Contents

[I. Command Responsibility Seggregation 2](#_Toc38447786)

[II. The fundamentals 2](#_Toc38447787)

[1. The Command process 2](#_Toc38447788)

[2. Design goal: Inline, background or out-of-process Command execution 3](#_Toc38447789)

[III. The “Single” Command definition in question 4](#_Toc38447790)

[1. The Command as a whole 4](#_Toc38447791)

[2. (Hypothesis) Common sense: differentiating external Command and ValidatedCommand 4](#_Toc38447792)

[a. On the Outside 4](#_Toc38447793)

[b. Inside the End Point 4](#_Toc38447794)

[c. Conclusion 4](#_Toc38447795)

[3. The real issue is about handling the Command… 4](#_Toc38447796)

[IV. Command POCO, ReceivedCommand & Command handler 5](#_Toc38447797)

[1. ICommand is a IClosedPoco 5](#_Toc38447798)

[2. The Fire&Forget or Publish thing 6](#_Toc38447799)

[3. The ReceivedCommand wrapper 7](#_Toc38447800)

[4. Looking for the best possible CommandHandler definition 7](#_Toc38447801)

[V. The bestiary of Services 8](#_Toc38447802)

[1. Services must be categorized 8](#_Toc38447803)

[2. The Service “FrontType” category 8](#_Toc38447804)

[3. The FrontService virality. 9](#_Toc38447805)

[4. What is a Marshallable Service? 10](#_Toc38447806)

[a. The eventually required Model.IMarshaller<T>. 10](#_Toc38447807)

[b. The IsMarshallable claim. 10](#_Toc38447808)

[c. The case of the Marshallable Singleton 10](#_Toc38447809)

[d. Welcome to CRIS and DRIS. 11](#_Toc38447810)

[5. The final AutoService categories 12](#_Toc38447811)

[VI. Draft, To do & Garbage 14](#_Toc38447812)

# Command Responsibility Seggregation

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

## The Command process

The big picture of the Command processing is described below:

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 should also be 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.

## Design goal: Inline, background or out-of-process Command execution

The handler may be called straight from the EndPoint, or the actual execution may be deferred to a background service or to another process, may be on another machine. There are two main reasons why we would like to avoid processing a Command immediately.

* The easiest reason is scaling. Depending on the workload and/or the Command to be executed, one may need to rely on other machines to do the hard work.
* A more subtle reason is when a Command simply takes (or may take) a long time to be executed. Instead of forcing the caller to wait for the end of the execution, we activate the ‘A’ of VISAM: the request is technically “finished” with a ‘A’ returned code (freeing any associated resources) and the job is queued to a background service. Once processed, the caller will receive the result (or the error).

# The “Single across the System” 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) “external” Command become 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 must 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. Under this hypothesis, 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 is about handling the Command…

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 “real Service” is a totally useless indirection that only brings complexity. (This kind of 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 he developed had to rely on other Services or existing API that *de facto* require this IAuthenticationInfo to do their 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!

Before analyzing this in-depth, we need to better define what is a Command and its handler.

# Command POCO, ReceivedCommand & Command handler

## ICommand is a IClosedPoco

We use a ICommand interface to mark the commands because marker interfaces proved to be easy to understand, increase code readability and ease discovery implementations.

But there is more to this: this ICommand interface is marked with a [CKTypeDefiner] attribute that automatically registers any specialized interfaces or implementation classes as a “Closed Poco”.

A Poco is an original construct that is supported by the CK.StObj.Model of CK-Database. It implements simple Mixins across assemblies (that don’t know each other) of a root type (the first interface that extends the IPoco interface and that the others extend).

|  |
| --- |
| Interfaces that extends IPoco (or IClosedPoco) don’t need to be implemented by the developer: an implementation class that unifies all the related interfaces is automatically generated. |

The IClosedPoco is a refinement of the IPoco marker that ensures that a unified interface exists in the code base (across the multiple assemblies). This unified interface MUST extend all the different definitions of the same root: it must “know them”. If such a “Unified Interface” cannot be found it is an error.

This guaranties that the developer knows (is aware of) all the “parts” of the Mixin that exist among the different assemblies that he uses and that, somewhere in his code base, he defined this unified, final, type.

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 defines 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

{

}

Thanks to the IClosedPoco concept and with the help of the [CKTypeSuperDefiner] attribute that allows its direct specializations to be Definer and not the “final type of interest”, we can offer “partial command definition” that share common Command aspects:

/// <summary>

/// Marker interface to define mixable command parts.

/// </summary>

[CKTypeSuperDefiner]

public interface ICommandPart : IClosedPoco

{

}

The drawback of using IPoco and IClosedPoco here is that it creates a dependency on CK.StObj.Model since, contrary to the other markers and attributes of CK-Database like CKTypeDefiner, these interfaces are not “duck typed” interfaces and cannot be defined locally (this is because they are used in the constraint of the Poco factory interface).

However, this dependency may be required anyway if a lot of Code Generation and static analysis happens to be necessary to implement the whole Command handling.

## 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.

An easy 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 simply be ICommand<NoWaitResult>.

## The ReceivedCommand wrapper

Once received by an End Point, a Command must be decorated with some meta information (described above in the picture). This can be represented by a simple ReceivedCommand wrapper 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).

Before analyzing the dependencies and work on the “Service Ontology”, let’s talk a little bit more about this “Handler” that executes the Command.

## Looking for the best possible CommandHandler definition

In our previous CRS, Command handlers were standard Services that implement an interface that looks like the following one (for a command without result):

public interface ICommandHandler<TCommand> where TCommand : ICommand

{

Task HandleAsync( ReceivedCommand<TCommand> command );

}

Note that this definition allows a class to implement multiple Command handlers and that is a good thing, however its exact form is not important since the first question to answer is “Do we need an interface?”.

This interface has, at least, one important drawback: it forces an Asynchronous implementation. Async variation is not always what we want. Adding a synchronous version (Handle only) would oblige the developer to implement both, which means either:

* Really implement both that is the best option in terms of quality for the caller but can be a real nightmare to develop and maintain.
* Choose one and adapt the other: either use “sync on async” anti-pattern or return a Task.FromResult or Task.CompletedTask.

Another idea would be to have 2 handler interfaces: one for each handler method.

Another drawback: the signature is “closed” by design of what an interface is… This prevents us to benefits of parameter injection.

Let’s imagine a totally different approach that is based on duck typing: any function that accepts a ReceivedCommand<TCommand> as one of its parameters is a handler.

With CK-Database, this can be done more easily than without it: the discovery can be done at setup time and code generated to register the handlers once for all.

This “duck type” definition obviously needs some refinements:

* A handler is method that is named **Handle** or **HandleAsync**.
* It must have one and only one parameter that is either an ICommand or a ReceivedCommand<TCommand>.
* If the ICommand is [not] a ICommand<TResult> then the return of the Handle must be [void] TResult and HandleAsync must return a [Task] Task<TResult>.
  + Note that covariance of the return type may be supported (this must be discussed but should be rather easy to do).
* The handler must be a public method of an IAutoService. This is to ease the discovery and support auto implementation of companion methods/adapter.
* Thanks to Code Generation, adapters between asynchronous/synchronous can be automatically supported.

We’ll come back to this handler (and to the ReceivedCommand wrapper) later, after having investigated a bit more whether such handler can support our primary design goal that is to secure and automate as much as possible Inline, background and out-of-process Command execution.

# The bestiary of Services

## Services must be categorized

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.
* By using a configuration like <ExternalSingletonTypes /> elements for other interfaces.

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 and/or 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 is not marshallable** 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.

Note that 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.

Nevertheless, we allow this IsMarshallableAttribute to be applied to an interface and a base class: this means that there exists a way to marshal the interface/base class and that when marshalling this the developer assumes that there will be no differences whatever the underlying implementation or final type is.

### 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 (in terms of Process) configuration (like the IOptions<> family).

### Welcome to CRIS and DRIS.

Since marshalling cross-process requires more stuff that remaining inside the same process and dispatching among multiple process implies to be able to address/name them, the out-of-process option is more complex to implement. We split the implementation in two packages:

* 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, and there must be a way to identify the other processes from the Front process.

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 (that expose their Handle/HandleAsync methods): it is simple to consider that the Command handlers are just IAutoService 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 final AutoService categories

This work on Service kind results (after many bad ideas and as of April 2020) in one simplified enumeration that is exposed to developers:

|  |
| --- |
| /// <summary>  /// Defines Auto services flags.  /// </summary>  [Flags]  public enum AutoServiceKind  {  /// <summary>  /// Not a service we handle or external service for which  /// no lifetime nor front binding is known.  /// </summary>  None = 0,  /// <summary>  /// This is a front service, bound to the front process that cannot be used directly in  /// another process and needs to be marshalled to any other process (a typical example is  /// the IOptions&lt;&gt; implementations for instance).  /// </summary>  IsFrontProcessService = 1,  /// <summary>  /// This is a front service bound to the End Point: even inside the front process, it cannot be used directly  /// (a typical example of such service is the HttpContext). To be used by a background service,  /// it must be IsMarshallable.  /// This flag implies IsFrontProcessService AND IsScoped: a Front only service is necessarily Scoped  /// (since a Singleton is, by design, available in the whole process).  /// </summary>  IsFrontService = 2,  /// <summary>  /// This service is marchallable. This is independent of IsFrontProcessService and IsFrontService.  /// </summary>  IsMarshallable = 4,  /// <summary>  /// This service must be registered as a Singleton.  /// </summary>  IsSingleton = 8,  /// <summary>  /// This service must be registered as a Scoped service.  /// </summary>  IsScoped = 16,  /// <summary>  /// This is applicable only to interfaces. It states that the service is not unique: interfaces marked with  /// this flag must all be registered, associated to each of their implementation.  /// </summary>  IsMultipleService = 32  } |

Under the hood, inside CK.StObj.Engine, the current categorization is a little bit more complicated:

|  |
| --- |
| /// <summary>  /// Defines the "services" kind and life times and invalid combination of  /// IAutoService and IRealObject.  /// </summary>  [Flags]  public enum CKTypeKind  {  /// <summary>  /// Not a service we handle or external service for which  /// no lifetime nor front binding is known.  /// </summary>  None,  /// <summary>  /// Front process service.  /// This flag has to be set for IsFrontService to be set.  /// </summary>  IsFrontProcessService = 1,  /// <summary>  /// Service is bound to the End Point. The service is necessarily bound to front  /// process (IsFrontProcessService is also set) AND this is necessarily IsScoped.  /// </summary>  IsFrontService = 2,  /// <summary>  /// Marshallable service.  /// This is independent of IsFrontProcessService and IsFrontService flags.  /// </summary>  IsMarshallable = 4,  /// <summary>  /// Singleton flag.  /// External (Auto) services are flagged with this (without the IsAutoService bit).  /// </summary>  IsSingleton = 8,  /// <summary>  /// Scoped flag.  /// External (Auto) services are flagged with this (without the IsAutoService bit).  /// </summary>  IsScoped = 16,  /// <summary>  /// Multiple registration flag (services must be registered with TryAddEnumerable instead of TryAdd).  /// See IsMultipleAttribute.  /// External (Auto) services are flagged with this (without the IsAutoService bit).  /// </summary>  IsMultipleService = 32,  /// <summary>  /// Auto service flag. This flag is set if and only if the type is marked with a IAutoService interface marker.  /// </summary>  IsAutoService = 64,  /// <summary>  /// A IPoco marked interface.  /// </summary>  IsPoco = 128,  /// <summary>  /// A real object is a singleton.  /// </summary>  RealObject = IsSingleton | 256,  /// <summary>  /// Simple bit mask on IsFrontService | IsFrontProcessService.  /// </summary>  FrontTypeMask = IsFrontService | IsFrontProcessService,  /// <summary>  /// Simple bit mask on IsScoped | IsSingleton.  /// </summary>  LifetimeMask = IsScoped | IsSingleton  } |

# Closer to the Handlers: Lifetime, Batching & Local Commands

Sometimes multiple Commands must be grouped and executed together: in its simplest form, a batch is an ordered list of commands that must be executed. Another correlated aspect to discuss is the possibility to “send a Command” from the process itself, without any ReceivedCommand from any Front End (think to a simple Timer that must send a Command on its due time).

Before analyzing these subjects, let’s describe a little bit more a Command handler.

## Lifetime of a Command handler

A command handler is a IAutoService. It’s even simpler to say that any IAutoService CAN be a Command handler. First, this means that all the rules that apply to Auto Services apply here like Service replacement (via [ReplaceAutoServiceAttribute] or constructor parameter) and lifetime management.

To be a Command handler a IAutoService must expose at least one public named Handle or HandleAsync method as described previously: one (and only one) of its parameter is a ICommand or a ReceivedCommand<TCommand>. The other parameters of the Handle methods must be resolved (this is parameter injection in its purest form). Thanks to this, a typical “Command Handler Service” (with only one type of command handled) implementation looks like this:

“class wide”, shared dependencies

public class SampleCommandHandler : IAutoService

{

readonly IAuthorizationServer \_auth;

readonly IMailSenderService \_mailer;

public SampleCommandHandler( IAuthorizationServer auth, IMailSenderService mailer )

{

\_auth = auth;

“Command specific” dependencies

\_mailer = mailer;

}

public void Handle( IVoidAuthorizedCommand cmd, IActivityMonitor monitor )

{

\_auth.Check( cmd.ActorId );

monitor.Info( $"Handling VoidAuthorized: {cmd.Parameter}, {cmd.ActorId}" );

}

}

This makes easy to group related command handlers in a place where common concerns are easy to share and maintain (and specific needs are easily spotted). However here, the attentive reader may say “The Activity Monitor is useful to each handler, why isn’t it injected in the constructor once for all?”. You can perfectly do this… but this would make your SampleCommandHandler a Scoped service… Here, as long as the underlying implementation of the IAuthorizationServer and IMailSenderService are Singletons (which they seem to be), our command handler is “optimal”: it is resolved along with its constructor dependencies, allocated and inialized once for all (at the first use), and the specifics and/or scoped services required must then be resolved each time a command must be processed.

And if you want to ensure that this handler remains a Singleton, whatever the implementation of the Mail and Authorization services are or become, simply use the ISingletonAutoService interface marker instead of IAutoService: there will be an error during the Setup of the System. (This “magic” is proudly supported by the CK-StObj layer.)

The conclusion so far could hardly be simpler: a Command handler is a IAutoService with methods that handle a ICommand object.

## Local command… What about the ReceivedCommand wrapper?

In the typical Command handler above, the Handle method accepts the IVoidAuthorizedCommand object, not the ReceivedCommand wrapper that we introduced previously in order to replace/specify the rather unspecific, fuzzy and undetermined “CommandContext” notion:

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;

}

What does this wrapper bring to the handling of a Command itself? Nothing.

CommandId, CallerId, CorrelationId are for traceability, persistence possibly (the CommandId identifies the command instance) and to route the result (or the error) back to the caller (thanks to the CallerId) and this latter responsibility is the job of the End Point (or any intermediate router), but not the job of the Command handler! And the fact that we are AsynchronousHandlingMode? Who cares? Not the handler for sure…

My intuition is that we’d better extend this “caller information” to a more general and powerful concept: Call Stack.

Commands have a Call Stack just like regular functions. This Call Stack starts with an initial StackFrame that contains the Front information and a handler can send another command, wait (or not) for its result and continue its job: Commands can “send other Commands”, Commands become reusable, composable.

Let’s dream: with a bit of persistence, a Command handler could be turned into a kind of “Workflow” that orchestrate calls between “WorkflowItems” (using [Durable Tasks](https://github.com/Azure/durabletask/) maybe?).

For now, it’s enough to replace the ReceivedCommand by the CommandCallStack and its Frame.

public sealed class CommandCallStack : IReadOnlyList<Frame>

{

public class Frame

{

public string CommandId { get; }

public string CallerId { get; }

public string CorrelationId { get; }

public DateTime CreationTime { get; }

}

}

The “Receiver” notion fades away and it’s now easier to deal with a Command sent by the system itself (the Timer/Reminder case). This Command’s Frame has simply no CallerId, or a CallerId that denotes this System itself: a null or empty string is a good candidate.

Should this Call Stack be injected into a handler method? Not always for sure since a typical command handler has nothing to do with it. However, some handlers MAY need this: the CommandCallStack can appear in the method parameters (and it will be injected).

## Handling the command Results: from CallStack to ResultTree

At this point, with the introduction of the call stack, Commands look like functions (more accurately, command handlers are functions), and somehow this is true. However, one important thing differs: granularity. Commands are (and must be) coarse grained actors in a System: they must be conceived to encapsulate high level operations.

### The Command granularity

For instance, consider the following scenario: in a document management system in which a CreateDocument command exists, we are asked to develop a batch import of document from an external source: we need to support a CreateManyDocuments command.

Unfortunately, the wrong way to do this may seem the obvious one: considering the Many commands as a simple list of unitary commands and iterating on it to call as many CreateDocument as needed. This is wrong!

Each Command must be treated as a semantical action on the system, it must be handled for what it means to the System and its Users, not for what it is in terms of implementation.

The right way to do this is nearly the same: it is the underlying implementation of CreateCommand that may be solicited as many times as needed, but NOT the command itself emitted: what MUST BE executed (and observed externally) is one execution of the CreateManyDocuments command with 3712 documents, not 3712 CreateCommand executions.

This is not to say that the execution of a Command must not trigger other Command executions (otherwise why would we introducing a Command Call Stack?). But when “a Command sends Commands”, this must be to implement “coordinated processes”, support high level workflows of actions. Such high-level orchestrations are usually called “Saga” and this is what the Command Call Stack aims to support.

Saga implies multiple command executions, but this multiplicity must be under control, must preserve the ability of a human being to reason about the System (10, 20, 50 commands may be?). Whenever too many commands are triggered from an initial Command, then some “bigger” Commands are required, and something must be refactored.

We believe that Commands’ granularity (“size” vs. “cardinality”) must be considered a fundamental metric of any System that uses the Command pattern.

### The Result of multiple Commands

Based on the previous discussion, we’re introducing here a “strange” capability into our Command system: we are going to capture (and make visible) the combined results of multiple commands as the detailed set of each execution.

This breaks the parallel between “function calls” and “command handling”: results of subordinated executions become visible to callers[[5]](#footnote-5).

Why are we doing this? Exactly for this reason: breaking the parallel to impose a new vision to the Architects:

* Commands are not Functions.
* There are no such things as “low-level” vs. “high-level”, “minor” vs. “major” commands.
* A Command that triggers other Commands orchestrates their work
  + Designs them
  + Handles their results
* Just like a Human may designed, handled their results and took decisions on this basis!

To enforce this idea, we’ll name any Command that triggers a call to another one a “Saga Command”. The multiple results of the different calls will be captured in a ResultTree: the execution of a Saga Command results in a composite result where its own Result (if any) is returned along with the results from the triggered commands.

## Batching: how multiple commands must be handled at once?

Batching requires definitions: there can be multiple ways to “batch” things (or is this my imagination?):

* Error sensitivity: can some of the commands fail silently or every command are “critical”.
* Atomicity: should a Batch of Commands execution fails completely whenever one of the Command fails?
* Serialization: should all the Commands be executed one after the others or some of them should/can be executed in parallel?

|  |
| --- |
| For the record (Batching must be simple) |
| We keep the batch notion here simple. Based on the questions above, I started the dangerous loop of “one could do or need this…” (the [YAGNI](https://martinfowler.com/bliki/Yagni.html) syndrome). I envisioned a more powerful batch that may be called “micro-program” or “macro-command” that would allow more “reuse” of existing Commands. Something that started simple (with a small set of enumerated options) and ended up being as complex as the following DSL (this is just a stupid sample):  using( RunOption.OnErrorContinue() )  {  const u = CreateUser( “Albert” );  if u.Success  {  SetUserDetails( u.UserId, “Camus”, 1913, 1960 );  }  else using( RunOption.OnErrorThrow() )  {  SetUserDetails( CreateUser( “Albert2” ).UserId, “Camus (Albert)”, 1913, 1960 );  }  }  This would be perfectly implemented, if needed, with an InterpretedCommand and its string Code property... and maybe even a string Language one. |

Even if the “micro-program” above is totally out of the scope of Batching, it introduces a new participant into the “Command Handling System” we are designing: the “RunOption” above is a kind of property of a more general ExecutionContext that is a requirement as long as **more than one command** is at stake and we want to be able to somehow “coordinate” their respective execution.

As soon as we want to execute 2 commands atomically, things get complicated. The simplest batch implementation (that has no impact on the design) is just to execute the commands one after the other as if they arrived separately on the End Point (this maybe called a “transport layer only” or “external” batching capability). But if we want 2 commands to success or fail together (the A of ACID), we need coordination through a kind of ExecutionContext.

The Atomicity (something is totally done or not at all) can be achieved thanks to only 2 mechanisms (the two of them may be mixed):

* The easy way is to use transactional-aware sub-systems and do a commit at the end of all the operations. (And of course, to rollback everything as soon as something went wrong.)
* The hard way is by implementing Compensations mechanisms, typically using a Saga pattern.

|  |
| --- |
| **Important** |
| The reader, from this point, should know the basics of what a transaction is and how compensations work. The following article is clear and concise and exposes this bare minimum: <https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture/> |

The problem here is that we (must) know nothing about the real Commands, the sub-systems they interact with. We must maximize the “usability” of this “Command Handling System” so that these 2 mechanisms CAN BE used by concrete command handlers WHEN they need it.

### When Transactions are used

In this case, the command handler interacts with a sub-system in which one can Start a Transaction and Commit or Rollback it. When 2 commands (C1 and C2) interact with the same sub-system, being atomic is easy as long as:

* A “parent” transaction is opened “above” them.
  + C1 starts a (nested) transaction  
    C1 commits – nothing is really committed
  + C2 starts a (nested) transaction  
    C2 commits – nothing is really committed
* This “parent” transaction is committed if and only if no error occurred after the whole execution.

Any transactional APIs (the System.Data.Common.DbTransaction) enables such nested transactions (and if only one transaction level is supported, it is easy to offer a nested transaction counter above it).

**Important:** We absolutely reject the use of any implicit transaction mechanism such as the [System.Transactions.TransactionScope](https://docs.microsoft.com/en-us/dotnet/api/system.transactions.transactionscope). Transaction control is complex and as such must be as explicit as possible.

The first (rather easy) question is whether a parent transaction always be opened? After all, C1’s success may not be correlated to C2: not all batches are equal. There must be one BatchCommand with options (an Atomic property for example) or different type of batches.

The second one is more subtle: how can we properly isolate the different pieces of code that are required to support this?

A (required) part of the answer lies in the sub-systems themselves. For instance, a typical handler uses a ISqlCallContext to interact with one (or more) Sql Server databases. This ISqlCallContext provides the handler with connection controllers that can control regular SqlConnection objects or SqlConnection and SqlTransaction pairs for databases, and the code is the same whether the connection is enlisted in a transaction or not.

The point above guaranties that the handler’s code has no explicit dependencies on the ambient transaction, but nothing tells us that the handler uses Sql Server, PostGres and/or[[6]](#footnote-6) MySql or any other transactional sub-systems. When a Batch must be Atomic, the “connection” to the sub-system should be enlisted to its “transaction manager” (or whatever is required to ensure the transactionality) prior to the handler’s execution: instead of simple “connection”, it must be given a “transactional connection”. And once the Commands have been executed there must be a way to agree on the success or failure.

The conclusion hers is that all this stuff highly depends on the real API and sub systems at stake. Even the configuration of such batch can hardly be generalized. Any such “coordination system” imply specific code and specific infrastructure/dependencies: let’s NOT try to handle this in a generic manner.

It is expected that patterns will appear, and more-or-less complete out-of-the-box solutions will be made available to support transactional execution.

### Without transaction support

There are two ways to implement an action for which eventual side-effects depend on a future (out-of-control) result.

* The first one is (once all the required tests and decisions have been made) to postpone the resulting side effect: a OnSuccess deferred action registration is enough.
* The second one is to apply the side effects immediately and to register a OnError deferred action that can be called to cancel those side effects.

We may offer any Command handler an access to these capabilities. This can be supported by something like this:

/// <summary>

/// Registrar for asynchronous and synchronous actions and error handlers

/// that operates on a generic T parameter.

/// </summary>

public interface IActionRegistrar<T>

{

/// <summary>

/// Registers a new asynchronous success handler.

/// This will be called after the last action, any exception thrown by the handler

/// will be logged and ignored. A success handler is not allowed to register any

/// new action or error handler but it can register another success handler if needed.

/// </summary>

/// <param name="handler">The success handler to register.</param>

void OnSuccess( Func<T, Task> handler );

/// <summary>

/// Registers a new asynchronous error handler.

/// This will be called if an action throws an exception. Any exception thrown by this handler

/// will be logged and ignored. An error handler is not allowed to register any

/// new action or success handler but it can register another error handler if needed.

/// </summary>

/// <param name="errorHandler">The error handler to register.</param>

void OnError( Func<T, Exception, Task> errorHandler );

}

We can generalize this:

* By handling more than regular asynchronous actions (Func<T, Task> for instance can easily be derived from Func<T, ValueTask> or Action<T>).
* By extending this definition to register the initial actions: void Add( Func<T, Task> action );
* By considering the generic parameter T to be itself the IActionRegistrar, we can extend this registrar to act as a [trampoline](https://en.wikipedia.org/wiki/Trampoline_(computing)) where any action can defer other actions.

And to finalize this API we can reproduce the standard exception-based error handling by adding a 4th kind of handlers:

/// <summary>

/// Registers a new asynchronous final handler.

/// This will be called after success or error handlers. Any exception thrown by this handler

/// will be logged and ignored. A final handler is not allowed to register any

/// new action, success or error handler but it can register another final handler if needed.

/// </summary>

/// <param name="finalHandler">The final handler to register.</param>

void Finally( Func<T, Task> finalHandler );

This “registrar” is in fact a generic, solid, easy to use (totally optional) and powerful Command Execution Context.

## Sum up

The three parts above have led us to the following conclusions:

* A Command handler is a IAutoService with public Handle and HandleAsync methods, that may return a result, that handle a ICommand object with the help of any number of injected parameters.
* The ReceivedCommand wrapper is dead
  + The CommandCallStack generalizes it.
  + The System can send a Command lo itself: nothing differentiates a Command received from an external application and the one launched by a local timer.
* Multiple Command execution Results must be combined in a ResultTree that must be available to the caller.
* Any “command execution” should have access to a IActionRegistrar that enables multiple command execution to interact in the most possible easy way.

We are now able to specify an important part of the system: the CommandExecutor that is the layer right above a command handler and is in charge of calling the handler function.

# Draft, To do & Garbage

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 );

}

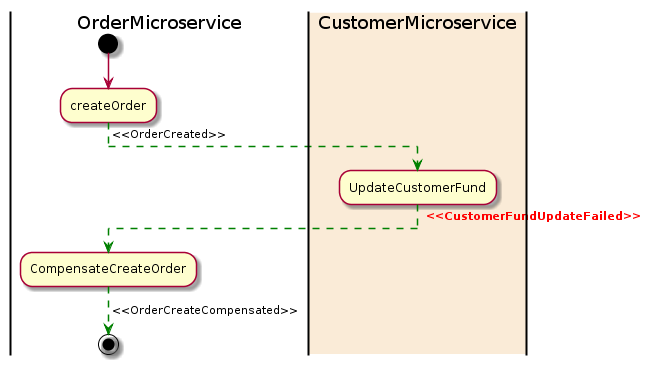
This is NOT a good idea: the Handler must be independent of the infrastructure. If we do this, the Handler cannot be tested independently and HAS TO be CKSetup. This is BAD.

* + 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.

# Annex

## From developers.redhat.com: Patterns for distributed transactions within a microservices architecture

By [Keyang Xiang](https://developers.redhat.com/blog/author/kxiang/" \o "Keyang Xiang) October 1, 2018

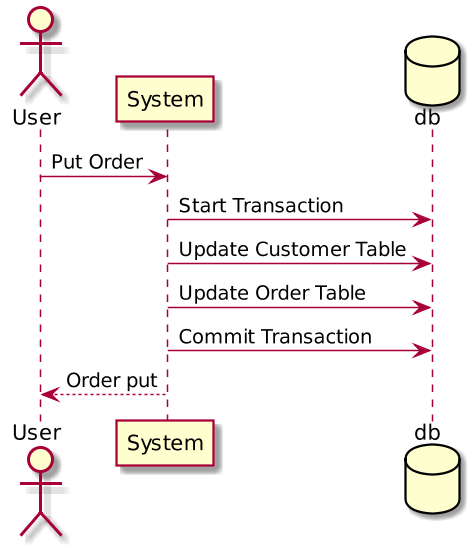


[Microservices](https://developers.redhat.com/topics/microservices/) architecture (MSA) has become very popular. However, one common problem is how to manage distributed transactions across multiple microservices. This post is going to share my experience from past projects and explain the problem and possible patterns that could solve it.

### What is a distributed transaction?

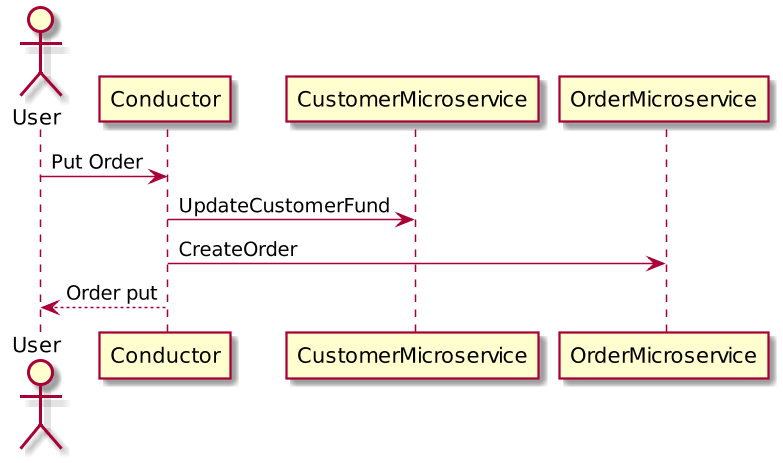
When a microservice architecture decomposes a monolithic system into self-encapsulated services, it can break transactions. This means a local transaction in the monolithic system is now distributed into multiple services that will be called in a sequence.

Here is a customer order example with a monolithic system using a local transaction:

[](https://developers.redhat.com/blog/wp-content/uploads/2018/09/Untitled-UML-4.png)

In the customer order example above, if a user sends a Put Order action to a monolithic system, the system will create a local database transaction that works over multiple database tables. If any step fails, the transaction can roll back. This is known as ACID (Atomicity, Consistency, Isolation, Durability), which is guaranteed by the database system.

When we decompose this system, we created both the CustomerMicroserviceand the OrderMicroservice, which have separate databases. Here is a customer order example with microservices:

[](https://developers.redhat.com/blog/wp-content/uploads/2018/09/Untitled-UML-5.png)

When a Put Order request comes from the user, both microservices will be called to apply changes into their own database. Because the transaction is now across multiple databases, it is now considered a distributed transaction.

### What is the problem?

In a monolithic system, we have a database system to ensure ACIDity. We now need to clarify the following key problems.

#### **How do we keep the transaction atomic?**

In a database system, atomicity means that in a transaction either all steps complete or no steps complete. The microservice-based system does not have a global transaction coordinator by default. In the example above, if the CreateOrder method fails, how do we roll back the changes we applied by the CustomerMicroservice?

#### **Do we isolate user actions for concurrent requests?**

If an object is written by a transaction and at the same time (before the transaction ends), it is read by another request, should the object return old data or updated data? In the example above, once UpdateCustomerFund succeeds but is still waiting for a response from CreateOrder, should requests for the current customer’s fund return the updated amount or not?

### Possible solutions

The problems above are important for microservice-based systems. Otherwise, there is no way to tell if a transaction has completed successfully. The following two patterns can resolve the problem:

* 2pc (two-phase commit)
* Saga

#### Two-phase commit (2pc) pattern

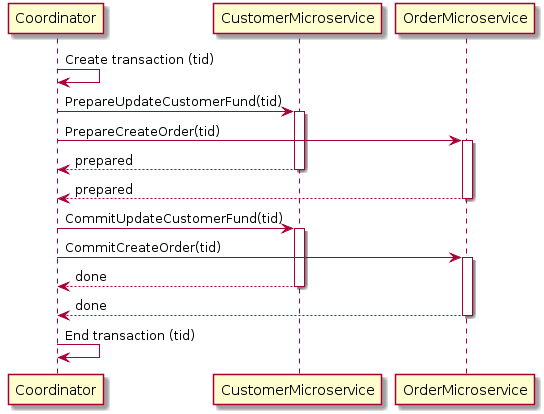
2pc is widely used in database systems. For some situations, you can use 2pc for microservices. Just be careful; not all situations suit 2pc and, in fact, 2pc is considered impractical within a microservice architecture (explained below).

So what is a two-phase commit?

As its name hints, 2pc has two phases: A prepare phase and a commit phase. In the prepare phase, all microservices will be asked to prepare for some data change that could be done atomically. Once all microservices are prepared, the commit phase will ask all the microservices to make the actual changes.

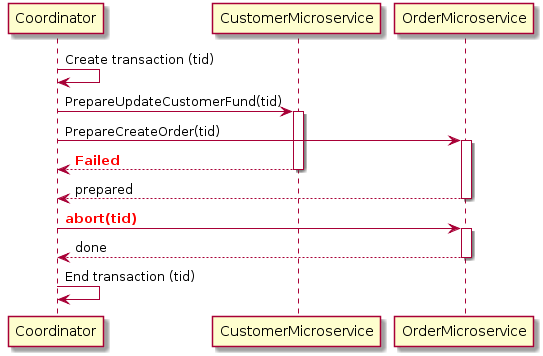
Normally, there needs to be a global coordinator to maintain the lifecycle of the transaction, and the coordinator will need to call the microservices in the prepare and commit phases.

Here is a 2pc implementation for the customer order example:

[](https://developers.redhat.com/blog/wp-content/uploads/2018/09/Untitled-UML-6.png)

In the example above, when a user sends a put order request, the Coordinator will first create a global transaction with all the context information. It will then tell CustomerMicroservice to prepare for updating a customer fund with the created transaction. The CustomerMicroservice will then check, for example, if the customer has enough funds to proceed with the transaction. Once CustomerMicroservice is OK to perform the change, it will lock down the object from further changes and tell the Coordinator that it is prepared. The same thing happens while creating the order in the OrderMicroservice. Once the Coordinator has confirmed all microservices are ready to apply their changes, it will then ask them to apply their changes by requesting a commit with the transaction. At this point, all objects will be unlocked.

If at any point a single microservice fails to prepare, the Coordinator will abort the transaction and begin the rollback process. Here is a diagram of a 2pc rollback for the customer order example:

[](https://developers.redhat.com/blog/wp-content/uploads/2018/09/Untitled-UML-7.png)

In the above example, the CustomerMicroservice failed to prepare for some reason, but the OrderMicroservice has replied that it is prepared to create the order. The Coordinator will request an abort on the OrderMicroservice with the transaction and the OrderMicroservice will then roll back any changes made and unlock the database objects.

#### Benefits of using 2pc

2pc is a very strong consistency protocol. First, the prepare and commit phases guarantee that the transaction is atomic. The transaction will end with either all microservices returning successfully or all microservices have nothing changed.  Secondly, 2pc allows read-write isolation. This means the changes on a field are not visible until the coordinator commits the changes.

#### Disadvantages of using 2pc

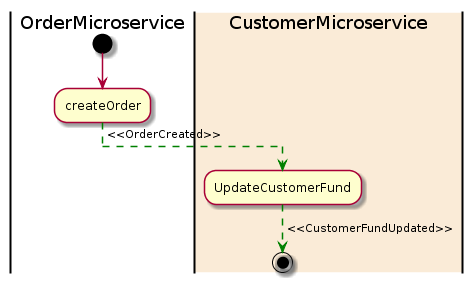
While 2pc has solved the problem, it is not really recommended for many microservice-based systems because 2pc is synchronous (blocking). The protocol will need to lock the object that will be changed before the transaction completes. In the example above, if a customer places an order, the “fund” field will be locked for the customer. This prevents the customer from applying new orders. This makes sense because if a “prepared” object changed after it claims it is “prepared,” then the commit phase could possibly not work.

This is not good. In a database system, transactions tend to be fast—normally within 50 ms. However, microservices have long delays with RPC calls, especially when integrating with external services such as a payment service. The lock could become a system performance bottleneck. Also, it is possible to have two transactions mutually lock each other (deadlock) when each transaction requests a lock on a resource the other requires.

#### Saga pattern

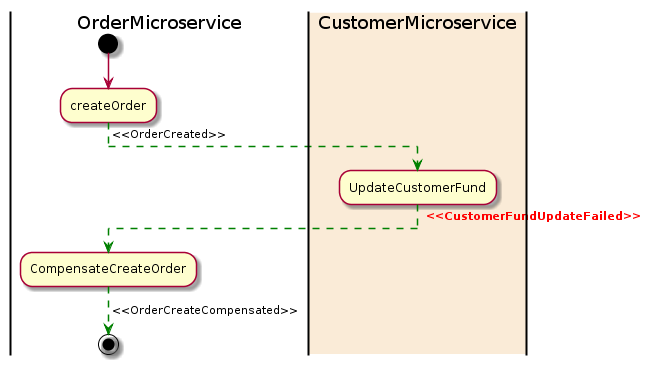
The Saga pattern is another widely used pattern for distributed transactions. It is different from 2pc, which is synchronous. The Saga pattern is asynchronous and reactive. In a Saga pattern, the distributed transaction is fulfilled by asynchronous local transactions on all related microservices. The microservices communicate with each other through an event bus.

Here is a diagram of the Saga pattern for the customer order example:

[](https://developers.redhat.com/blog/wp-content/uploads/2018/09/Untitled-UML-8.png)

In the example above, the OrderMicroservice receives a request to place an order. It first starts a local transaction to create an order and then emits an OrderCreated event. The CustomerMicroservice listens for this event and updates a customer fund once the event is received. If a deduction is successfully made from a fund, a CustomerFundUpdated event will then be emitted, which in this example means the end of the transaction.

If any microservice fails to complete its local transaction, the other microservices will run compensation transactions to rollback the changes. Here is a diagram of the Saga pattern for a compensation transaction:

[](https://developers.redhat.com/blog/wp-content/uploads/2018/09/Untitled-UML-9.png)

In the above example, the UpdateCustomerFund failed for some reason and it then emitted a CustomerFundUpdateFailed event. The OrderMicroservice listens for the event and start its compensation transaction to revert the order that was created.

#### Advantages of the Saga pattern

One big advantage of the Saga pattern is its support for long-lived transactions. Because each microservice focuses only on its own local atomic transaction, other microservices are not blocked if a microservice is running for a long time. This also allows transactions to continue waiting for user input. Also, because all local transactions are happening in parallel, there is no lock on any object.

#### Disadvantages of the Saga pattern

The Saga pattern is difficult to debug, especially when many microservices are involved. Also, the event messages could become difficult to maintain if the system gets complex. Another disadvantage of the Saga pattern is it does not have read isolation. For example, the customer could see the order being created, but in the next second, the order is removed due to a compensation transaction.

#### Adding a process manager

To address the complexity issue of the Saga pattern, it is quite normal to add a process manager as an orchestrator. The process manager is responsible for listening to events and triggering endpoints.

### Conclusion

The Saga pattern is a preferable way of solving distributed transaction problems for a microservice-based architecture. However, it also introduces a new set of problems, such as how to atomically update the database and emit an event. Adoption of the Saga pattern requires a change in mindset for both development and testing. It could be a challenge for a team that is not familiar with this pattern. There are many variants that simplify its implementation. Therefore, it is important to choose the proper way to implement it for a project.

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)
5. Of course, for “fire and forget” commands (modeled as ICommand<NoWaitResult>) there is no result of any kind. [↑](#footnote-ref-5)
6. When more than one transactional sub-system must be synchronized, this starts to be much more complicated and this is out of the scope of this document. See <https://en.wikipedia.org/wiki/Distributed_transaction> as a start about distributed transactions. [↑](#footnote-ref-6)