BluSoftwave: Force Instrumenter

Command Query Segregation Responsibility (CQRS) with Event Sourcing

Bill Anderson | bill.anderson@blusoftwave.com

Apex

Table of Contents

What is Force Instrumenter?

Force Instrumenter was designed to allow Realtime Observability of your Salesforce Transactions (Apex or Flow). There was no framework or tool that provided a means to allow Realtime inspection of transactions. Either one would eventually get the issues in an email, 24 hours later or not at all.

There was also a lack of discipline in development to Shift Left. Shift left is the practice of moving testing, quality, and performance evaluation early in the development process. Force Instrumenter allows a developer the ability to piece together a transaction with Tracers. These Tracers support Command, Query and Service entities. This helps understand the basic flow, and baseline resources and metrics without performing any heavy development.

At BluSoftwave, we pride ourselves on delivering measurable quality with a proven architectural design. This document walks through elements of Design Patterns used to provide a Shift Left allow Realtime Observability of your Salesforce Transactions.

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 list of to-dos. 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. CQRS originated by Bertrand Myer. Meyer states that every method should be either a query or a command. We take that common design pattern 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.

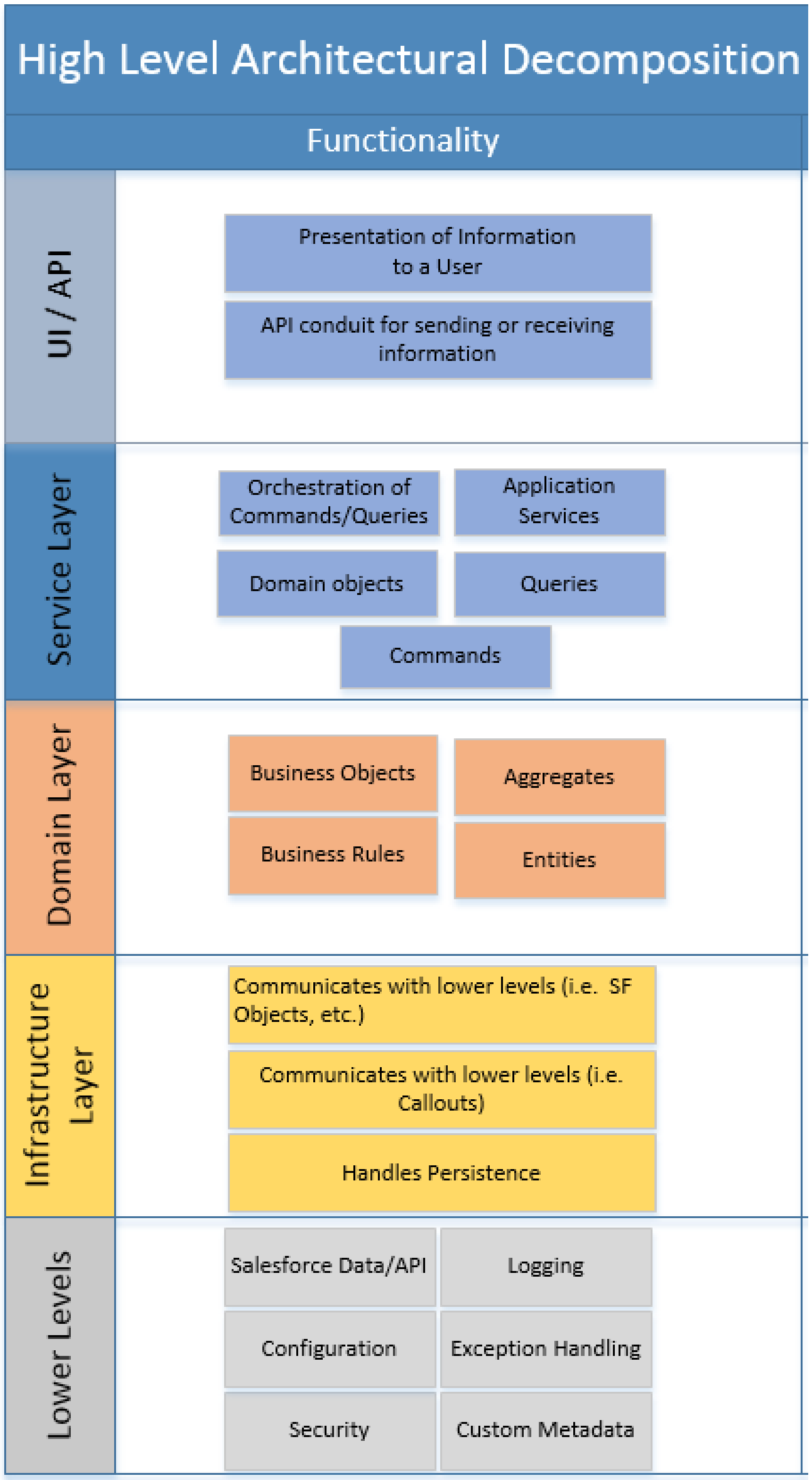


Figure 1 N-tiered Architecture

Our Path to CQRS?

When customers start down the path of customization, i.e., *Apex*, they need to first consider *Clicks, Not Code*[1]. 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/or a Data Transfer Object (DTO).

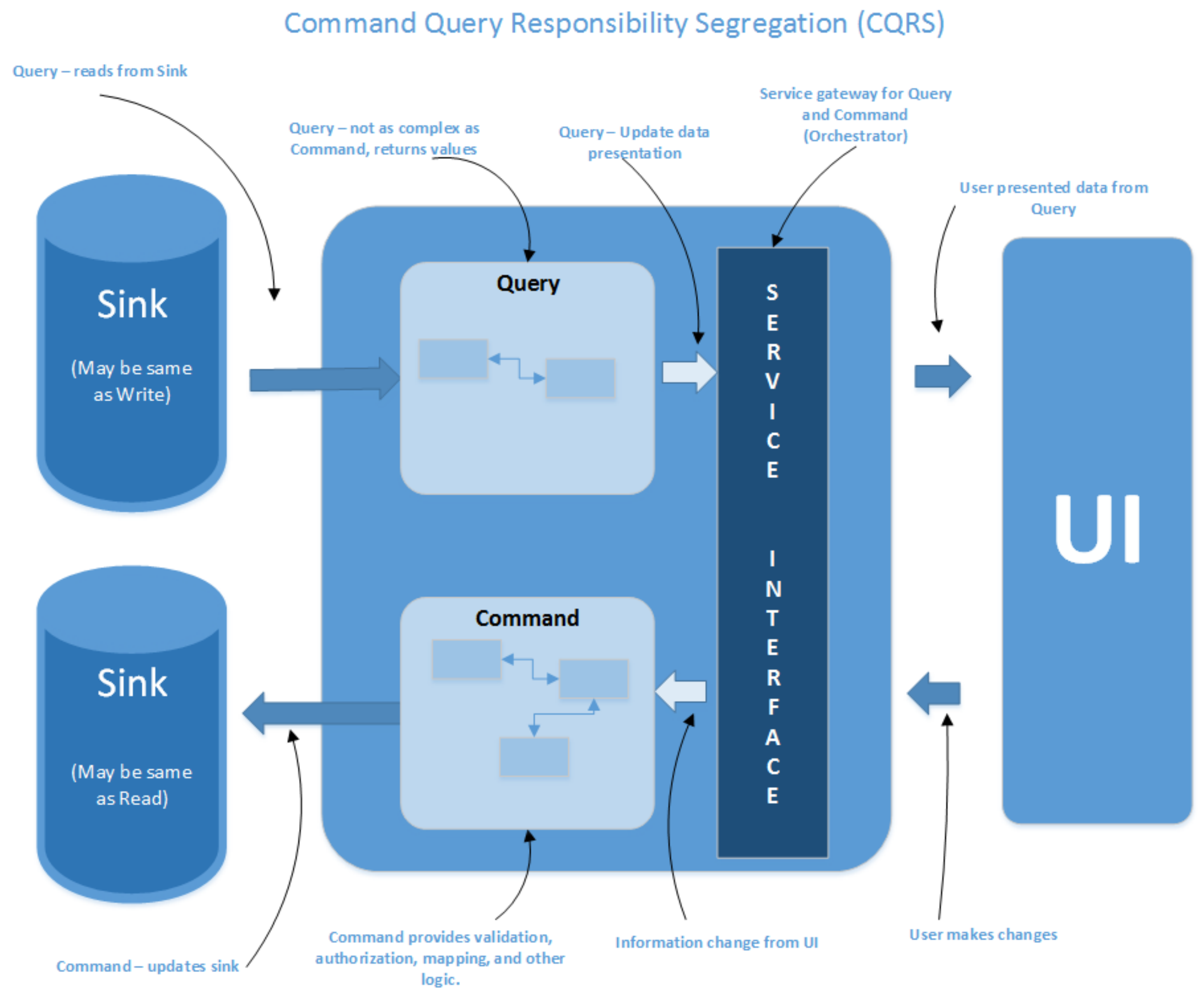
The lines become blurred and difficult to manage; in addition, there is no distinction between data consumed by different personas. For example, a financial customer is often given the same Salesforce Object (i.e., Account, Contact, etc.) as an internal employee, i.e., Financial Planner. This leakage 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 sorts 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 (with Event Sourcing) to the Rescue*!

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

[1] One should also properly architect Clicks as well; however, that too is another conversation.


Command Query Responsibility Segregation (CQRS)
Query – reads from Sink
Query – not as complex as Command, returns values
Query – Update data presentation
Service gateway for Query and Command (Orchestrator)
User presented data from Query
Sink
(May be same as Write)
Query
SERVICE INTERFACE
UI
Command
Sink
(May be same as Read)
Command – updates sink
Command provides validation, authorization, mapping, and other logic.
Information change from UI
User makes changes

Definitions of 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 the 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 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 correct 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 **<u>should always be one Handler mapping to one action</u>**. 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 one 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 of CQRS

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 (when written well)
- 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. Or, define a common naming convention for auto-injection.
- Custom Metadata can become daunting to manage and subject to errors if not managed

Architecture Diagrams

Architectural design references are provided here to assist the relationships discussed below

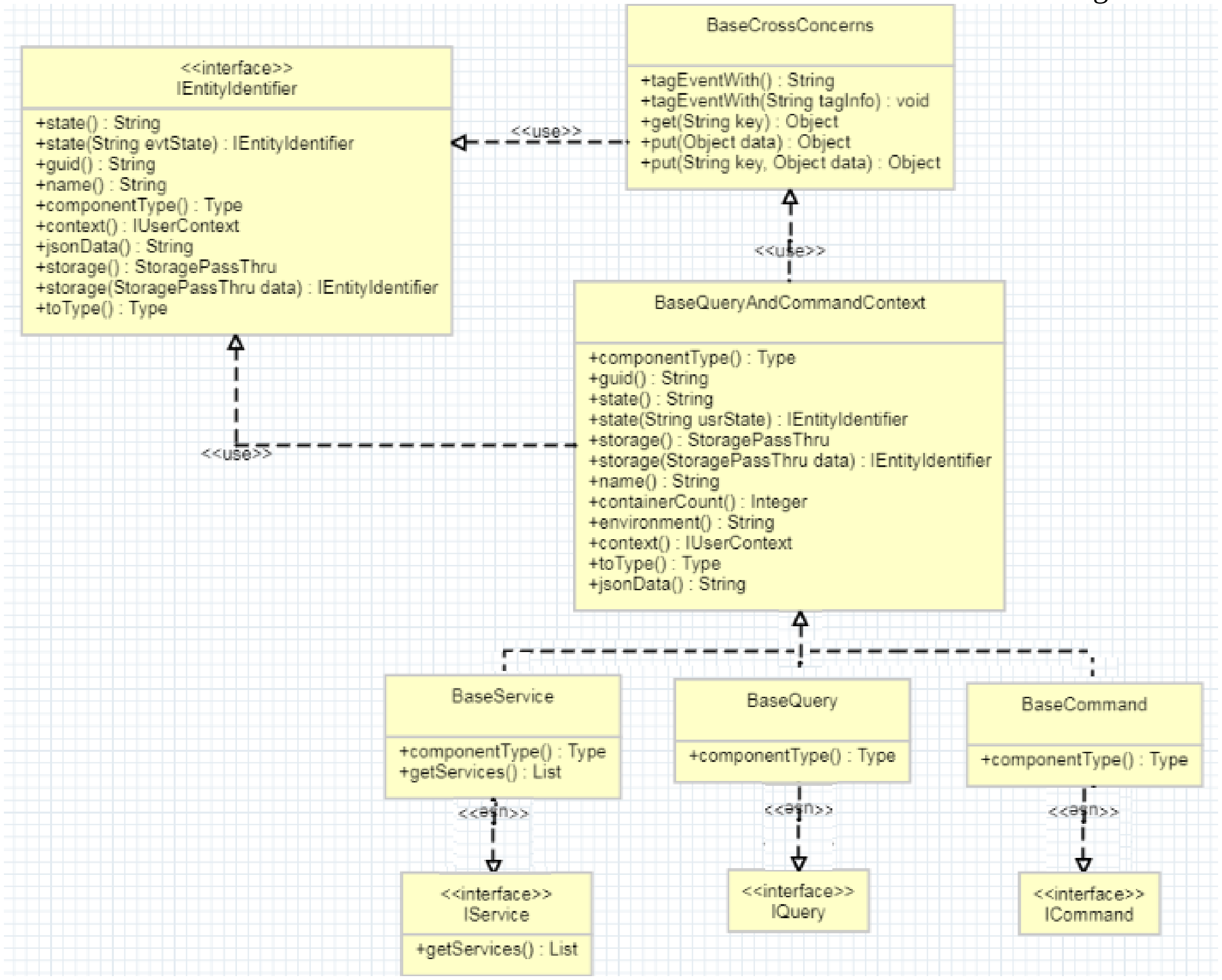


Figure 2 CQS Static Class Diagram

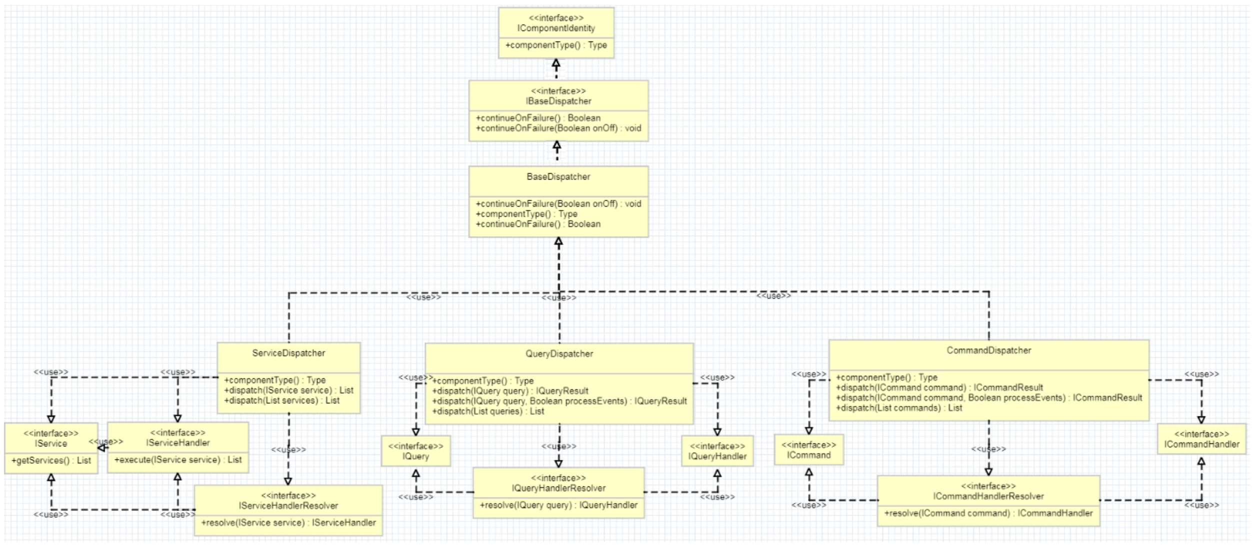

<<interface>>
IComponentIdentity
+componentType() : Type
<<interface>>
IBaseDispatcher
+continueOnFailure() : Boolean
+continueOnFailure(Boolean onOff) : void
BaseDispatcher
+continueOnFailure(Boolean onOff) : void
+componentType() : Type
+continueOnFailure() : Boolean
<<use>>
ServiceDispatcher
+componentType() : Type
+dispatch(IService service) : List
+dispatch(List services) : List
QueryDispatcher
+componentType() : Type
+dispatch(IQuery query) : IQueryResult
+dispatch(IQuery query, Boolean processEvents) : IQueryResult
+dispatch(List queries) : List
CommandDispatcher
+componentType() : Type
+dispatch(ICommand command) : ICommandResult
+dispatch(ICommand command, Boolean processEvents) : ICommandResult
+dispatch(List commands) : List
<<interface>>
IService
+getServices() : List
<<interface>>
IServiceHandler
+execute(IService service) : List
<<interface>>
IServiceHandlerResolver
+resolve(IService service) : IServiceHandler
<<interface>>
IQuery
<<interface>>
IQueryHandler
<<interface>>
IQueryHandlerResolver
+resolve(IQuery query) : IQueryHandler
<<interface>>
ICommand
<<interface>>
ICommandHandler
<<interface>>
ICommandHandlerResolver
+resolve(ICommand command) : ICommandHandler

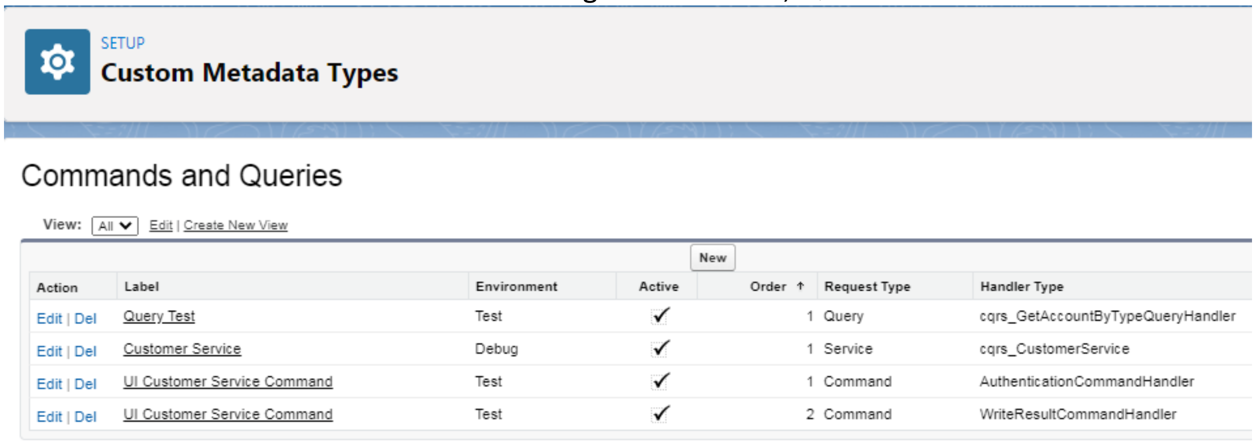
Figure 3 CQS Dispatcher

Resolver

The Resolver provides the Dispatchers (Command, Query and Service) the ability to resolve handlers. There are three resolvers which segregates the functionality of resolving the handlers in the Command, Query or Service Dispatcher. This was done to support single responsibilities and maintenance ease. Each share base functionality. Further discussion will be around Command and Query Dispatcher.

Both the Command Handler Resolver, ***CommandHandlerResolver*** and the Query Handler Resolver***, QueryHandlerResolver,*** inherit from the same base class, ***HandlerResolverBase***[2]. 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 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*.

Further, as it has been mentioned that a Command, Query or Service will have an associated Handler, and there is a Custom Setting option that can be utilize. If on, the Handler does not have to be registered in the Custom Metadata, as the underlying system can find extension (assuming one follows the convention of appending, ***Handler***, to the name). For example, if one

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

creates a Command, *AccountCommand*, there should be a corresponding handler, *AccountCommand**Handler***[3]***.***

HandlerResolverBase

The ***HandlerResolveBase*** wraps the injection of ***IResolver*** in a <u>protected virtual</u> 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*** <u>ONLY</u> needs to know about resolving a handler, ***ICommanHand***ler, for an ***ICommand*.**

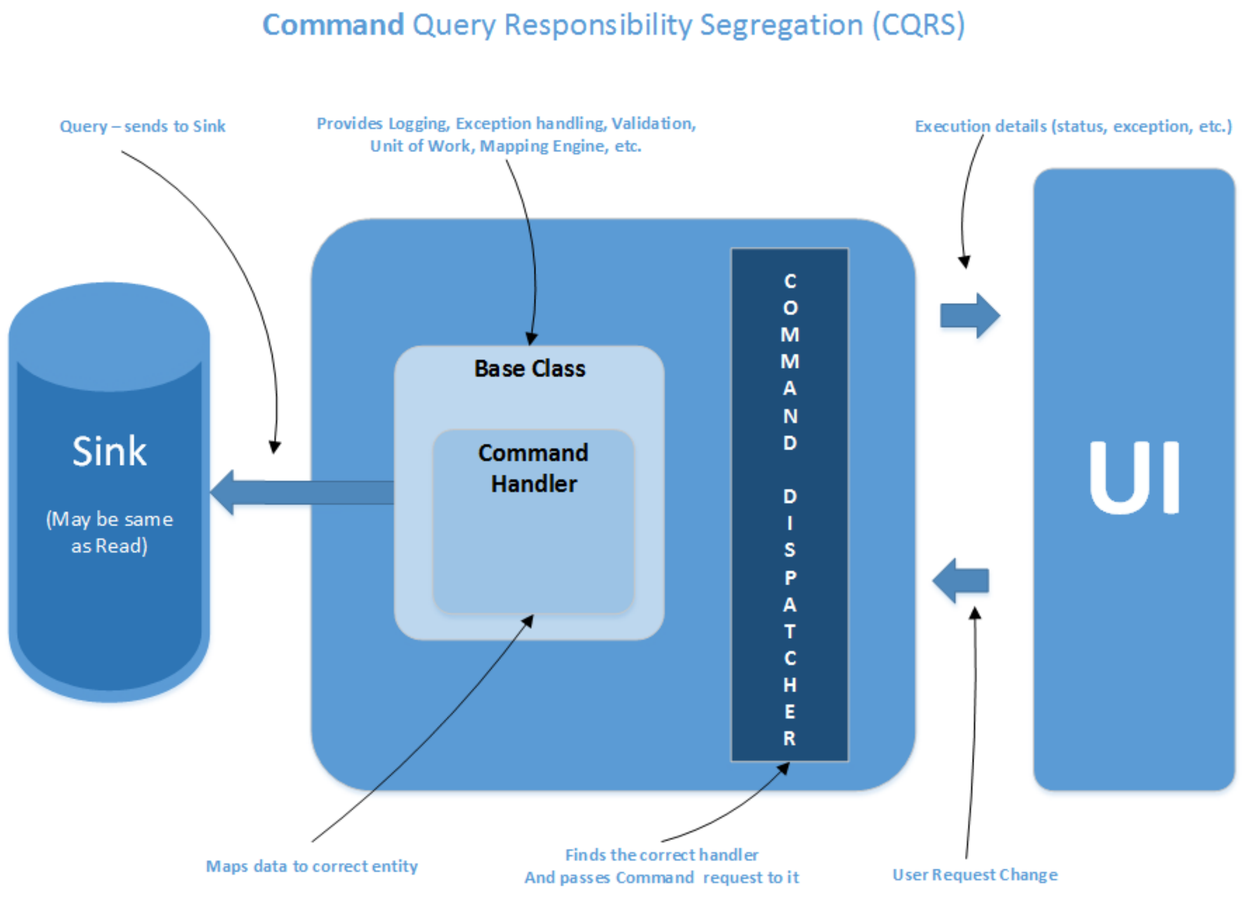
This supports Interface Segregation **(*I*** in SOL***I***D**).**

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 "*<u>Command</u>*" and inherit from ***ICommand***.

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

[3] As noted in the use of this Design Pattern, this causes a proliferation of classes.


Command Query Responsibility Segregation (CQRS)
Query – sends to Sink
Provides Logging, Exception handling, Validation, Unit of Work, Mapping Engine, etc.
Execution details (status, exception, etc.)
Sink
(May be same as Read)
Base Class
Command Handler
COMMAND DISPATCHER
UI
Maps data to correct entity
Finds the correct handler And passes Command request to it
User Request Change

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 common references to,
    - ***Guid* –** Unique Identifier
    - ***Name* –** Name of CQS
    - ***Context* –** User Context
    - ***ComponentType* –** Type of CQS (i.e., Command)

The above information becomes useful for auditing and playback (as 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.

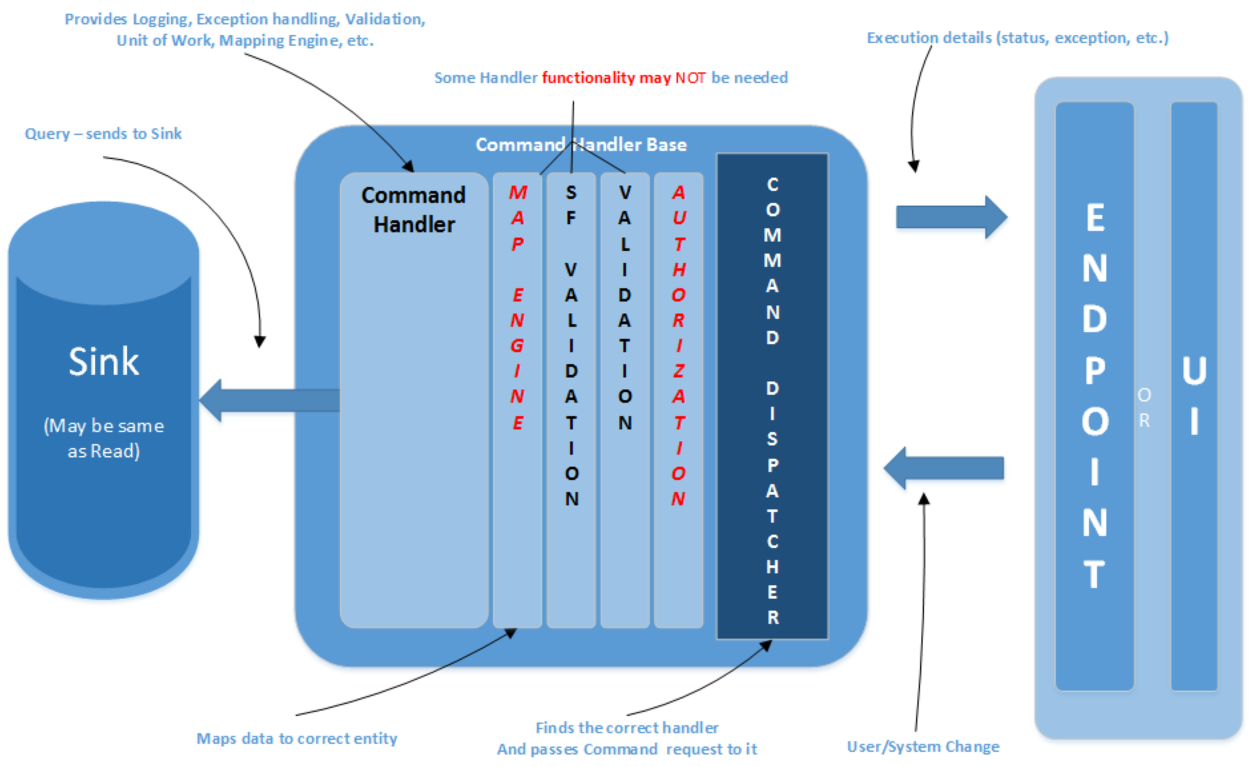
Command Handlers

Command Handlers are usually mapped one-to-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** - a Cross-Cutting-Concern, 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)

Command Query Responsibility Segregation (CQRS) Handler Components


Provides Logging, Exception handling, Validation, Unit of Work, Mapping Engine, etc.
Execution details (status, exception, etc.)
Some Handler functionality may NOT be needed
Query – sends to Sink
Command Handler Base
Command Handler
MAP ENGINE
SF VALIDATION
VALIDATION
AUTHORIZATION
COMMAND DISPATCHER
ENDPOINT
OR
UI
Sink
(May be same as Read)
Maps data to correct entity
Finds the correct handler And passes Command request to it
User/System Change

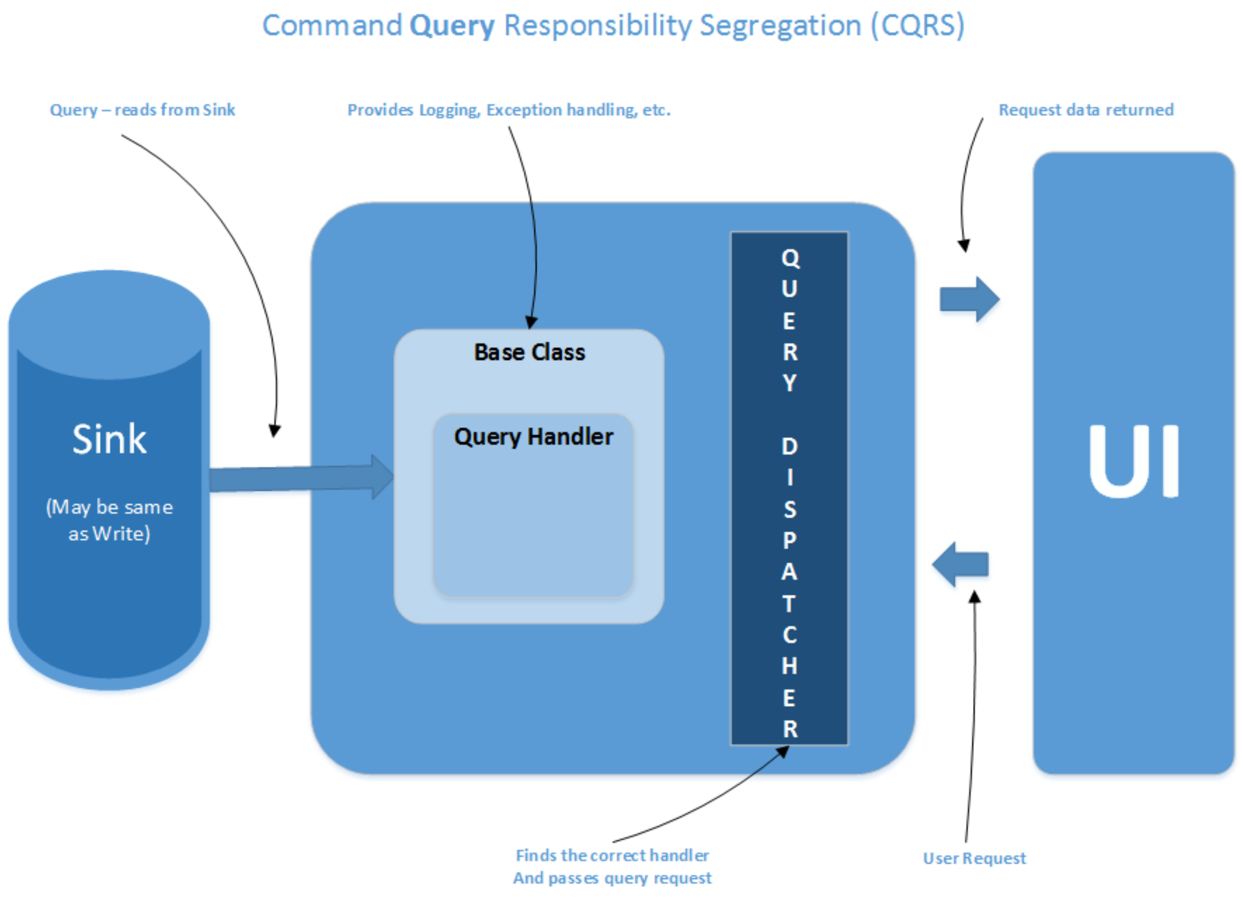
Query - CQRS

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:

- ***IQuery*** – Demarcation of a Query.
- ***IEntityIdentifier –*** Includes references to,
  - ***Guid –*** Unique Identifier
  - ***Name –*** Name of CQS
  - ***ComponentType –*** Type of CQS (i.e., Query)

Providing ample information about a query, 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, the naming convention for all query classes end in "*Query*" and inherit from ***IQuery***.


Command **Query** Responsibility Segregation (CQRS)
Query – reads from Sink
Provides Logging, Exception handling, etc.
Request data returned
Sink
(May be same as Write)
Base Class
Query Handler
QUERY DISPATCHER
UI
Finds the correct handler And passes query request
User Request

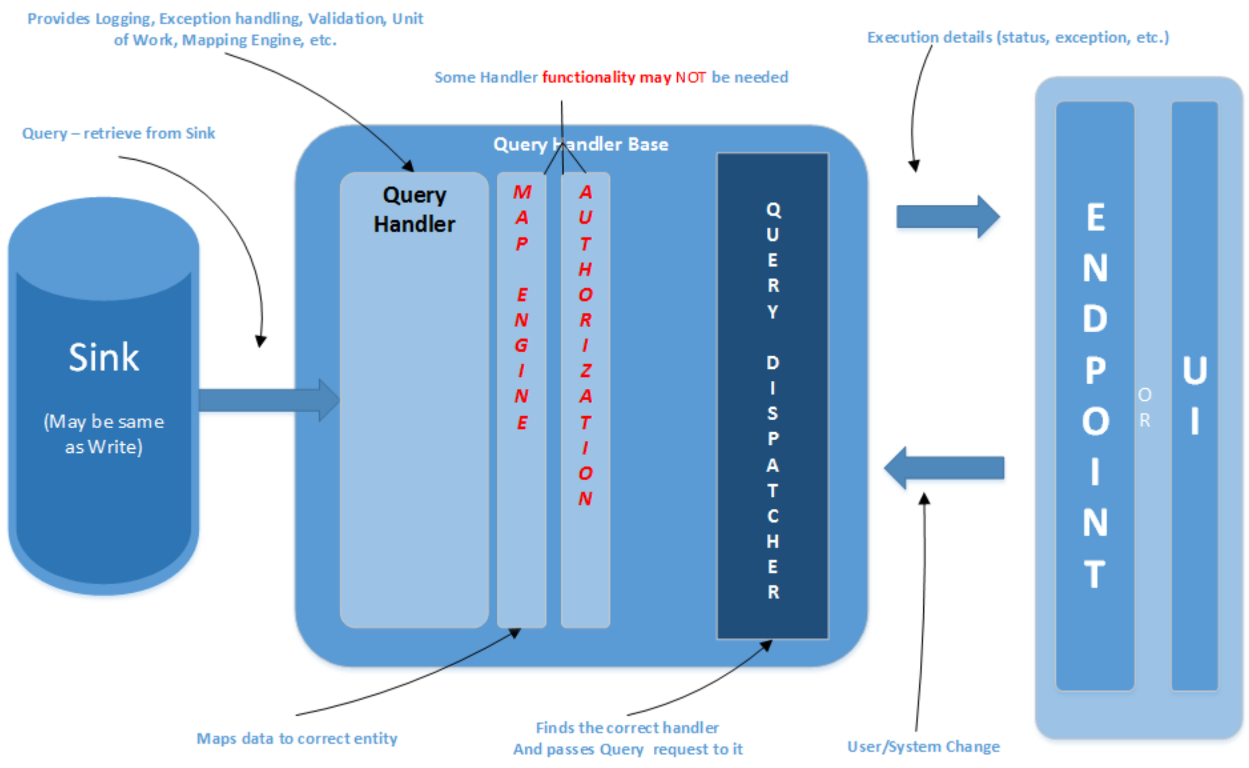
Query Handlers

Query Handlers are usually mapped one-to-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.

Command **Query** Responsibility Segregation (CQRS) Handler Components


Provides Logging, Exception handling, Validation, Unit of Work, Mapping Engine, etc.
Execution details (status, exception, etc.)
Some Handler functionality may NOT be needed
Query – retrieve from Sink
Query Handler Base
Query Handler
MAP ENGINE
AUTHORIZATION
QUERY DISPATCHER
Sink
(May be same as Write)
ENDPOINT
OR
UI
Maps data to correct entity
Finds the correct handler And passes Query request to it
User/System Change

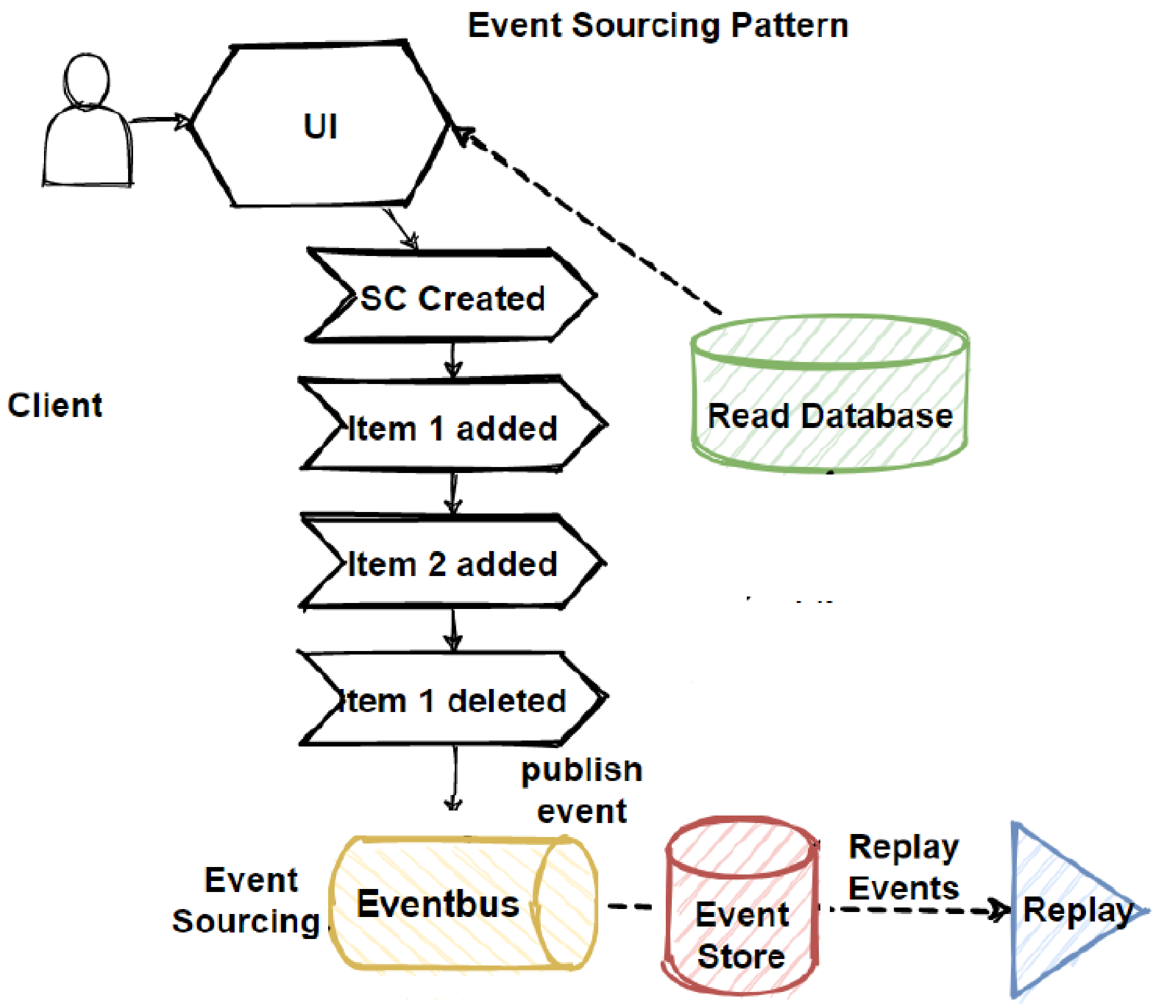
Query Dispatcher

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

CQRS with Event Sourcing

With Event sourcing, application state is stored as a sequence of events. Each event represents a set of changes to the data. The current state is constructed by replaying the events. In addition, event sourcing provides a mean of taking a snapshot of resources (i.e. instrumenting) consumed.

Simple flow scenario shows how operations flow from an Event Sourcing Pattern. CQS items are published and then placed into an Event Store. These events can then be used to replay, analyze and discern behavior (i.e. performance, debug, exceptions, etc.)



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.

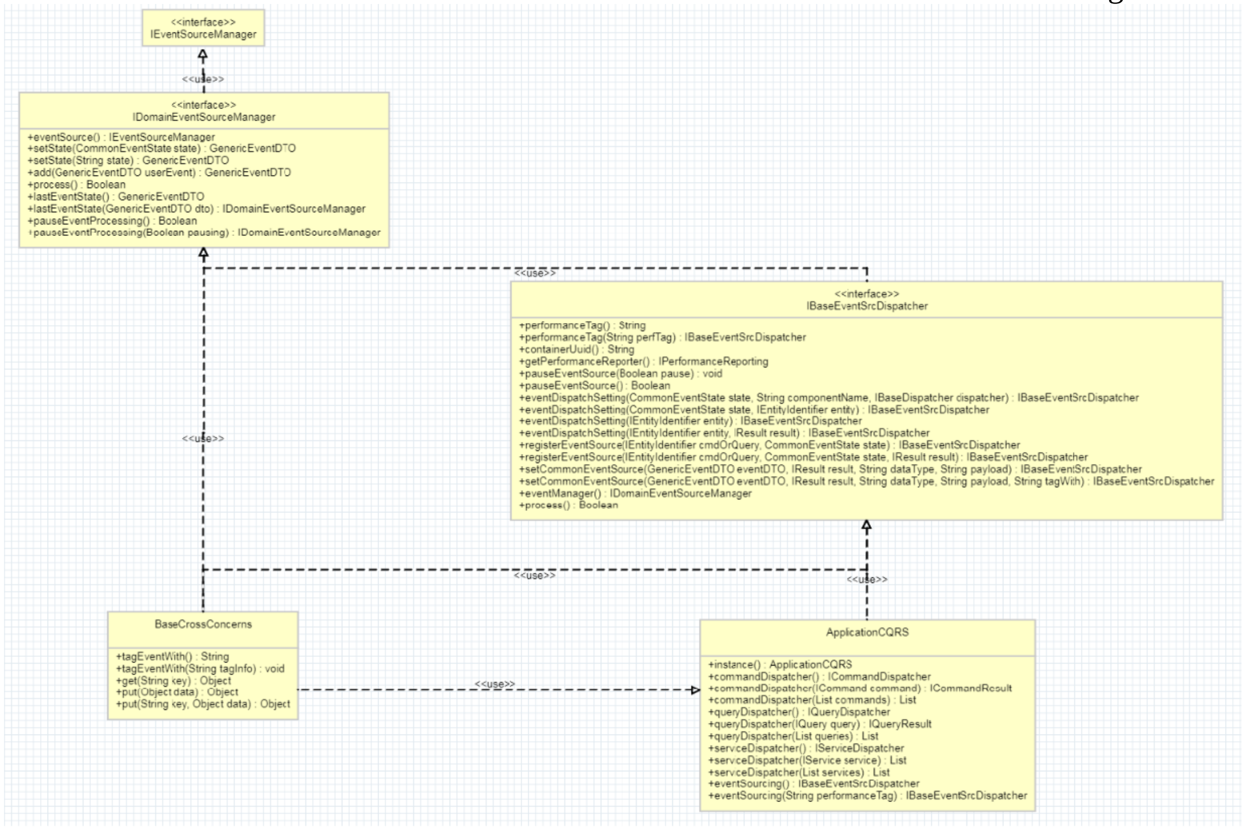

<<interface>>
IEventSourceManager
<<use>>
<<interface>>
IDomainEventSourceManager
+eventSource() : IEventSourceManager
+setState(CommonEventState state) : GenericEventDTO
+setState(String state) : GenericEventDTO
+add(GenericEventDTO userEvent) : GenericEventDTO
+process() : Boolean
+lastEventState() : GenericEventDTO
+lastEventState(GenericEventDTO dto) : IDomainEventSourceManager
+pauseEventProcessing() : Boolean
+pauseEventProcessing(Boolean pausing) : IDomainEventSourceManager
<<use>>
<<interface>>
IBaseEventSrcDispatcher
+performanceTag() : String
+performanceTag(String perfTag) : IBaseEventSrcDispatcher
+containerUuid() : String
+getPerformanceReporter() : IPerformanceReporting
+pauseEventSource(Boolean pause) : void
+pauseEventSource() : Boolean
+eventDispatchSetting(CommonEventState state, String componentName, IBaseDispatcher dispatcher) : IBaseEventSrcDispatcher
+eventDispatchSetting(CommonEventState state, IEntityIdentifier entity) : IBaseEventSrcDispatcher
+eventDispatchSetting(IEntityIdentifier entity) : IBaseEventSrcDispatcher
+eventDispatchSetting(IEntityIdentifier entity, IResult result) : IBaseEventSrcDispatcher
+registerEventSource(IEntityIdentifier cmdOrQuery, CommonEventState state) : IBaseEventSrcDispatcher
+registerEventSource(IEntityIdentifier cmdOrQuery, CommonEventState state, IResult result) : IBaseEventSrcDispatcher
+setCommonEventSource(GenericEventDTO eventDTO, IResult result, String dataType, String payload) : IBaseEventSrcDispatcher
+setCommonEventSource(GenericEventDTO eventDTO, IResult result, String dataType, String payload, String tagWith) : IBaseEventSrcDispatcher
+eventManager() : IDomainEventSourceManager
+process() : Boolean
<<use>>
<<use>>
<<use>>
BaseCrossConcerns
+tagEventWith() : String
+tagEventWith(String tagInfo) : void
+get(String key) : Object
+put(Object data) : Object
+put(String key, Object data) : Object
<<use>>
ApplicationCQRS
+instance() : ApplicationCQRS
+commandDispatcher() : ICommandDispatcher
+commandDispatcher(ICommand command) : ICommandResult
+commandDispatcher(List commands) : List
+queryDispatcher() : IQueryDispatcher
+queryDispatcher(IQuery query) : IQueryResult
+queryDispatcher(List queries) : List
+serviceDispatcher() : IServiceDispatcher
+serviceDispatcher(IService service) : List
+serviceDispatcher(List services) : List
+eventSourcing() : IBaseEventSrcDispatcher
+eventSourcing(String performanceTag) : IBaseEventSrcDispatcher

Figure 4 Salient Classes for Event Sourcing

Caveat-Preemptor

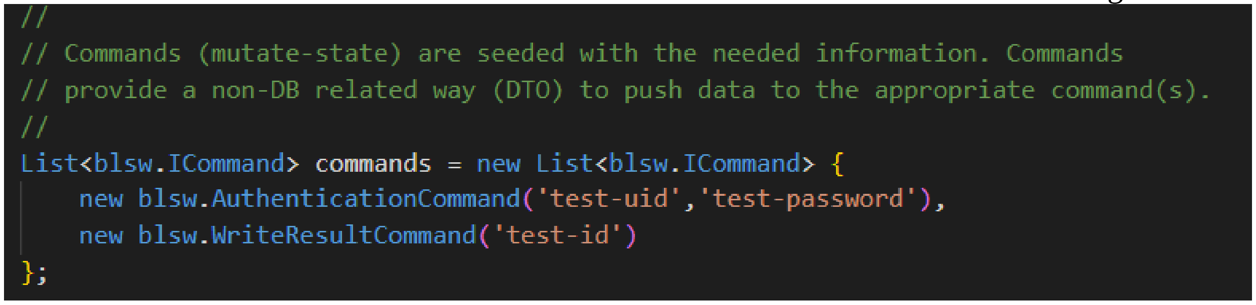
The sample code is just that. The code utilizes Core Force Instrumented namespace, **blsw** and found in Core Force Instrumented Managed Package. The samples are contained in an **Unlocked** Package.

Sample Command

The simplest thing to do is start with a sample command. This example is found in the source tree ***(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.).

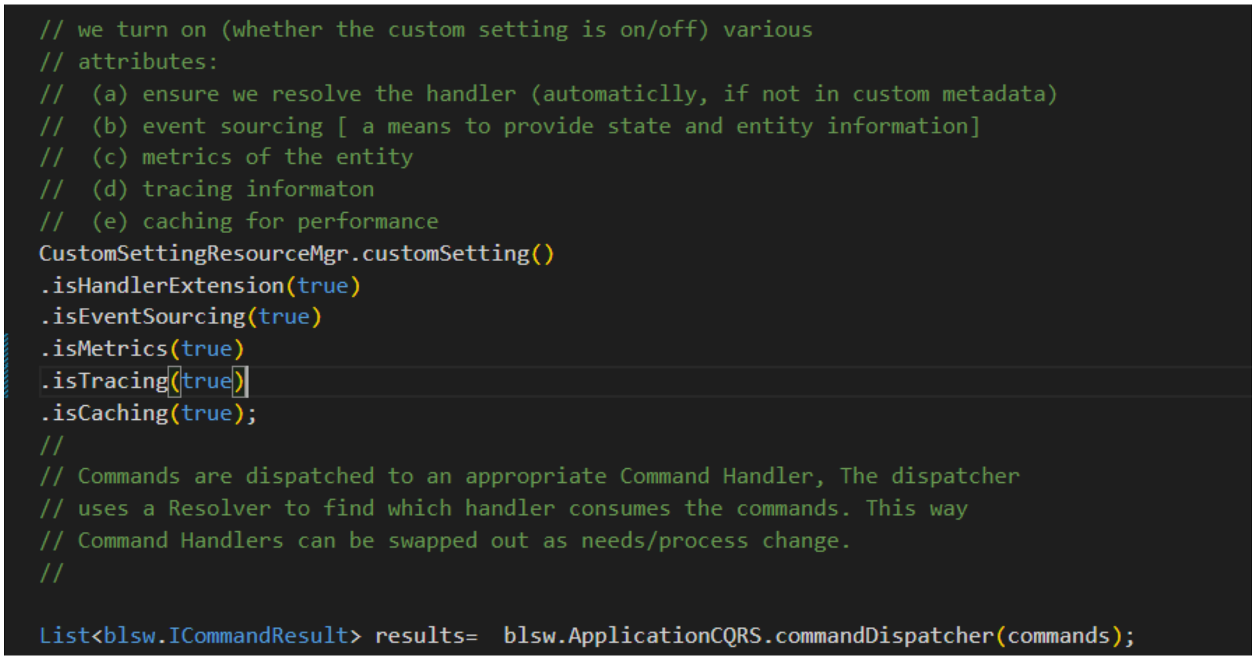
```
// batches up outgoing text
List<String> outputData = new List<String>();
blsw.StopWatch m_watch  = new blsw.StopWatch();

// set the environment we want; otherwise, will default to debug in anonymous mode.
blsw.ApexEnvironment.setEnvironment(blsw.ApexConstants.ALL_CATEGORY);
// DTO == Data Transfer Object
//
// Commands (mutate-state) are seeded with the needed information. Commands
// provide a non-DB related way (DTO) to push data to the appropriate command(s).
//
List<blsw.ICommand> commands = new List<blsw.ICommand> {
    new AuthenticationCommand('test-uid','test-password'),
    new WriteResultCommand('test-id')
};


// we turn on (whether the custom setting is on/off) various
// attributes:
//  (a) ensure we resolve the handler (automaticlly, if not in custom metadata)
//  (b) event sourcing [ a means to provide state and entity information]
//  (c) metrics of the entity
//  (d) tracing informaton
//  (e) caching for performance
blsw.CustomSettingResourceMgr.customSetting()
.isHandlerExtension(true)
.isEventSourcing(true)
.isMetrics(true)
.isTracing(true)
.isCaching(true);
//
// Commands are dispatched to an appropriate Command Handler, The dispatcher
// uses a Resolver to find which handler consumes the commands. This way
// Command Handlers can be swapped out as needs/process change.
//

List<blsw.ICommandResult> results=  blsw.ApplicationCQRS.commandDispatcher(commands);
// how many results do we have
Integer numOfResults        = results.size();

// iterate over the results
for (Integer inx=0; inx < numOfResults; inx++ ) {
    // results
    outputData.add ((inx+1) + ') ++++++++++++++++RESULTs++++++++++++++++++++++++++++'
                    + '\nCommand(s) Result Successful ?      :' + results[inx].success()
                    + '\nCommand(s) Result Exception         :' +
results[inx].theException()
                    + '\nCommand(s) Result Has Exception?    :' +
results[inx].hasException()
                    + '\nCommand(s) Result Type              :' + results[inx].typeOf()
                    );
}
system.debug(string.join(outputData,'\n'));
system.debug('[3]FINAL Duration: ' + m_watch.toString()  );
```

If we narrow down the above sample, we see the created collection of Commands. Each Command provides information (think function arguments) for the specific Command Handler.



```
//
// Commands (mutate-state) are seeded with the needed information. Commands
// provide a non-DB related way (DTO) to push data to the appropriate command(s).
//
List<blsw.ICommand> commands = new List<blsw.ICommand> {
    new blsw.AuthenticationCommand('test-uid','test-password'),
    new blsw.WriteResultCommand('test-id')
};
```

Above, is a collection of commands, that will be passed to the Command Dispatcher.

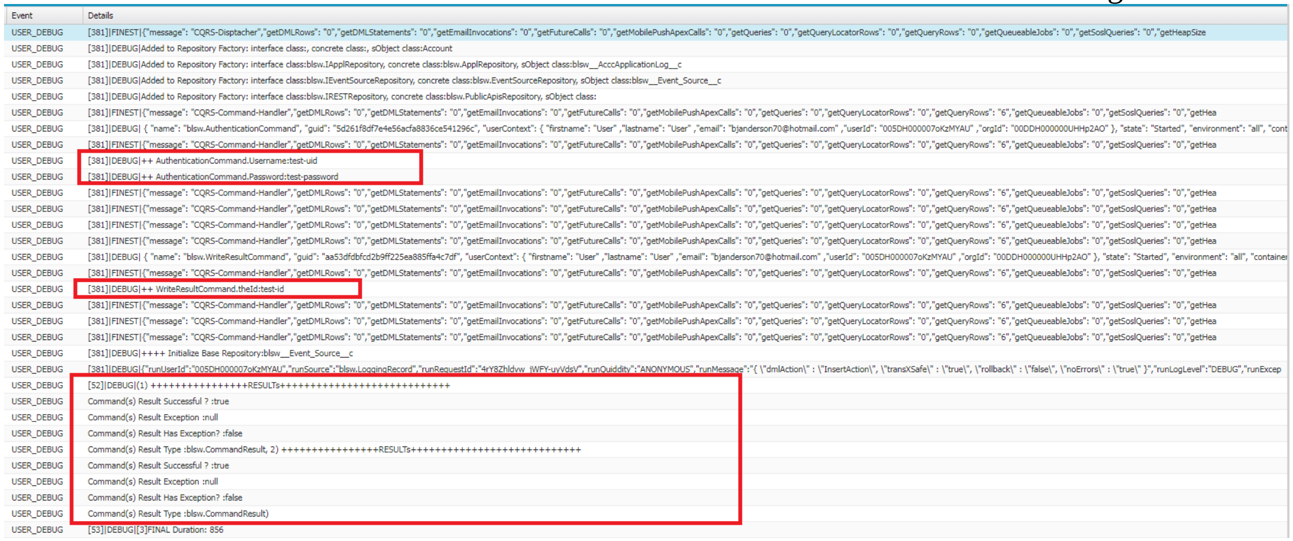


```
// we turn on (whether the custom setting is on/off) various
// attributes:
//  (a) ensure we resolve the handler (automaticly, if not in custom metadata)
//  (b) event sourcing [ a means to provide state and entity information]
//  (c) metrics of the entity
//  (d) tracing informaton
//  (e) caching for performance
CustomSettingResourceMgr.customSetting()
.isHandlerExtension(true)
.isEventSourcing(true)
.isMetrics(true)
.isTracing(true)
.isCaching(true);
//
// Commands are dispatched to an appropriate Command Handler, The dispatcher
// uses a Resolver to find which handler consumes the commands. This way
// Command Handlers can be swapped out as needs/process change.
//

List<blsw.ICommandResult> results=  blsw.ApplicationCQRS.commandDispatcher(commands);
```

Above, you create a Command Dispatcher[4], pass in the command collection, and the dispatcher finds the appropriate handlers and executes. In addition, there are some Custom Settings[5] that can be utilized to control various runtime features.

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

[5] Custom Settings can be defined with Org Defaults, Profile or User level; however, you may need to override.



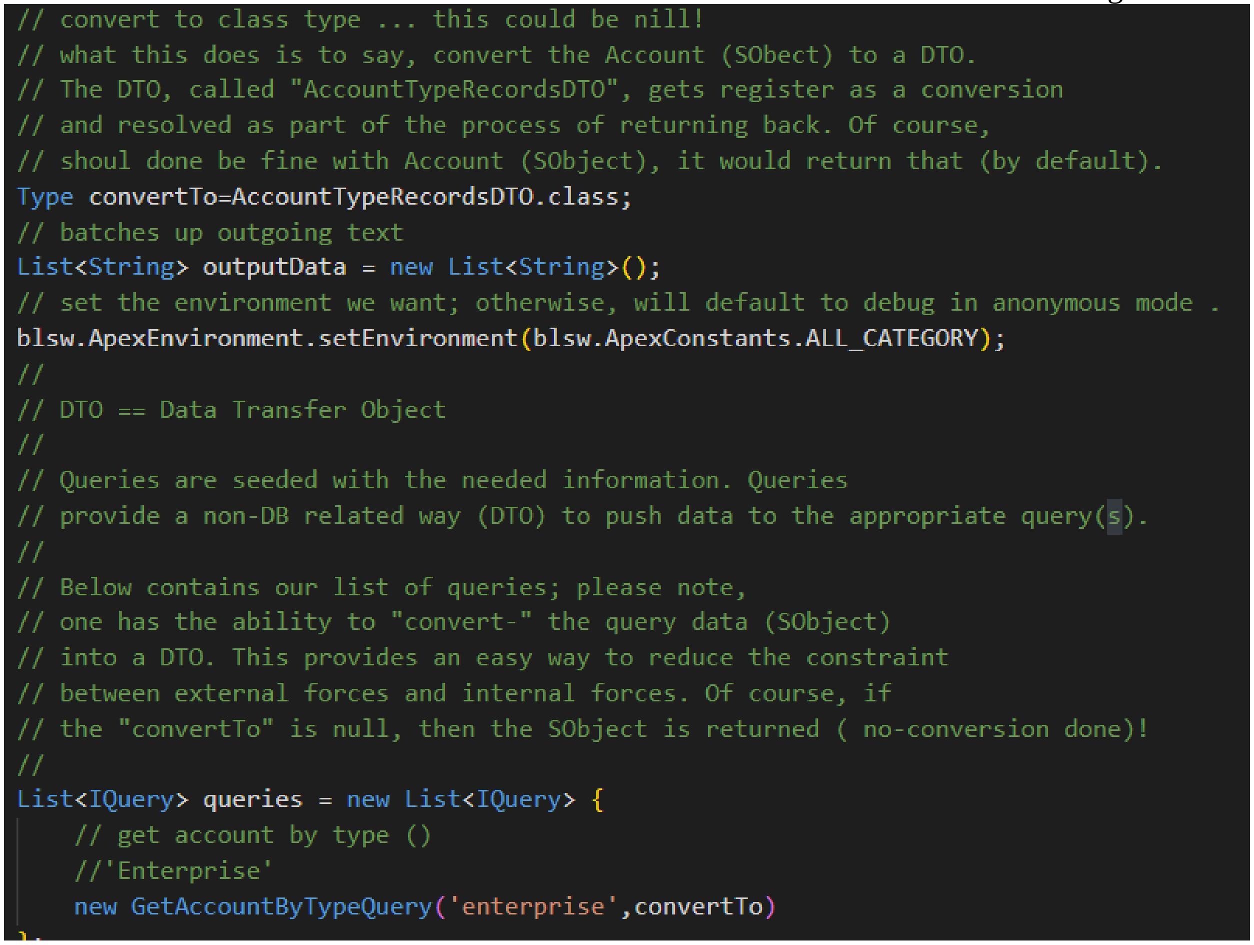
```
Event       Details
USER_DEBUG  [381]|FINEST|{"message": "CQRS-Disptacher","getDMLRows": "0","getDMLStatements": "0","getEmailInvocations": "0","getFutureCalls": "0","getMobilePushApexCalls": "0","getQueries": "0","getQueryLocatorRows": "0","getQueryRows": "0","getQueueableJobs": "0","getSoslQueries": "0","getHeapSize
USER_DEBUG  [381]|DEBUG|Added to Repository Factory: interface class:, concrete class:, sObject class:Account
USER_DEBUG  [381]|DEBUG|Added to Repository Factory: interface class:blsw.IApplRepository, concrete class:blsw.ApplRepository, sObject class:blsw__AcccApplicationLog__c
USER_DEBUG  [381]|DEBUG|Added to Repository Factory: interface class:blsw.IEventSourceRepository, concrete class:blsw.EventSourceRepository, sObject class:blsw__Event_Source__c
USER_DEBUG  [381]|DEBUG|Added to Repository Factory: interface class:blsw.IRESTRepository, concrete class:blsw.PublicApisRepository, sObject class:
USER_DEBUG  [381]|FINEST|{"message": "CQRS-Command-Handler","getDMLRows": "0","getDMLStatements": "0","getEmailInvocations": "0","getFutureCalls": "0","getMobilePushApexCalls": "0","getQueries": "0","getQueryLocatorRows": "0","getQueryRows": "6","getQueueableJobs": "0","getSoslQueries": "0","getHea
USER_DEBUG  [381]|DEBUG| { "name": "blsw.AuthenticationCommand", "guid": "5d261f8df7e4e56acfa8836ce541296c", "userContext": { "firstname": "User" ,"lastname": "User" ,"email": "bjanderson70@hotmail.com" ,"userId": "005DH000007oKzMYAU" ,"orgId": "00DDH000000UHHp2AO" }, "state": "Started", "environment": "all", "cont
USER_DEBUG  [381]|FINEST|{"message": "CQRS-Command-Handler","getDMLRows": "0","getDMLStatements": "0","getEmailInvocations": "0","getFutureCalls": "0","getMobilePushApexCalls": "0","getQueries": "0","getQueryLocatorRows": "0","getQueryRows": "6","getQueueableJobs": "0","getSoslQueries": "0","getHea
USER_DEBUG  [381]|DEBUG|++ AuthenticationCommand.Username:test-uid
USER_DEBUG  [381]|DEBUG|++ AuthenticationCommand.Password:test-password
USER_DEBUG  [381]|FINEST|{"message": "CQRS-Command-Handler","getDMLRows": "0","getDMLStatements": "0","getEmailInvocations": "0","getFutureCalls": "0","getMobilePushApexCalls": "0","getQueries": "0","getQueryLocatorRows": "0","getQueryRows": "6","getQueueableJobs": "0","getSoslQueries": "0","getHea
USER_DEBUG  [381]|FINEST|{"message": "CQRS-Command-Handler","getDMLRows": "0","getDMLStatements": "0","getEmailInvocations": "0","getFutureCalls": "0","getMobilePushApexCalls": "0","getQueries": "0","getQueryLocatorRows": "0","getQueryRows": "6","getQueueableJobs": "0","getSoslQueries": "0","getHea
USER_DEBUG  [381]|FINEST|{"message": "CQRS-Command-Handler","getDMLRows": "0","getDMLStatements": "0","getEmailInvocations": "0","getFutureCalls": "0","getMobilePushApexCalls": "0","getQueries": "0","getQueryLocatorRows": "0","getQueryRows": "6","getQueueableJobs": "0","getSoslQueries": "0","getHea
USER_DEBUG  [381]|FINEST|{"message": "CQRS-Command-Handler","getDMLRows": "0","getDMLStatements": "0","getEmailInvocations": "0","getFutureCalls": "0","getMobilePushApexCalls": "0","getQueries": "0","getQueryLocatorRows": "0","getQueryRows": "6","getQueueableJobs": "0","getSoslQueries": "0","getHea
USER_DEBUG  [381]|DEBUG| { "name": "blsw.WriteResultCommand", "guid": "aa53dfdbfcd2b9ff225ea885ffa4c7df", "userContext": { "firstname": "User" ,"lastname": "User" ,"email": "bjanderson70@hotmail.com" ,"userId": "005DH000007oKzMYAU" ,"orgId": "00DDH000000UHHp2AO" }, "state": "Started", "environment": "all", "container
USER_DEBUG  [381]|FINEST|{"message": "CQRS-Command-Handler","getDMLRows": "0","getDMLStatements": "0","getEmailInvocations": "0","getFutureCalls": "0","getMobilePushApexCalls": "0","getQueries": "0","getQueryLocatorRows": "0","getQueryRows": "6","getQueueableJobs": "0","getSoslQueries": "0","getHea
USER_DEBUG  [381]|DEBUG|++ WriteResultCommand.theId:test-id
USER_DEBUG  [381]|FINEST|{"message": "CQRS-Command-Handler","getDMLRows": "0","getDMLStatements": "0","getEmailInvocations": "0","getFutureCalls": "0","getMobilePushApexCalls": "0","getQueries": "0","getQueryLocatorRows": "0","getQueryRows": "6","getQueueableJobs": "0","getSoslQueries": "0","getHea
USER_DEBUG  [381]|FINEST|{"message": "CQRS-Command-Handler","getDMLRows": "0","getDMLStatements": "0","getEmailInvocations": "0","getFutureCalls": "0","getMobilePushApexCalls": "0","getQueries": "0","getQueryLocatorRows": "0","getQueryRows": "6","getQueueableJobs": "0","getSoslQueries": "0","getHea
USER_DEBUG  [381]|FINEST|{"message": "CQRS-Command-Handler","getDMLRows": "0","getDMLStatements": "0","getEmailInvocations": "0","getFutureCalls": "0","getMobilePushApexCalls": "0","getQueries": "0","getQueryLocatorRows": "0","getQueryRows": "6","getQueueableJobs": "0","getSoslQueries": "0","getHea
USER_DEBUG  [381]|DEBUG|++++ Initialize Base Repository:blsw__Event_Source__c
USER_DEBUG  [381]|DEBUG|{"runUserId":"005DH000007oKzMYAU","runSource":"blsw.LoggingRecord","runRequestId":"4rY8Zhldvw_jWFY-uyVdsV","runQuiddity":"ANONYMOUS","runMessage":"{ \"dmlAction\" : \"InsertAction\", \"transXSafe\" : \"true\", \"rollback\" : \"false\", \"noErrors\" : \"true\" }","runLogLevel":"DEBUG","runExcep
USER_DEBUG  [52]|DEBUG|(1) +++++++++++++++RESULTs++++++++++++++++++++++++++++
USER_DEBUG  Command(s) Result Successful ? :true
USER_DEBUG  Command(s) Result Exception :null
USER_DEBUG  Command(s) Result Has Exception? :false
USER_DEBUG  Command(s) Result Type :blsw.CommandResult, 2) +++++++++++++++RESULTs++++++++++++++++++++++++++++
USER_DEBUG  Command(s) Result Successful ? :true
USER_DEBUG  Command(s) Result Exception :null
USER_DEBUG  Command(s) Result Has Exception? :false
USER_DEBUG  Command(s) Result Type :blsw.CommandResult)
USER_DEBUG  [53]|DEBUG|[3]FINAL Duration: 856
```

The highlighted text[6], in read, shows the execution of the two (contrived) commands.

Sample Query

Like the sample command there is a sample query. This example is found in the source tree (***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.)

[6] The text is only available if Tracing is turned on (i.e., isTracing(true)).



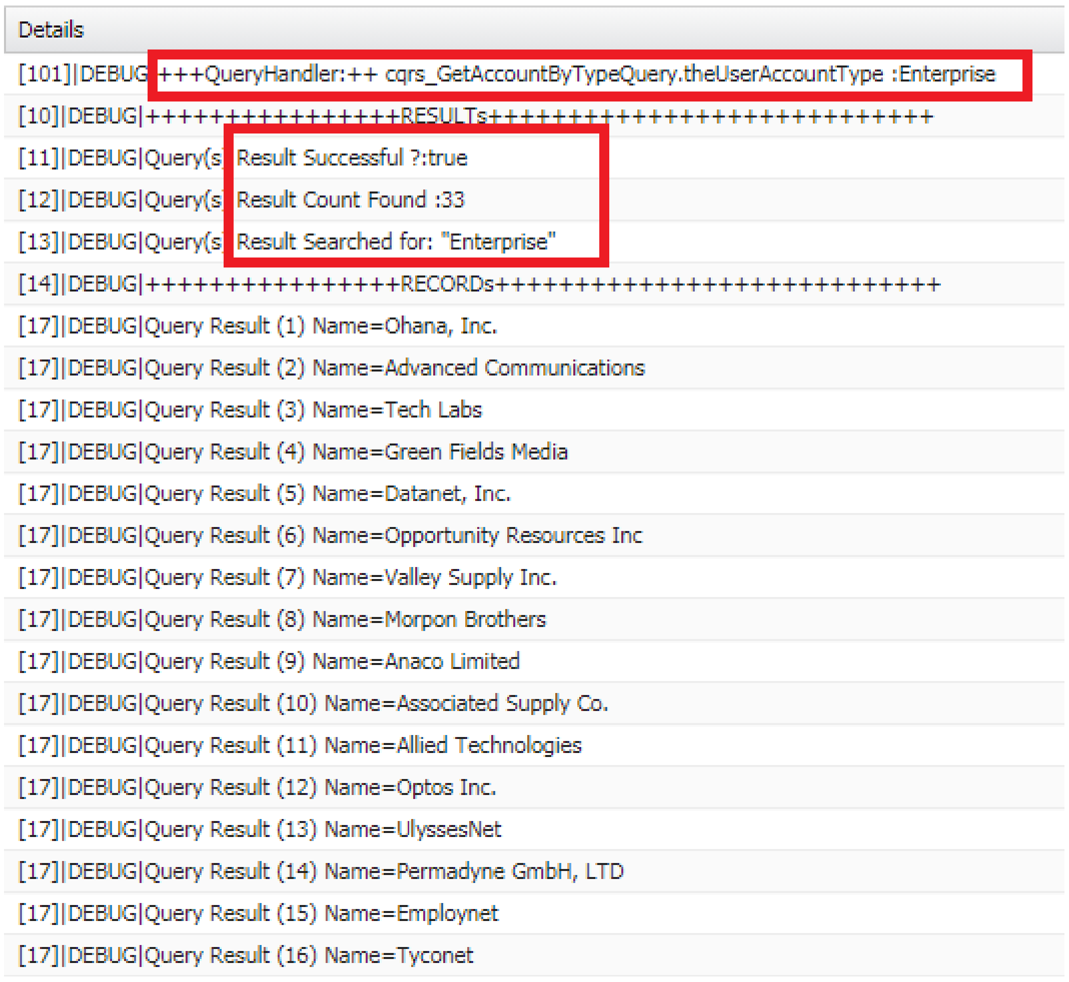
```
// convert to class type ... this could be nill!
// what this does is to say, convert the Account (SObect) to a DTO.
// The DTO, called "AccountTypeRecordsDTO", gets register as a conversion
// and resolved as part of the process of returning back. Of course,
// shoul done be fine with Account (SObject), it would return that (by default).
Type convertTo=AccountTypeRecordsDTO.class;
// batches up outgoing text
List<String> outputData = new List<String>();
// set the environment we want; otherwise, will default to debug in anonymous mode .
blsw.ApexEnvironment.setEnvironment(blsw.ApexConstants.ALL_CATEGORY);
//
// DTO == Data Transfer Object
//
// Queries are seeded with the needed information. Queries
// provide a non-DB related way (DTO) to push data to the appropriate query(s).
//
// Below contains our list of queries; please note,
// one has the ability to "convert-" the query data (SObject)
// into a DTO. This provides an easy way to reduce the constraint
// between external forces and internal forces. Of course, if
// the "convertTo" is null, then the SObject is returned ( no-conversion done)!
//
List<IQuery> queries = new List<IQuery> {
    // get account by type ()
    //'Enterprise'
    new GetAccountByTypeQuery('enterprise',convertTo)
```

Following the same flow as a command, the above is a collection of (one) query, where we create a Query Dispatcher[7], pass in the query collection, and the dispatcher finds the appropriate handlers and executes. Please note the use of the class, *AccountTypeRecordsDTO*. This class a ViewModel. The ***ViewModel*** is a value converter, meaning it is responsible for exposing (converting) the data objects from the model in such a way they can be easily managed and presented to various personas. All Queries provide the ability to define a ViewModel.

In the example above, *AccountTypeRecordsDTO,* represents a view on the Account. Why a ViewModel? There are times when you want to provide a view of a model that reflects what a persona understands and/or allowed to view. It provides a constant abstraction whereby the Model (i.e. Account) can vary without detracting from a persona view. It provides a specialize lens of a Model.

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

Our query method, `findAccountRecordsByAccountType(accountType)` passes in an account-type, value of *enterprise*, and gets back a collection (Data Transfer Object) of Account Type Records[8].

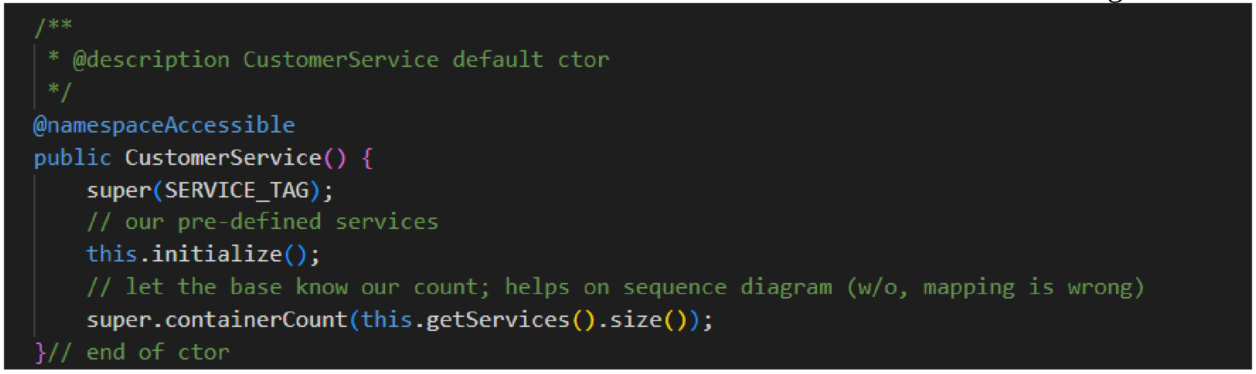


```
Details
[101]|DEBUG +++QueryHandler:++ cqrs_GetAccountByTypeQuery.theUserAccountType :Enterprise
[10]|DEBUG|++++++++++++++++RESULTs+++++++++++++++++++++++++++
[11]|DEBUG|Query(s Result Successful ?:true
[12]|DEBUG|Query(s Result Count Found :33
[13]|DEBUG|Query(s Result Searched for: "Enterprise"
[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

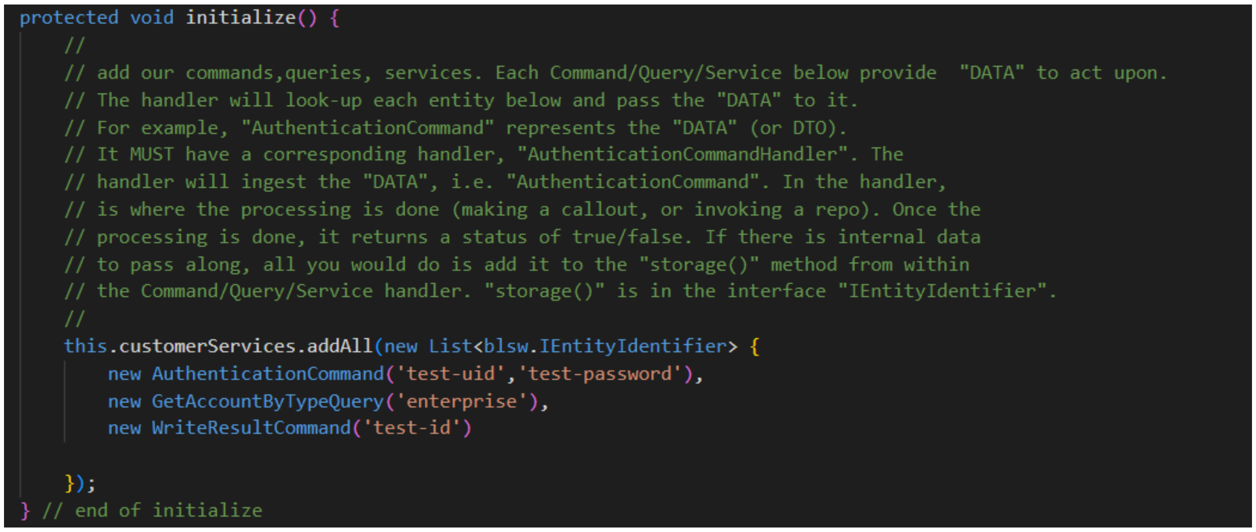
The Commands, Queries and Services can be used to form a single Service. The service could be *Customer Search, Customer Management, Order Management*, etc., or any process that defines a service. In this example, we create a Customer Service. The Customer Service just contains a query and two commands; it could contain a sundry of services relevant to Customer Service.

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



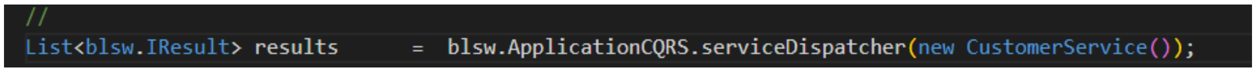
```
/**
 * @description CustomerService default ctor
 */
@namespaceAccessible
public CustomerService() {
    super(SERVICE_TAG);
    // our pre-defined services
    this.initialize();
    // let the base know our count; helps on sequence diagram (w/o, mapping is wrong)
    super.containerCount(this.getServices().size());
}// end of ctor
```

Figure 5 CustomService constructor



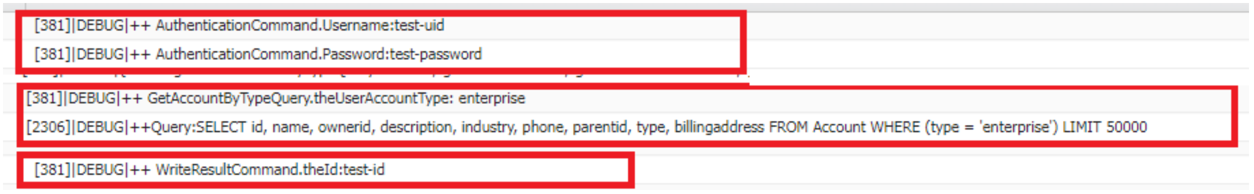
```
protected void initialize() {
    //
    // add our commands,queries, services. Each Command/Query/Service below provide  "DATA" to act upon.
    // The handler will look-up each entity below and pass the "DATA" to it.
    // For example, "AuthenticationCommand" represents the "DATA" (or DTO).
    // It MUST have a corresponding handler, "AuthenticationCommandHandler". The
    // handler will ingest the "DATA", i.e. "AuthenticationCommand". In the handler,
    // is where the processing is done (making a callout, or invoking a repo). Once the
    // processing is done, it returns a status of true/false. If there is internal data
    // to pass along, all you would do is add it to the "storage()" method from within
    // the Command/Query/Service handler. "storage()" is in the interface "IEntityIdentifier".
    //
    this.customerServices.addAll(new List<blsw.IEntityIdentifier> {
        new AuthenticationCommand('test-uid','test-password'),
        new GetAccountByTypeQuery('enterprise'),
        new WriteResultCommand('test-id')

    });
} // end of initialize
```

Figure 6 CustomService Commands and Queries



```
//
List<blsw.IResult> results      =  blsw.ApplicationCQRS.serviceDispatcher(new CustomerService());
```

The above Customer Service, **CustomerService**, uses a resolver to return the service handler. The Service executes each of the elements (Commands, Queries, Services) that make up a Service.



```
[381]|DEBUG|++ AuthenticationCommand.Username:test-uid
[381]|DEBUG|++ AuthenticationCommand.Password:test-password
[381]|DEBUG|++ GetAccountByTypeQuery.theUserAccountType: enterprise
[2306]|DEBUG|++Query:SELECT id, name, ownerid, description, industry, phone, parentid, type, billingaddress FROM Account WHERE (type = 'enterprise') LIMIT 50000
[381]|DEBUG|++ WriteResultCommand.theId:test-id
```

Summary

Using Force Instrumenter (CQRS with Event Sourcing pattern) provides a very flexible design. The architectural goals achieved,

- Ease of use
- Simplicity
- Scalability
- Auditability (with Event Sourcing)
- Replay (with Event Sourcing)
- Reuse / Packaging
- Performance

Simply define your commands, queries, and/or services and place them 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, queries and services. Of course, It is not the only way to design a system architecturally, but it provides a smoother adoption.