellow to green in TDD.
- *Design Patterns* are defined patterns that are proven, named, communicated, and explained. For example, if I say "*Factory*" or "*Singleton*", I have communicated intent & usage in one word. Makes for short design meetings!

Finally, the reference to classes in the document are contained within a namespace, ***blsw***.

Advantages and Disadvantages

Below is a list of advantages and disadvantages of a CQRS design,

Advantages

- Single Responsibility

- Clear Separation of Concerns (Read and Write)
- Easy to test and validate a business process
- Performant
- Easy to plug in new functionality in a business process
- Can generate code

Disadvantages

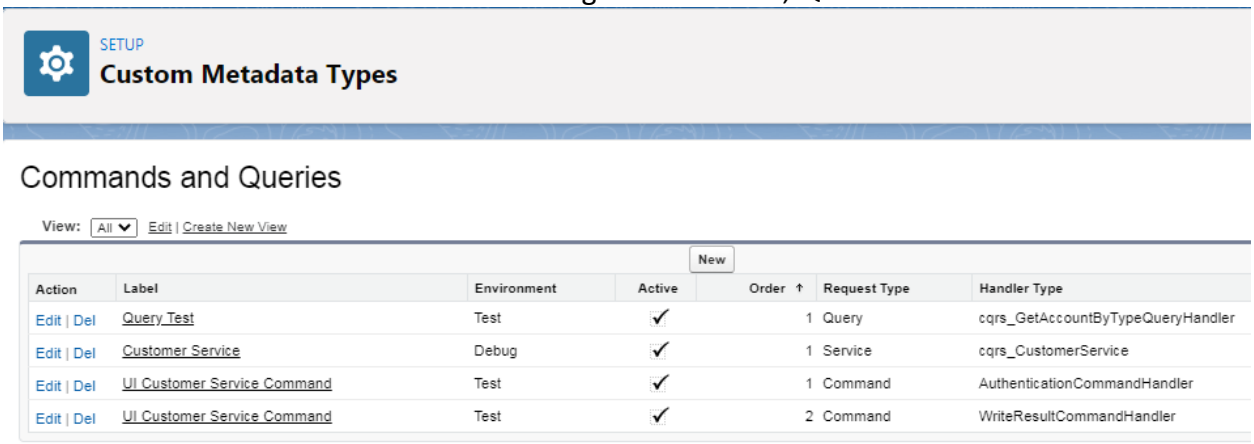
- For simple implementations this may be too much
- For each command, query or service, there is a handler. This can proliferate to a large number if not governed well. Administrators will need to maintain a correct business process functionality mapping.
- Custom Metadata can become daunting to manage and subject to errors if not managed

Resolver

The Resolver provides the Dispatchers (Command and Query) the ability to resolve handlers. There are two resolvers which segregates the functionality of resolving the handlers in the Command Dispatcher and Query Dispatcher. This was done to support single responsibilities and maintenance ease.

Both the Command Handler Resolver, ***CommandHandlerResolver*** and the Query Handler Resolver***, QueryHandlerResolver,*** inherit from the same base class, ***HandlerResolverBase***[1]. This **protected** base class injects the resolver and provides the salient method, ***resolve***, to its child classes. If there are any further adjustments needed with resolution of handlers, the changes can be isolated in the base class.

As noted, the base class is a *protected virtual* class. This provides the ability to extend behavior. The resolver uses a custom metadata to bring in Commands, Queries and Services.



Action	Label	Environment	Active	Order ↑	Request Type	Handler Type
Edit | Del	Query Test	Test	✓	1	Query	cqrs_GetAccountByTypeQueryHandler
Edit | Del	Customer Service	Debug	✓	1	Service	cqrs_CustomerService
Edit | Del	UI Customer Service Command	Test	✓	1	Command	AuthenticationCommandHandler
Edit | Del	UI Customer Service Command	Test	✓	2	Command	WriteResultCommandHandler

Please note the label and the request type is used to look up either service, command, or query. Of course, there are many options that can be implored to this effort. The Command and

[1] Generic (Templated) class would be nice Apex!

Query Dispatchers look at the *Request type* and *Handler Type* to find the concrete implementation (not shown). The Service Provider uses the resolver to determine the service by request type (i.e., *Service*) and *Label*.
In addition, as it has been mentioned that a Command, Query or Service will have an associated Handler, there is a Custom Setting option that can be turned. If on, the Handler does not have to be registered, as the underlying system can find extension (assuming one follows the convention of appending, ***Handler***, to the name).

HandlerResolverBase

The ***HandlerResolveBase*** wraps the injection of ***IResolver*** in a protected virtual method, **getResolver()**. Why? This allows the following:

1. Change the underlying resolver by overriding method (extension)
2. Easy to test with mocks

This supports Open-Closed Principle **(**O in SOLID**)** – *Open for extension, Closed for modification*.

Command Dispatcher Resolver

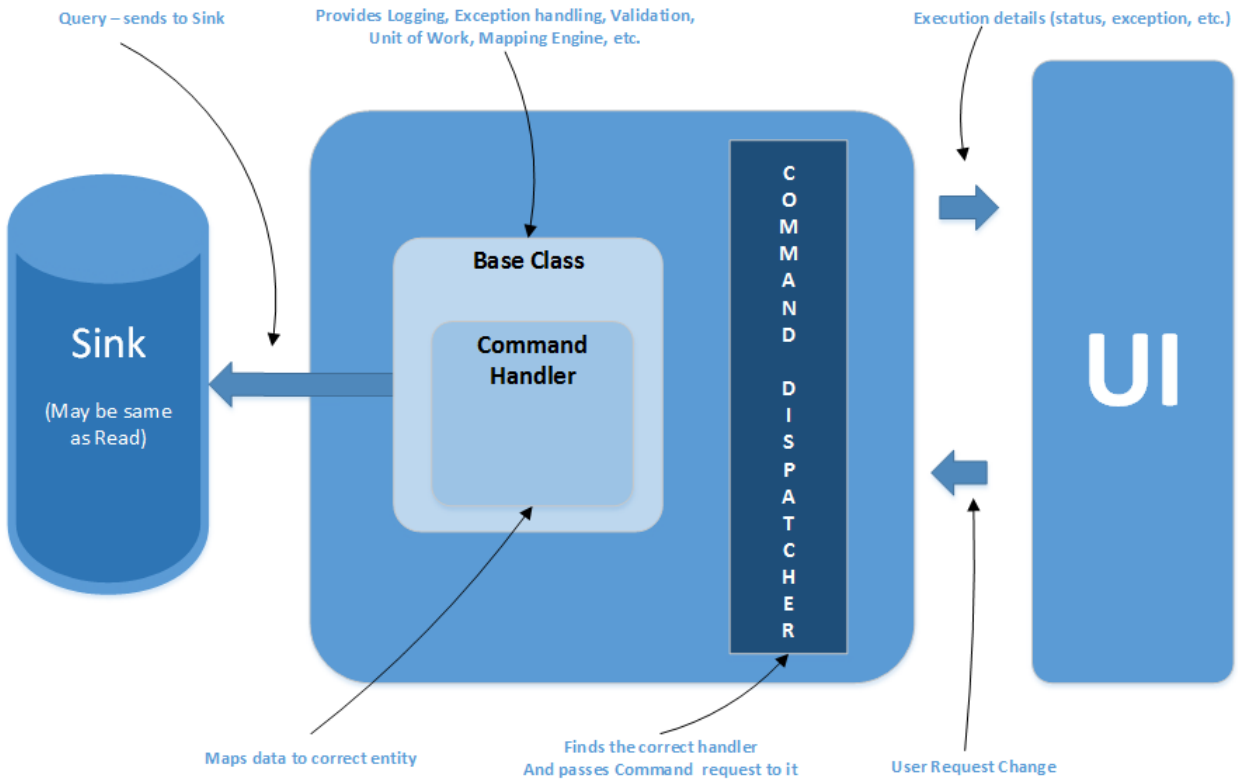
The Command Dispatcher uses a ***CommandHandlerResolver*** to resolve a command to the correct command handler. This class inherits from ***HandlerResolveBase*** and implements ***ICommandHandlerResolver***. Why? ***CommandHandlerResolver*** ONLY needs to know above resolving a handler, ***ICommanHand***ler, for an ***ICommand*****.**

This supports Interface Segregation **(*I*** in SOL***I***D**).**

Command - CQRS

The Command part of CQRS is shown in the diagram below. The Command contains the request from an Actor (User/System). The command is a specific task, a single responsibility. For example, update customer account (*UpdateCustomerAccountCommand*) or create an order for customer (*CustomerOrderUpdateCommand*). Please note, all commands classes end in "*Command*" and inherit from ***ICommand***.

The next sections break down a Command, Command Dispatcher and Command Handler.



Command

A Command provides functionality around a mutating state. A command will inherit from ***ICommand*** and ***IEntityIdentifier***. These two abstractions provide the following:

- ***ICommand*** – Demarcation of a Command.
- ***IEntityIdentifier* –** Includes references to,
  - ***Guid* –** Unique Identifier
  - ***Name* –** Name of command
  - ***Context* –** User Context
  - ***ComponentType* –** Type of command

In the event we do not need the above functionality we can remove from interface. However, to provide ample information about a command, the above information becomes useful for auditing and playback (if needed); thinking about *event sourcing*.

Command Dispatcher

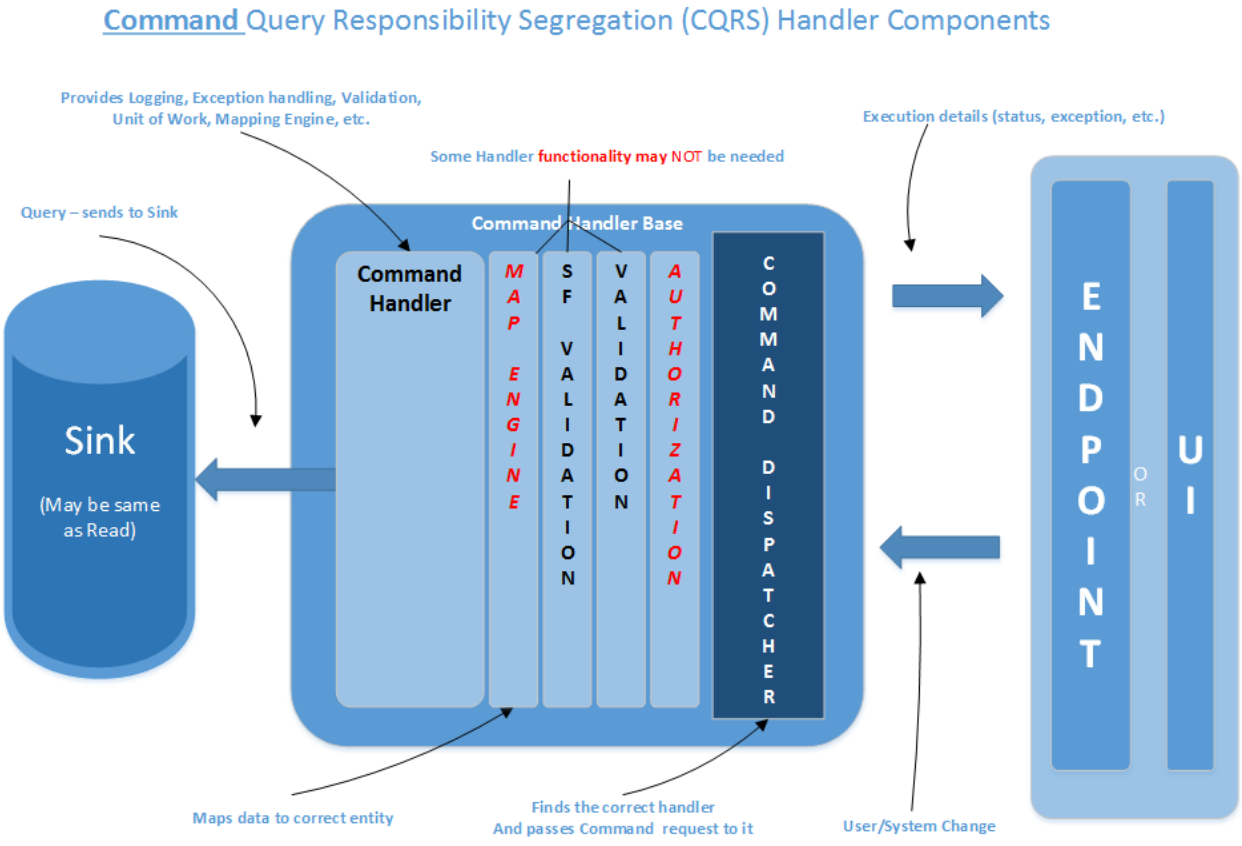
A Command Dispatcher calls a resolver, i.e., ***CommandHandlerResolver***, to resolve the handler for a Command. A Command **has** an associated Handler. A Command Dispatcher can dispatch a single or collection of Commands.

Command Handlers

Command Handlers are usually mapped one-for-one with a Command. For example, if one has a Command, *UpdateNameByAccountIdCommand*, you will have a corresponding handler, *UpdateNameByAccountIdCommandHandler.* One will have to be mindful of Apex class name limit (255). The handler is called from a Command Dispatcher, *CommandDispatcher*.

A Command Handler will also encapsulate functionality such as, but not limited to,

- **Unit of Work** – Ensure the integrity of a Command state.
- **Caching** – Cache previous request
- **Logging** - May not be needed, but if not a Cross-Cutting-Concerns, it unifies output
- **Map Engine** – maps DTO to Domain and vice-versa
- **Validation** – Validate incoming data. May also have another validator for Salesforce,
- **Salesforce Data Validation** – Validate the integrity of data into Salesforce. Note, this is different from the above validation (which validates the user input to the command)



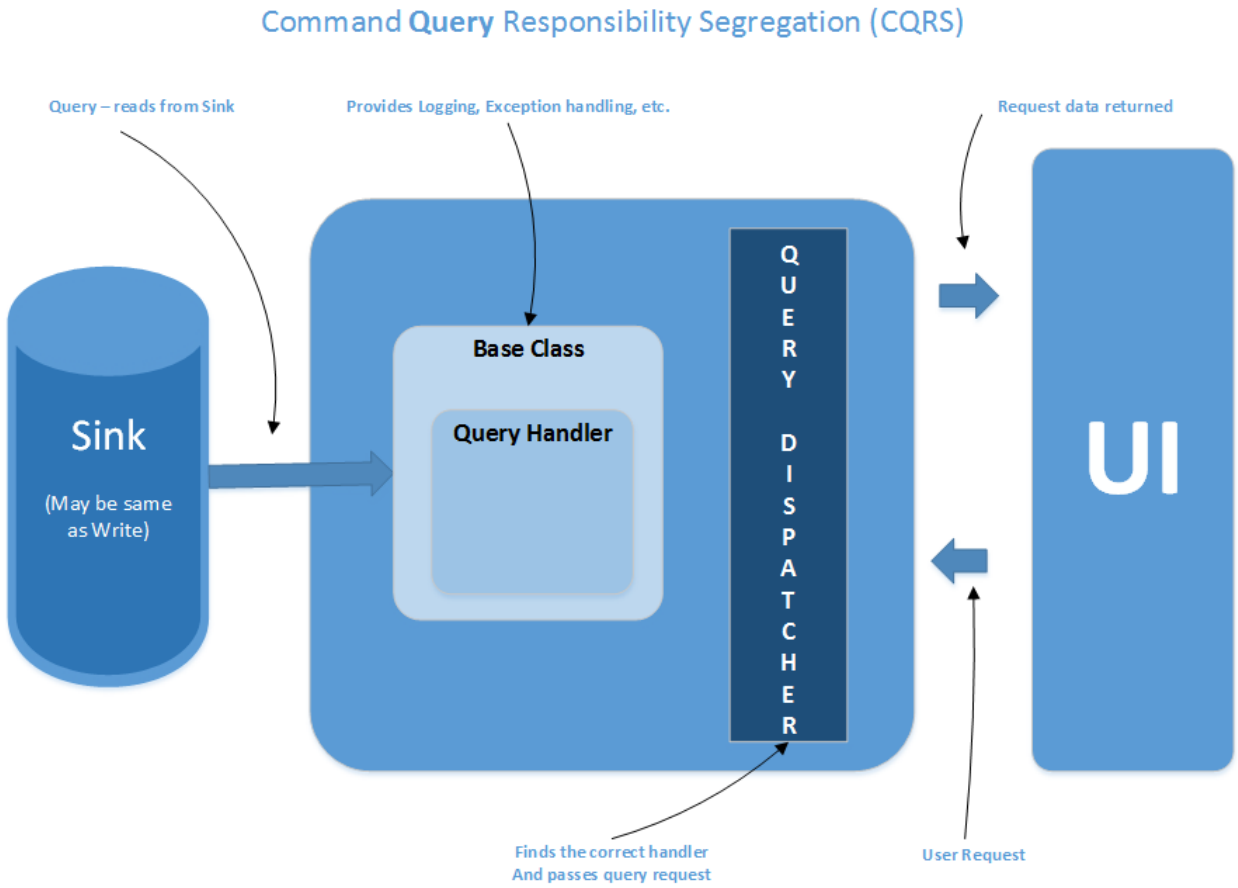
Query - CQRS

A Query provides functionality around retrieval of information – non-mutating state. A query will inherit from ***IQuery*** and ***IEntityIdentifier***. These two abstractions provide the following:

- **Query** – Demarcation of a Query.
- ***IEntityIdentifier* –** Includes references to,
    - ***Guid* –** Unique Identifier
    - ***Name* –** Name of command
    - ***ComponentType* –** Type of command

Providing ample information about a command, the above information becomes useful for auditing and playback (if needed).

The Query part of CQRS is shown in the diagram below. The Query contains the request from an Actor (User/System). The query is a specific retrieval, a single responsibility. For example, get customer account from Id (*GetCustomerAccountQuery*) or get order details from order id (*GetOrderDetailsQuery*). Please note, all query classes end in "*Query*" and inherit from ***IQuery***.

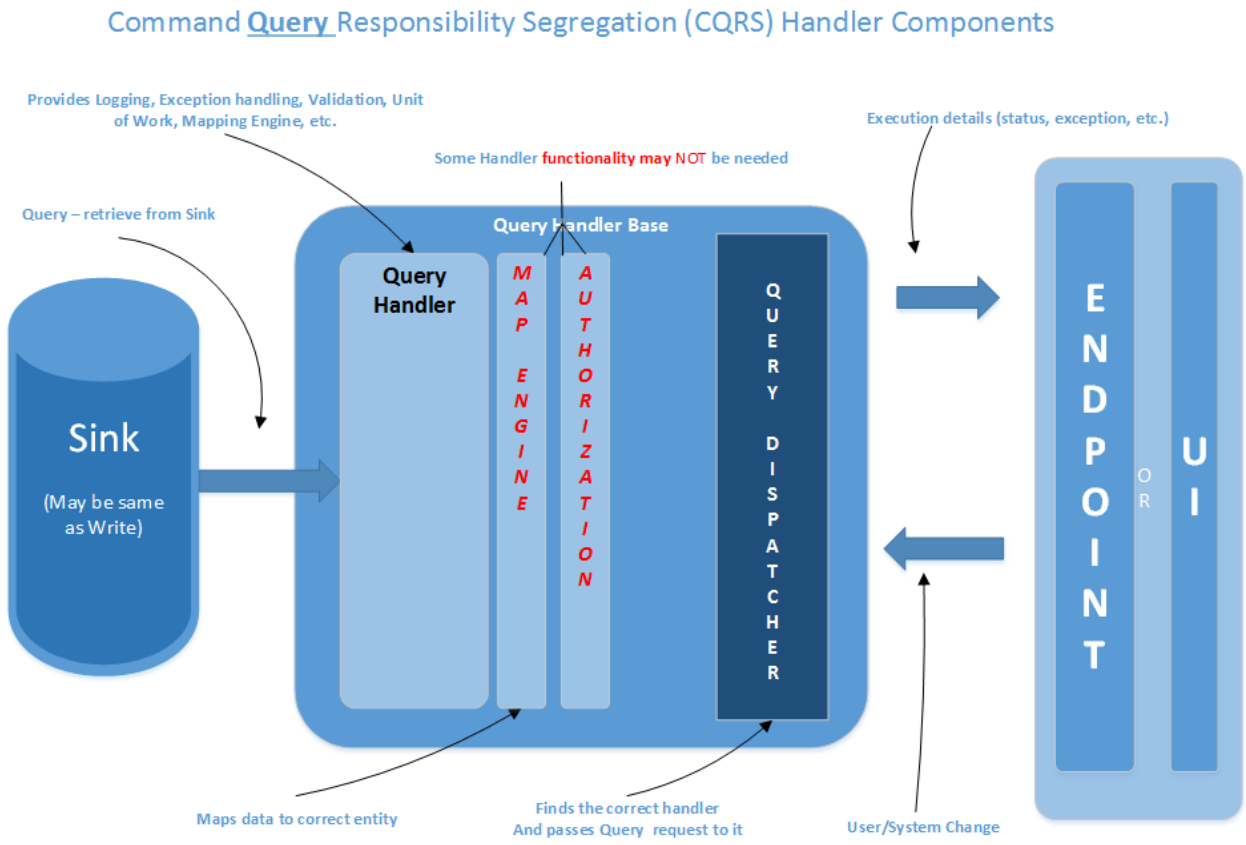


Query Handlers

Query Handlers are usually mapped one-for-one with a Query. For example, if one has a Query, *GetAccountIdByNameQuery*, you will have a corresponding handler, *GetAccountIdByNameQueryHandler.* One will have to be mindful of Apex class name limit (255). The handler is called from a Query Dispatcher, *QueryDispatcher*.

A Query Handler may not have as much functionality as a Command, but limited to,

- **Logging** – Logging user defined output.
- **Caching** – Previous request
- **Map Engine** – maps DTO to Domain and vice-versa.
- **Validation** – Validate incoming data.



Query Dispatcher

A Query Dispatcher calls a resolver, i.e., ***QueryHandlerResolver***, to resolve the handler for a Query. A Query has an associated Handler. A Query Dispatcher can dispatch a single or collection of Queries.

Sample Code

The sample code below follows the same steps as defined above. The code shows the Command, Query, Handlers, Dispatchers and Results. It is an exemplar to allow users to explore and understand CQRS.

Caveat-Preemptor

The sample code is just that. There is room for **much** improvement and robustness. Finally, the code uses the namespace **blsw**.

Sample Command

The simplest thing to do is start with a sample command. This example is found in the source tree ***(./scripts/apex/exampleCommand.apex***). Please note this is a raw form and can be used in a service supporting specific functionality (i.e., Customer Management, Lead Management, etc.)

```
List<blsw.ICommand> commands = new List<blsw.ICommand> {
    new blsw.AuthenticationCommand('test-uid','test-password'),
    new blsw.WriteResultCommand('test-id')
};
Blsw.ICommandResult result= new blsw.CommandDispatcher().dispatch(commands);
System.debug('+++++++++++++++++RESULTs++++++++++++++++++++++++++++');
System.debug('Command(s) Result Successful?:' + result.success());
System.debug('Command(s) Result:' + result);
================================
```

Above, is a collection of commands, creates a Command Dispatcher[2], passes in the command collection, and the dispatcher finds the appropriate handlers and executes.

```
Details
[118]|DEBUG|+++++++++++++++++++++++++++++++++++++++++++++++++
[118]|DEBUG|+++++ Command Name :cqrs_AuthenticationCommand
[118]|DEBUG|+++++ Command Guid :469e8f38c9bcf6591eaf5c29fa33bacf
[118]|DEBUG|+++++ Command User Context:util_UserContext:[FIRSTNAME=firstname, LASTTNAME=lasttname, ORD_ID=orgId, PROF
[118]|DEBUG|+++++ Command Environment :debug
[118]|DEBUG|+++++ cqrs_AuthenticationCommand.Username :test-uid
[118]|DEBUG|+++++ cqrs_AuthenticationCommand.Password :test-password
[118]|DEBUG|+++++++++++++++++++++++++++++++++++++++++++++++++
[118]|DEBUG|+++++ Command Name :cqrs_WriteResultCommand
[118]|DEBUG|+++++ Command Guid :103e7e6aa5a6466c0c3a42f38fb213d1
[118]|DEBUG|+++++ Command User Context:util_UserContext:[FIRSTNAME=firstname, LASTTNAME=lasttname, ORD_ID=orgId, PROF
[118]|DEBUG|+++++ Command Environment :debug
[118]|DEBUG|+++++ cqrs_WriteResultCommand.theId :test-id
[6]|DEBUG|+++++++++++++++++RESULTs++++++++++++++++++++++++++++
[7]|DEBUG|Command(s) Result Successful?:true
[8]|DEBUG|Command(s) Result:cqrs_CommandResult:[cqrs_Result.resultException=null, cqrs_Result.successful=true]
```

[2] As you see, there is no error checking done

Sample Query

Like the sample command there is a sample query. This example is found in the source tree ***(./scripts/apex/exampleQuery.apex***). Please note, this too is a raw form and can be used in a service supporting specific functionality (i.e., Customer Management, Lead Management, etc.)

```
// set up / arrange
List<blsw.IQuery> queries = new List<blsw.IQuery> {
    // get account by type ()
    new blsw.GetAccountByTypeQuery('Enterprise')
};
Integer inx=1;
// act
Blsw.IQueryResult result= new blsw.QueryDispatcher().dispatch(queries);
// results
System.debug('+++++++++++++++++RESULTs++++++++++++++++++++++++++++');
System.debug('Query(s) Result Successful ?:' + result.success());
System.debug('Query(s) Result Count Found :' + result.results().size());
System.debug('Query(s) Result Searched for: "' + ((blsw.GetAccountByTypeQuery)q
ueries[0]).theUserAccountType() + '"' );
System.debug('+++++++++++++++++RECORDs++++++++++++++++++++++++++++');
// iterate over the results
for ( blsw.AccountTypeRecordsDTO dto: (List<blsw.AccountTypeRecordsDTO>)result.
results() ) {
    System.debug('Query Result (' + inx++ + ') Name=' + dto.name);
}
```

Following the same flow as a command, the above is a collection of (one) query, where we create a Query Dispatcher[3], passes in the query collection, and the dispatcher finds the appropriate handlers and executes.

[3] No error checking, this is sample code only!

Details
[101]|DEBUG+++QueryHandler:++ cqrs_GetAccountByTypeQuery.theUserAccountType :Enterprise
[10]|DEBUG|++++++++++++++++RESULTs++++++++++++++++++++++++++++
[11]|DEBUG|Query(s Result Successful ?:true
[12]|DEBUG|Query(s Result Count Found :33
[13]|DEBUG|Query(s Result Searched for: "Enterprise"
[14]|DEBUG|++++++++++++++++RECORDs++++++++++++++++++++++++++++
[17]|DEBUG|Query Result (1) Name=Ohana, Inc.
[17]|DEBUG|Query Result (2) Name=Advanced Communications
[17]|DEBUG|Query Result (3) Name=Tech Labs
[17]|DEBUG|Query Result (4) Name=Green Fields Media
[17]|DEBUG|Query Result (5) Name=Datanet, Inc.
[17]|DEBUG|Query Result (6) Name=Opportunity Resources Inc
[17]|DEBUG|Query Result (7) Name=Valley Supply Inc.
[17]|DEBUG|Query Result (8) Name=Morpon Brothers
[17]|DEBUG|Query Result (9) Name=Anaco Limited
[17]|DEBUG|Query Result (10) Name=Associated Supply Co.
[17]|DEBUG|Query Result (11) Name=Allied Technologies
[17]|DEBUG|Query Result (12) Name=Optos Inc.
[17]|DEBUG|Query Result (13) Name=UlyssesNet
[17]|DEBUG|Query Result (14) Name=Permadyne GmbH, LTD
[17]|DEBUG|Query Result (15) Name=Employnet
[17]|DEBUG|Query Result (16) Name=Tyconet

Sample Service

The Commands and Queries can be used to form a Service. The service could be Customer Search, Customer Management, Order Management, etc. In this example, we create a Customer Service. The Customer Service just contains a query method; it could contain a sundry of services relevant to Customer Service.

```
//
// Call a Service directly
//
Integer inx=1;
String serviceName='Customer Service';
String accountType = 'enterprise';
// get the service by name
blsw.CustomerService service =  (blsw.CustomerService) blsw.ServiceProvider.ne
wInstance().getService( serviceName);
// show some service information
System.debug('+++++++++++++++++RESULTs++++++++++++++++++++++++++++');
System.debug('Service Name:' + service.name());
System.debug('Service Guid:' + service.guid());
System.debug('+++++++++++++++++RECORDs++++++++++++++++++++++++++++');
//
// iterate over the results
//
for ( blsw.AccountTypeRecordsDTO dto: service.findAccountRecordsByAccountType(
accountType)) {
    System.debug('Service Result (' + inx++ + ') Name=' + dto.name);
}
```

The above Customer Service, **blsw.CustomerService**, uses a provider to return the service by name[4]. The service has similar attributes of a Command or Query (*name, guid, user-context*). Our service method, `findAccountRecordsByAccountType(accountType)` passes in an account-type, value of *enterprise*, and gets back a collection (Data Transfer Object) of Account Type Records[5].

[4] Could overload the Provider to return a service using a Type (instead of name)

[5] The code in the repo does load in Accounts and Contacts if you use the install script; or use *./scripts/genrecord.sh*

Details
[9]|DEBUG|++++++++++++++++RESULTs++++++++++++++++++++++++++++
[10]|DEBUG|Service Name:cqrs_CustomerService
[11]|DEBUG|Service Guid:18f67f5356014076ca2cf5269f9dd8ed
[12]|DEBUG|++++++++++++++++RECORDs++++++++++++++++++++++++++++
[101]|DEBUG|+++QueryHandler:++ cqrs_GetAccountByTypeQuery.theUserAccountType :enterprise
[16]|DEBUG|Service Result (0) Name=Ohana, Inc.
[16]|DEBUG|Service Result (1) Name=Advanced Communications
[16]|DEBUG|Service Result (2) Name=Tech Labs
[16]|DEBUG|Service Result (3) Name=Green Fields Media
[16]|DEBUG|Service Result (4) Name=Datanet, Inc.
[16]|DEBUG|Service Result (5) Name=Opportunity Resources Inc
[16]|DEBUG|Service Result (6) Name=Valley Supply Inc.
[16]|DEBUG|Service Result (7) Name=Morpon Brothers
[16]|DEBUG|Service Result (8) Name=Anaco Limited
[16]|DEBUG|Service Result (9) Name=Associated Supply Co.
[16]|DEBUG|Service Result (10) Name=Allied Technologies
[16]|DEBUG|Service Result (11) Name=Optos Inc.
[16]|DEBUG|Service Result (12) Name=UlyssesNet
[16]|DEBUG|Service Result (13) Name=Permadyne GmbH, LTD
[16]|DEBUG|Service Result (14) Name=Employnet
[16]|DEBUG|Service Result (15) Name=Tyconet
[16]|DEBUG|Service Result (16) Name=Harmon Consulting

Summary

Using CQRS pattern provides a very flexible design. The architectural goals achieved,

- Ease of use
- Simplicity
- Scalability
- Auditability
- Reuse / Packaging
- Performance

Simply define your commands and queries, put them together in a service offering as needed and you can limit the brittleness and fragility you started with from the start. In addition, you can audit, calculate performance, and introduce/inject facets within the chain of commands and queries. Of course, It is not the only way to design a system architecturally, but it provides a smoother adoption.

Appendix: Improvements

A lot of improvements can be made in the design as it was done in a few days. Below is a list of considerations:

- Unit Tests are lacking (even quality Unit Tests).
- Asynchronous Commands behavior (that should not be difficult)
- Resolver functionality can be substituted for Force-DI (as your dependency injector)
- Resolver class does too much and could be simplified.
- Caching can be done to make more performant.
- Cross-Cutting Concerns (CCC) were left out; though, some hooks are in place to extend.
- Use of Unit of Work (UoW)
- Use of Validators to validate data
- Map Engine to resolve SObjects to/from DTOs (or view models)
- Audit Trail
- Replay Ability