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, see the excellent analysis here: <https://vladikk.com/2017/03/20/tackling-complexity-in-cqrs/>).
* 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()… Did we make any progress here? My point is: this “true Service” is a totally useless indirection that only brings complexity. (This indirection should be used for any piece of information that has to be created, that is not necessarily known up-front but not for such a priori known information.)

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:

1. [**Basic Objective**] 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).
2. [**Ultimate Objective**] At best, we may generate the full code of the Execution Host…

This may seem quite crazy objectives, 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 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.
* 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” or FrontEndONLYService does the job. There must be way to mark Services with this “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.

This introduces a new category about Services that we call the “FrontType”.

# The Service “FrontType” category

Introducing this FrontEndOnlyService category is easy and is enough to achieve our Basic Objective that is to secure the background service execution…

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” from 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. This initial Service instance is created by the End Point (there is no other place to resolve/instantiate/know it) so they are somehow “FrontEndService” but not “FrontEndOnlyService”, they are “MarshallableService” AND on the “Front” side.
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 “marshallable” 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.

A typical bad smell would be a Service that conveys values driven by the user context (any kind of user info) or front configuration but also infrastructure related data such as a backend IP address… This “schizophrenic” service 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 Ultimate Objective:

* By marking some services with **IFrontAutoService** (final name of the **FrontEndOnlyServices**), we can control that they cannot be used by any command handler.
* By enabling some of them to be **Marshallable**, we can offer desynchronized command processing by handling the transfer of the required front related information to the background process (the ReceivedCommand and the marshallable services must be serialized).

# The FrontService virality.

The “Marshallable” aspect cancels the “Front” aspect: once a IFrontAutoService is marshallable, it is no more “Front Only” and can be used by background processes (as long as it is marshalled of course).

The virality of the IFrontAutoService marker takes this into account: any Service that depends on one or more IFrontAutoService **that are not marshallables** is also a IFrontAutoService.

This propagation is computed by analyzing the constructor parameters of the final, most specialized, service type (this doesn’t use the closure of the dependencies that is used to resolve the service dependency graph).

# What is a Marshallable Service?

## The eventually required Model.IMarshaller<T>.

To be marshallable, a marshaller for the service must exist. An implementation of the CK.StObj.Model.IMarshaller<T> interface must be available (resolvable in the DI container) for each marshallable service[[1]](#footnote-1).

public interface IMarshaller<T>

{

/// <summary>

/// Writes any information to the binary writer that Read(ICKBinaryReader) will use to

/// instanciate a copy of the service.

/// </summary>

/// <param name="writer">The writer.</param>

/// <param name="service">The service to marshall.</param>

void Write( ICKBinaryWriter writer, T service );

/// <summary>

/// Reads previously written data and recreates a service instance.

/// </summary>

/// <param name="reader">The binary reader to use.</param>

/// <returns>The marshalled service.</returns>

T Read( ICKBinaryReader reader );

}

Note: The marshall/unmarshall process (simply called Write/Read here) may be a simple value serialization/deserialization, or may result in the creation of a proxy/stub or any other artefacts. This is where Marshalling differs from Serialization.

We have decided to consider that being Marshallable is a “claim“: one CAN perfectly be Marshallable without any available Marshaller… as long as no marshalling is required. In the other way around, we decided to consider that the existence of a CK.StObj.Model.IMarshaller<T> auto service[[2]](#footnote-2) *de facto* declares that T is marshallable (even if T is not marked by any marker interface nor externally declared). This is the reason why the final interface has no constraint at all on its T parameter (there is no type constraint **where T : IAutoService**).

## The IsMarshallable claim.

Question: Is a IMarshaller<T> where T is an interface able to marshal any implementation of it?

This question applies to any abstraction (marshalling any type that would be “above” the real, final type). And the funny thing is that this question has no absolute answer: it depends on the very nature of the interface/abstraction AND the way it is implemented.

However, a safe answer can be provided here: only the final, most specialized type is really marshallable. Above it, all marshallers are liars!

Based on this remark:

* We MUST NOT propagate the IsMarshallableService flag to specialized types: it solely applies to the exact targeted type.
* The idea of using an interface to claim the IsMarshallable aspect must be rejected: The IsMarshallable aspect is better modeled with a non-inheritable Attribute (that targets only classes).

# The case of the Marshallable Singleton

A marshallable singleton is much “stranger” to envision than a marshallable scoped service. Nearly all marshallable services that come to mind are naturally scoped: they are bound to the user (AuthenticationInfo for instance), to the currently connected context (a tenant, a culture, etc.).

What can be “Marshallable Singletons”? Like other marshallable services, they are more “Data” than “Code”, and their data is necessarily independent of any user/connection context, their data must be dependent of the “Front”, the End Point itself: this is typically a configuration that must be used by the remote/detached context to correctly handle the command.

Eventually, Marshallable Singletons are useful when and only when multiple Fronts share a common backend executor and that the execution of a Command depends on the Front (or more precisely depends on a configuration of the Front) that received it.

This exhibit the fact that if we allow multiple Fronts to share a common background executor, the background DI topology is not the same as the front one: the ApplicationServices are the “really shared” services, then comes a layer of Singletons per Front, and then comes the Scoped ones. This is like what must be deployed in a multi-tenant architecture: this post (<https://benfoster.io/blog/asp-net-core-dependency-injection-multi-tenant>) explains the concept and a way to achieve this in the AspNetCore conformant DI context[[3]](#footnote-3).

In the NetCore framework, configurations are exposed and used through [IOptions<T>](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.options.ioptions-1?view=dotnet-plat-ext-3.1), [IOptionsSnapshot<T>](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.options.ioptionssnapshot-1?view=dotnet-plat-ext-3.1) and [IOptionsMonitor<T>](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.options.ioptionsmonitor-1?view=dotnet-plat-ext-3.1) interfaces[[4]](#footnote-4): Their behavior are quite different and this is a good example of a service’s marshalling subtlety.

|  |  |  |
| --- | --- | --- |
| Ioptions | Singleton | Created once at the very first use and never changed. (Note that IOptions has been the very first mechanism available to access configuration data when NetCore was DNX.) |
| IOptionsSnapshot | Scoped | Created at most once per request: its Value can change dynamically during the application lifetime (whenever the configuration files/input change) but remains the same for each request. |
| IOptionsMonitor | Singleton | Exposes a CurrentValue that can change at any time and a OnChange callback registration function. |

The first IOptions is a simple marshall-by-value process that must occur the first time a background service that depends on it is instantiated. It’s the same for IOptionsSnapshot except that the marshalling must be done with each call (like any other scoped services).

The Monitor case is very different. The background implementation cannot retrieve its CurrentValue dynamically (by calling the Front) since the OnChange capability would be lost.

The only way is to monitor the changes on the Front side and either:

1. Immediately sends a message to the background process with the updated configuration value so that changes are reflected in near real time (and OnChange is triggered).
2. Synchronizes the changed configuration value each time a background service that depends on it will be solicited by the command handler (and trigger the OnChange right before the handling of the command). (This is a debounced implementation.)

The first option is the safest one since it enables background services to be as “reactive” as their Front deployment.

This discussion above showed that we eluded an important aspect: there is more than one “background”, because there is at least two kind of Front services, the ones that are bound to the User/Connection/EndPoint (the IAuthenticationInfo is the perfect example) and the ones depend on the Front configuration (like the IOptions<> family).

Welcome to CRIS and DRIS.

* CRIS: the command handlers are in process (in the same process as the End Point), but behind a queue. This supports the A of VISAM, commands are handled Asynchronously so that costly/lengthy computations can be deferred. For this background handling, only EndPoint based services need to be marshalled since configurations are Process wide.
* DRIS: Extends CRIS with out-of-process command handling. The 2 kind of services must obviously be marshalled.

With DRIS in the landscape, one need to refine the current Front services classifications, there are FrontEndPointService and FrontProcessService and “marshallable” cancels both but the distinction between EndPoint and Process Front services must be kept to be able to implement the “executor” and/or the “remote executor”.

We are not obliged to distinguish Command handler Services: it is simple to consider that the Command handlers are just IAutoServices like any others that MUST NOT be IFrontAutoService to be able to do their job in the background.

Another kind of beasts that cannot be Front Services are Real Objects. Real Objects are the basement of the System, they are true singletons, proxies to existing objects: they are by design not related to “Front”/”User interaction side” objects: a Real Object can be a Singleton service (a Real Object typically implements a default implementation of an Auto service) but cannot be a Front service.

# The Fire&Forget or Publish thing

It is quite common for Command oriented frameworks to expose 2 verbs:

* Send: to “send a command” to the System, this usually implies for the caller to wait for the result (to handle potential errors).
* Publish: to “publish an event” to the System, meaning that something happened, the caller is notifying the System of this occurrence and has obviously nothing to wait for.

The introduction of this notion of “event” here is, up to us, unfortunate. The System will certainly generate events (later/behind/after) consequently to the command execution and we prefer keeping this “event effect” on the back side of the System. What does enter a System are just Commands: some of them may have the semantics of an event in the external world, but from the System point of view, it is a Command that needs to be executed, that may trigger a side effect on the System itself, just as any other Command.

The Publish verb however introduces a useful concept in the game: the fire & forget call pattern. And we don’t have it for the moment.

Some Commands must be like Events: carry data that will be interpreted by the back end, may have side-effects on the System (like any other Commands) and potentially generate “true” events BUT doesn’t need to be awaited in any manner.

Our Commands can have an expected Result, this is modeled with these 2 interfaces.

/// <summary>

/// The base command interface marker.

/// Any type that extends this interface define a new command type.

/// </summary>

[CKTypeDefiner]

public interface ICommand : IClosedPoco

{

}

/// <summary>

/// Decribes a type of command that expects a result (that can be of any type).

/// </summary>

/// <typeparam name="TResult">Type of the expected result.</typeparam>

[CKTypeDefiner]

public interface ICommand<TResult> : ICommand

{

}

A way to specify this behavior is to introduce a special return type (a “NoWaitResult”):

/// <summary>

/// Type marker for result of a fire & forget command.

/// </summary>

public sealed class NoWaitResult { }

Fire & forget commands must be ICommand<NoWaitResult>.

# The CommandModel

In Crs implementation, the Command Model that describes the command in terms of

===================== 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>.

1. There is no setup time detection for this currently (only runtime errors): a check should be done at the very end of the registration process and only for marshallable services that are used by non-Front services. [↑](#footnote-ref-1)
2. Any class (even abstract) that is an IAutoService that supports IMarshaller<T> found during the very first registration step will trigger the fact that T is a Marshallable service. The fact that it’s an abstract class that is not on a concrete path is ignored at this level: this inconsistency must be detected at the end of the process just like any missing Marshaller of a marked IMarshallableService. [↑](#footnote-ref-2)
3. Things have changed since 2016 regarding the DI support and capability. See for instance <https://github.com/dotnet/aspnetcore/issues/5949>. Even StructureMap has been deprecated in favor of the new Lamar (<https://jeremydmiller.com/2018/06/14/lamar-1-0-faster-modernized-successor-to-structuremap/>). [↑](#footnote-ref-3)
4. See <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options> for an overview of the Options API. [↑](#footnote-ref-4)