CK-Crs has been our first attempt to handle the Command part of CQRS. Some important aspects have been explored like:

* The role of the Ambient Values (as a direct consequence of the endpoint frontier).
* The VISAM codes.
* The “Command can have Result” approach (that is quite different from other CQRS and even DDD command handling approaches).
* The fact that a Command execution can transparently occur:
  + Inline with its reception (simple and direct model, *à la* MVC action).
  + Inside the End Point process but in a background service for long processes or to accommodate a peak load.
  + Out-of-Process, behind a Queue or with any other Distributed Computing fancy stuff.

All these key points are validated, nevertheless, with the advent of the Automatic DI, another way to imagine a CRS implementation is possible. This is the opportunity to rethink some choices.

# The fundamentals

1 - The Command is a Plain Data Object that is of given type (its type being possibly inferred from its shape/fields) that **contains every bits of information required to execute the action**.

2 - A Command may expect a Result. The Result is also a Plain Data Object that must be made available to the sender of the Command, once the Command has been successfully executed.

3 - The sender of the Command is called the Front End. The Front End crafts a Command and sends it to an End Point along with:

* Its Client Identifier, the CallerId that should enable the End Point to call the Front End back with the result of a deferred command (the ‘A’ answer).
* An optional CorrelationId that is a mere string.

4 - The End Point responds to the Front End with a VISAM Response that contains:

* A CorrelationId that may be (or not) the same as the optional incoming CorrelationId.
* The potential Result Data Object.
* A potential Error (that is a Plain Data Object) that when available contains details about ‘V’ or ‘I’ codes.
* The VISAM code:

|  |  |
| --- | --- |
| V | Validation error: the command failed to be validated. It has been rejected by the End Point. This is the equivalent of an http 400 error code. |
| I | Internal error: an error has been raised by the handling of the command. This is the equivalent of an http 500 error code. |
| S | The command has successfully been executed in a synchronous-way, its result is directly accessible by the client in the VISAM Response. |
| A | The execution of the command has been deferred. The VISAM Response does not contain any direct result. |
| M | This distinguishes all meta Result such as Command description, System capabilities, etc. |

5 - The End Point detailed process:

* Detects Meta commands and responds with the appropriate ‘M’ (or may be ‘V’ or even ‘I’) response.
* Validates the Command data against Command’s specific rules (such as ValidationAttributes on the POCO) with the help of ambient knowledge (such as the authentication info associated to the channel from which the Command reached the End Point).
* On failure, a “V” response is returned.
* On success:
  + Analysis of the Command has determined:
    - Whether handling mode is ‘S’ or ‘A’.
    - The Command handler to use.
  + A CommandContext is created that contains:
    - The handling mode.
    - The CallerId.
    - A unique CommandIdentifier.
    - A CorrelationId that may be (or not) the same as the optional incoming CorrelationId
  + On handling mode ‘S’:
    - Command handler is called with the Command and the CommandContext.
      * On error a ‘I’ Response is answered.
      * On success ‘S’ Response is answered with the result (if any).
  + On handling mode ‘A’:
    - ‘A’ Response is answered with the CommandIdentifier and CorrelationId.
    - The command hander execution is deferred by any means.
      * Once executed, the Front End is called back (thanks to the CallerId) with the Result (if any) or the potential error.

# The “Single” Command definition in question

## The Command as a whole

With CRS the Command is the same across the whole system. This was a way for us to “fight the implicit” in the architecture.

The best example is about authentication: the actor identifier (the “ActorId” field) of a command emitted by the frontend for instance must explicitly appear in the command before being sent to the backend, even if the front-to-back channel is bound to a well-known user.

The general principle is that a Command that reaches an End Point (End Points are exposed by the Back End) must contain every data that is required to handle it. The End Point **validates** any possible aspects of the Command data, including data that are (or can be) redundant with any potential “ambient knowledge” such as authentication information (or tenant identifier or culture or application identifier or… whatever you can imagine).

Our experiments have shown that this was a little bit annoying for the developer and that it requires some helpers to handle this: basically these “ambient values” must be made available to the Front End and “automatically” set as much as possible so that the developer doesn’t have to handle them explicitly.

However, even with this negative effect on the front, we maintain our choice: that is a “good constraint” that helps build solid Systems.

Once the Command has reached the End Point, CRS keeps the exact same Command object: it doesn’t make any difference between the “outside” and “inside” of the System. This unified, single, vision of the Command object may appear a little bit too “extreme” and seems to complicate somehow the actual handling of the Command.

What if we introduce a difference between the external and internal command?

## (Hypothesis) Common sense: differentiating external Command and ValidatedCommand

### On the Outside

Nothing should change on the external side: a command that reaches an End Point must be as “complete” as possible (the End Point should not add any information of its own to the command). The responsibility of the End Point is to validate the Command data against any available “ambient knowledge” and potentially expand/derive/project some of the Command data into a different form.

To support this, let’s introduce the notion of ValidatedCommand: the ValidatedCommand has successfully entered the System through an End Point. It is the ValidatedCommand that is being executed, not the “external” Command.

Command

End Point

Ambient Knowledge

ValidatedCommand

### Inside the End Point

In this schema, the End Point acts as a function that transforms the Command into a ValidatedCommand. This transformation includes validation: most of the work here is to apply validation rules on the Command data. In practice, only few real “transformations” are required. The most common example of this is related to Authentication.

An authenticated Command has an “ActorId” field. End Point uses the connection properties (typically the authentication token) to validate that the Command’s ActorId is the same as the connected one. There are now two possibilities:

* The ValidatedCommand has also a simple ActorId field: no transformation is involved.
* The ValidatedCommand has a IAuthenticationInfo AuthInfo field: the request authentication info object must be set, the ActorId field has been “transformed” into the IAuthenticationInfo more complex object.

What eventually matters here is the ValidatedCommand: what if the Command have had a “UserName” or a “UniqueEmail” field? The “binding” would have been different, but the result would have been the exact same ValidatedCommand (well, may be with the addition of the “UserName” or the “UniqueEmail” field).

Under this hypothesis, the real responsibility of the End Point is to bind any external Command models to defined, existing, ValidatedCommand. This is all about Model Binding.

Once a ValidatedCommand is available it has to be handled, may be in another context: ValidateCommand MUST BE serializable. Any part of it must be serializable (just like the basic Command).

In terms of conception, what makes the ValidatedCommand be authored with an IAuthenticatioInfo rather than the simple “ActorId” field?

Should the Command handle the subtleties of the impersonation? If yes, wouldn’t it be better to use two fields: “ActorId” and “ActualActorId”?

### Conclusion

Introducing the ValidatedCommand doesn’t bring any good. The simple, explicit, portable, POCO based, well-defined notion of Command fades out, replaced by a continuum of possible where the Command definition is “diluted” across the architecture.

## The real issue that ValidateCommand aimed to solve is about handling…

The temptation of the “common sense inspired” ValidatedCommand has been driven by an architecture concerns downstream the End Point. Command handlers are standard Services that looks like the following one:

public interface ICommandHandler<in TCommand, out TResult>

{

Task<TResult> HandleAsync( TCommand command, ICommandContext context );

}

These Services can hardly be singletons, most of them must interact with databases or other services that may be bound to an execution context: in the context of the “conforming container” standard DI of .Net, these are Scoped services.

A lot of dependencies, covering a large spectrum of functionalities, may be required by a Scoped Service. One of them is the IAuthenticationInfo: this is an easy way for a Service to obtain the “current user” and we consider this a good thing.

Note: Some may argue that the IAuthenticationInfo is not a Service, that it has nothing to do in the DI. They are not totally wrong: the IAuthenticationInfo is an immutable piece of data that delivers no other services as being itself. Let’s be more “pure” and define a real Service, the IAuthenticationService that exposes a method: IAuthenticationInfo GetCurrentAuthentication()… You see the point? This useless indirection only brings complexity.

The fact is that when a Command is handled by its handler in the Web End Point context, the IAuthenticationInfo is “naturally” available and it is normal for the developer to depend on it (by constructor injection). But, when the handling of the Command is deferred in another context (background service, external process, etc.), the “current user” doesn’t mean anything: the handler SHOULD NOT have been written this way!

The developer should have been more rigorous: The Command exposes an “ActorId” and this is the only piece of information that is required to process it, by no means the full IAuthenticationInfo was necessary!

Unfortunately, this is not totally the developer’s fault since the handler had to rely on other Services or existing API that *de facto* require this IAuthenticationInfo to do its job… the issue is beyond, deeper than, the handler itself.

The temptation to solve this by “enhancing”/”extending” the Command (the ValidateCommand hypothesis) is at best a workaround: the actual domain of the problem is on the Executor/Handler side, not on the Command side!

# Command Handling

Once received by an End Point, a Command is actually a Command and its CommandContext: both can be unified in a very simple ReceivedCommand that can even be modeled by a value type:

public readonly struct ReceivedCommand<TCommand> where TCommand : ICommand

{

public readonly TCommand Command;

public readonly bool AsynchronousHandlingMode;

public readonly string CommandId;

public readonly string CallerId;

public readonly string CorrelationId;

}

A handler is a function that accepts such a ReceivedCommand and returns a Result (or void) if the Command doesn’t expect a Result. This function depends on Services that may be bound to the Execution context (Scoped) or independent (Singletons). Among the Scoped services, some of them are available in the original End Point’s execution context (and not necessarily in all execution contexts).

One of the goals of the architecture is to easily route Commands to be processed inline, in a background service or out-of-process: as soon as the Command execution is not inline, these dangerous “bound to the End Point Services” must be handled:

* At least, we should be able to detect the issue and emit an error, be it at compile time (very unlikely) or a Setup time (more realistic).
* At best, we may generate the full code of the Execution Host…

This may seem quite a crazy objective, but it appears that, thanks to CK-Database, it can be achieved.

The very first thing is to identify these “End Point Bound Services” or, more generally, to categorize the Services at stake between the End Point receiver context and a remote/detached execution context.

To be able to execute a handler, the transitive closure of its dependencies (all its dependent services) must be available. Among them, the Services that interest us are the ones that:

* Like IAuthenticationInfo, their value cannot be known by the remote execution context.
* Like IHttpContext, they must not be used in any execution context other than the receiver’s.
* Like IDatabaseToUseConfiguration, they describe a configuration that is specific to the receiver’s End Point and must be honored, whatever the actual execution context local configuration can be.
* Imagine a mutualized Machine/Process that is dedicated to executing commands from multiple End Points, each of them being logically bound to a different database.

These kinds of Services must be categorized in a way or another. To fulfill the simplest objective (securing the System by detecting these issues), it is enough to detect any use of such Service and raise an error: a single category “ReceiverOnlyService” does the job.

There must be way to mark Services with this “ReceiverOnlyService” or “FrontEndOnlyService”, just like Services can be marked Scoped or Singletons:

* Thanks to a marker interface “IFrontEndOnlyService” for interfaces we define and control.
* By using a configuration similar to <ExternalSingletonTypes /> elements.

The second objective is more complex: to correctly execute the Command handler one need to provide it with substituted Services that “come from” / “act like” the Front End. (This is not always possible and by no means is there a guaranteed way of doing this correctly, but it should be positive for a lot of “standard” scenario.)

First, it is important to note that whatever we achieve in terms of “automatic deferred call behavior” the “FrontEndOnlyService” is still a must: some Services like the IHttpContext can NOT be used in a detached execution context. Among the three sample Services discussed above, two of them remain:

* Like IAuthenticationInfo, their value cannot be known by the remote execution context.
* Like IDatabaseToUseConfiguration, they describe a configuration that is specific to the receiver’s End Point and must be honored, whatever the actual execution context local configuration can be.

If they are semantically slightly different, they share an important aspect: they are more “Values” than real “Service”. And this makes total sense: code is (and must be) the same between the Front End and any detached execution context, data is obviously the varying factor here. So… we must “transfer data” form the Front End to the execution context: this is simply about serialization.

To summarize, there are 3 kinds of Services:

1. Normal, acting, Services (like the IEmailSender) that are pure “Code”. They must be available everywhere a piece of code needs them. This is basically the life of any DI based architecture nowadays. Nothing to do here.
2. Services that must be “serialized”, “marshaled” across the System, alongside with the Command that must be executed.
3. Services like IHttpContext that, even if they may be technically present in the detached execution context must not be used because they are bound to the infrastructure, they have nothing to do regarding the Command processing itself.

The “serializable” category is the more subtle. One can, at least, make one distinction among such Services:

* Services that carries configuration specific to the Front End and/or the Command are related to the “Ambient knowledge” of an End Point.
  + They should be easy to spot: authentication, tenant information, “current application”, “current culture translator”, etc.
* Services that brings any kind of “configuration” into the system, that are not directly, obviously, bound to any Command or business aspect.
  + Those may be harder to spot. Some of them may even be schizophrenic: a part of it should be the “locally available/defined” data and another part should be the “from the Front End” data…

The fact that this could be a difficult issue should not be a barrier. On the contrary, it should allow us to highlight ambiguities, to force us to clarify the functioning of certain processes and implementations. Consider the “schizophrenic” hypothesis for instance, this smells bad: this “Service” (whatever it is) is certainly a good candidate to the next refactoring session as it obviously mixes different concerns…

To conclude, with two categories we should be able to reach our highest objective:

* **FrontEndOnlyServices**: such service can NOT be used by any Command handler.
* **FrontEndAmbientServices**: such service MUST be serializable and be marshaled to any remote Command handler (along with the ReceivedCommand) that uses it.

===================== TODO =======================

Ideas:

* The (best) Handler is on the “Reality side”
* On the “Reality Side” (Primary Run)
  + Collects all the TCommand : ICommand (IClosedPoco) objects that have a concrete existing handler that implements ICommandHandler<TCommand> : IAutoService
  + These handlers are Services that must expose one or two methods:
    - Task<MyResult> HandleAsync( ReceivedCommand<MyCommand> command, …(any services)… )
    - MyResult Handle( ReceivedCommand<MyCommand> command, …(any services)… )

Or, if the Command has no Result:

* + - void Handle( ReceivedCommand<MyCommand> command, …(any services)… )
    - Task HandleAsync( ReceivedCommand<MyCommand> command, …(any services)… )
  + Note that Command that have NO associated handlers are ignored (a warning must be emitted).
  + Each of these handlers may be Singleton or Scoped (depending on the constructor parameters).
  + This is a CommandReceiver:
    - interface ICommandReceiver<T> where T: ICommand
    - {
    - bool IsFakeSync { get; }
    - bool IsFakeAsync { get; }
    - Task<VISAMResponse> HandleAsync( IServiceProvider sp, in ReceivedCommand<T> command )
    - VISAMResponse Handle( IServiceProvider sp, in ReceivedCommand<T> command )
    - }

Note that the CommandReceiver encapsulates/hides the Result aspect of the command (the API is unified).

* + For each Handler we can generate the Receiver that executes the Handler. This Receiver depends on the Handler (hence it shares its lifestyle):
    - Class ExecutorForMyCommand : ICommandReceiver<MyCommand>
    - {
    - readonly FinalHandlerForMyCommand \_handler;
    - Executor( FinalHandlerForMyCommand handler )
    - {
    - \_handler = handler;
    - }
    - bool IsFakeSync { get; } 🡸 True if no synchronous Handle exists on the FinalHandlerForMyCommand.
    - bool IsFakeAsync { get; } 🡸 True if no nsynchronous HandleAsync exists on the FinalHandlerForMyCommand.
    - Task<VISAMResponse> HandleAsync( IServiceProvider sp, in ReceivedCommand<T> command )
    - {
    - Here we must analyze the HandleAsync parameters and resolve the Services parameters (required or optional when null default) to generate calling code.

If the handler has no HandleAsync method (IsFakeAsync is true), the code must call the synchronous Handle and returns the Task.FromResult.

* + - }
    - VISAMResponse Handle( IServiceProvider sp, in ReceivedCommand<T> command )
    - {
    - Same as above except if the handler has no Handle method (IsFakeSync is true), the code must call the asynchronous HandleAsync and GetAwaiter.GetResult() on it.
    - }
    - }

Note that this Executor CAN be the specialization of the Handler with these 2 methods be overrides of a “abstract class CommandHandler<T> where T : ICommand” base class…

De facto, a CommandHandler<T> is the ultimate CommandReceiver<T>, this is why we can model this:

abstract class CommandHandler<T> : ICommandReceiver<T> where T : ICommand   
{

Task<VISAMResponse> ICommandReceiver.HandleAsync( IServiceProvider sp, in ReceivedCommand<T> command ) => DoHandleAsync(sp, command );

VISAMResponse ICommandReceiver.Handle( IServiceProvider sp, in ReceivedCommand<T> command ) => DoHandle(sp, command );

[ImplementMe]

protected abstract Task<VISAMResponse> DoHandleAsync( IServiceProvider sp, in ReceivedCommand<T> command );

[ImplementMe]

protected abstract VISAMResponse DoHandle( IServiceProvider sp, in ReceivedCommand<T> command );

}

This is NOT a good idea: the Handler must be independent of the infrastructure. If we do this, the Handler cannot be tested independently and HAS TO be CKSetup. This is BAD.

* + Another ICommandReceiver can be generated: the one that marshalls the call to another context. They may be actually 2 of them:
    - The Background marshaller
    - The Out-of-Process marshaller
  + Now, depending on the BinPath, we must handle a Command with one of the Receivers.
  + Note that if a Command must NOT be available in a BinPath, the <ExcludedTypes> of the Configuration must be used.
  + In some BinPath we have no other choice than marshalling the call: The Command exists (its Type is not excluded) but the final Handler is not available (recall that the Handler is on the Reality side of the System).
  + In some BinPath, we are free to handle the command locally or marshal it to another context.

# The Fire&Forget or Publish thing

A CRS command is a POCO based on a class that has no type constraint (there is no CommandBase class). A CRS Command can even declare an associated/expected Result by simply hold a nested class named “Result”. However, an ICommand<TResult> is available to simplify pipeline builder methods implementation (and understanding).

Cris adopts a different approach that is aimed to make Command definition across the System more “modular”. A Cris Command object must support a defined ICommand that is a IClosedPoco. A Command that expects a Result must support ICommand<TResult>.