Apollo

Kernel specification

Confidential

Revision number: 1

Date:

Name: P. van der Velde

# Disclaimer

This specification is not, by any stretch of the imagination, complete. It will need to be revised several times before it is complete. Currently several major parts are either missing or incomplete. This disclaimer will be updated to reflect any change in these sections. Finally a specification is supposed to be a ‘living’ document and therefore never complete. What were you thinking, better learn to live with this fact.

# Introduction

# Architecture

The core of the Apollo application consists of a service provider and a set of services which provide all the capabilities for the system. The use of services allows adding and changing services without having to make changes to other parts of the application. It should be noted that the use of services by itself does not enforce loose coupling however it should make it simpler to have a loosely coupled system.

The core elements are:

* **Bootstrapper:** Loads the initial instances of the kernel.
* **Kernel:** Controls the services
* **Kernel services:** Provide the capabilities for the core part of Apollo
* **AppDomain builder:** Builds the application domains necessary for the different services

## Bootstrapper

The main task for the boostrapper is to load all the kernel objects in the right order and provide them with their initial starting data. The order in which the bootstrapper performs its actions is:

* Link the assembly resolver to current AppDomain. The current domain will later be used for the User Interface (UI). This is especially important if Apollo is running as a plug-in to another application (e.g. CAD) where the initial AppDomain is the one created by the owner application.
* Use the AppDomain builder to create a new AppDomain and
  + Set the security levels
  + Set the search paths
  + Link assembly resolver to the AssemblyLoad event
* Inject a remote object loader into new AppDomain. This object loader will be used to create the kernel objects in the new AppDomain.
* Load the Kernel object in the new AppDomain and
  + Pass a reference to original AppDomain to the kernel. This original AppDomain will later be used for the User Interface.

Once the kernel is loaded the individual services can be loaded and started. The Apollo framework does not allow services to be loaded from external assemblies because these services run in the core of the system. Any instabilities or unsafe code would be able to cause a large amount of problems. The discovery and loading of the services is done with the help of MEF. Each service defines which methods it exports and which other services it needs directly. Furthermore each service can demand that certain services are available at the moment they get loaded.

A suggested load order is:

* Message service - The message service connects all the other services, thus all other services depend on this being available.
* Core - Provides the connection from the kernel to the services. Is required by the kernel in order to communicate with the other services. Should be started as early as possible but does require the message service and the LogSink to be functional.
* License service - The license service is required to start the persistence service. One part of the licensing will be the ability to deny file writes. This means that the license service is required for the persistence service.
* Persistence service - The persistence service is used for file writes, ranging from the simple log file to the more complex project files.
* LogSink - Deals with the log messages that are send by the other parts of the system. Requires that the persistence service is active.
* TimeLine - Provides the history features for the Apollo system. Needs to be up and running before the UI is started.
* UI - Provides the UI controls. Should be loaded eventually. The earlier it gets loaded the quicker the user can get started. Also loading early means that we can show progress.
* Plugin - Deals with the plugin loading.
* Project - Deals with the project. Depends on pretty much everything else.

The suggested load order may change slightly based on the dependencies between the different services.

Finally once the core system is up and running the bootstrapper objects will be released.

## AppDomain builder

The AppDomain builder is used to create AppDomains with all bells and whistles installed. The required items are:

* Security levels set. The level of security depends on the use of the AppDomain. E.g. only the initial and the persistence AppDomain are allowed to perform file I/O. The different security levels are:
  + Allow I/O
  + Allow showing UI elements
  + Allow creating AppDomains
  + Allow loading framework code
* Assembly resolution. Because it’s not always possible to set the private bin paths it is necessary to respond to assembly load failures and provide our own assembly path resolution methods.
* Error trapping. If an unhandled exception occurs in an AppDomain it should be caught to prevent the application from shutting down. Depending on the AppDomain it is possible to either:
  + Shutdown the AppDomain and start another one
  + Shutdown the application

It is important to remember that the error trapping and assembly resolution handling have to be setup from inside the newly created AppDomain, neither can be done from outside the AppDomain. This means that the AppDomain builder will have to have an initialization object which can be loaded in the AppDomain and then attach the correct behavior.

## Kernel

The Kernel object holds references to all the service objects and the AppDomains in which the individual services reside. The kernel is responsible for the loading and initialization and servicing of the different services. This means that the kernel takes care of the installing and uninstalling of services and also the starting, stopping and restarting of services.

The kernel also deals with all the requests for infrastructure from the services. In general these requests are:

* The creation of a new AppDomain for use by objects controlled by the service.
* The direct link to another service.

Finally the kernel is the part of the Apollo application that deals with all unhandled exceptions. The resolution of an unhandled exception is to either:

* Shutdown the service where the exception originated from and create a new instance of this service type. Thus creating a clean version of the service. This step can only be taken for the persistence service, the log service and the plugin repository. An unhandled exception in any of the other services requires a system shut down.
* Shutdown the system. This step is taken when replacing the service with a new version is not possible.

## Services

The main services, in order of loading, are:

* **Message service:** Is used to send information and requests between the services. Messages can be directed (from one service to another) or broadcast (from one service to whoever wants to know).
* **Core service:** Is used to allow the core elements (which are not services) to interact with the services on the service level.
* **User interface service:** Is used as the main entry point for the user interface. Holds most of the interface data structures and algorithms thereby allowing the real user interface to consist of nothing more than views and viewmodels.
* **License service:** Is used to validate the license keys and enable the other parts of the system to perform validity checks.
* **Persistence service:** Is used to write data to persistent storage (i.e. disk or network locations). This is also the only service that should be able to write to disk.
* **Log service:** Is used to log information to one or more event logs. Services can request the creation of specific logs or just log to the general application wide logs.
* **Timeline service:** Is used as the main collection source for all undo/redo capabilities. The timeline service tracks the order in which events happen and allows other services to track backwards or forwards along the timeline. Note that the timeline service does not store any of the data that is necessary to roll-back time.
* **Plug-in service:** Is used to track the available components.
* **Project service:** Holds all the project related capabilities.

All services are constructed so that they can be loaded in a separate AppDomain in order to ensure that services can be fully unloaded and restarted. Furthermore the use of separate AppDomains allows crashes to be partially contained thus allowing the service to be taken down without affecting the application too much.

### Message service

The message service provides message direction capabilities for the kernel of the Apollo application. Services can send messages directly to other services or they can broadcast a message to all services which are interested in getting the specific message type. All messages are posted to the message service which then forwards the messages to the desired recipients.

A message consists of a header, which contains the information required by the message service, and a body, which contains the information required by the recipient.

The header of a message contains the identification of the sender, the identification of the recipient and the ID number of the message. The identification of the different services is done by using a special name element which is unique to each service. Besides the services the message service also knows how to deal with messages send to all other services or no other service.

The body of a message contains the information the sender wanted to send to the recipient. It is important that this data is immutable and clonable so that messages can be copied and passed around without any hesitation. Besides this only a few types can be send through a message in order to prevent assembly loading failures when a sender tries to send an assembly specific type across.

Note: Because messages have to be pushed across AppDomain boundaries messages and their data has to be serializable.

In order to send or receive messages a service has to register with the message service. When registering the message service obtains a proxy to the service and the identifier of the service. Internally the message service maintains a mapping between the identification and the proxy, thus allowing it to forward messages and return message sending errors.

### Core

The core service provides a communication channel between the services and the kernel which controls the services. This communication channel allows services to send the following messages and requests.

* **Shut down request:** All services can request that the application is to be shut down. This message has extra data which specifies what kind of shut down (standard, error, crash) is requested. Services should be prepared for the shut down request to fail if it is a standard shut down.
* **Create new AppDomain request:** All services can request the creation of a new AppDomain for code separation reasons.
* **Restart request:** Several services can request that the kernel restart them in case of errors. The service will have the possibility to provide additional data which can be passed back after the restart.

The core service also enables messages to be sent from the kernel to the services. Services should be able to deal with the following messages:

* **Ping:** The system will ping all services upon request to see if they are able to respond to messages.
* **Confirm able to shut down:** In order for the system to perform a normal shut down all the services must confirm that they can shutdown. This allows services to cancel the shut down request. This ability is especially important for the project and user interface services which may have to check with the user if they want to cancel the shut down.
* **Shut down:** Requests the services to perform an orderly shutdown. This message cannot be ignored and any error conditions during the shutdown process will result in the abnormal termination of the service which experienced the error condition.

### Core UI

* Provides the basic interface to Apollo for User Interfaces. The Core UI provides all the necessary commands and data views that allow a UI to perform all the requested actions
* Provides a way for UI’s to work with Apollo without having to know the internal structure, or to have to store data about the project. UI’s should never have to keep their own data structures. Data should only be stored in the project and kernel sections.
* Provides the links for the appdomain info (e.g. search paths etc.). The kernel (or other parts) never load config files themselves. They ask the UI. This means we need messages:
  + Get data from configuration
  + Write data to configuration
  + Add assembly path
  + Remove assembly path

### License service

The license service takes care of the validation of the license. This validation occurs for two different paths in the application. The first path is the regular verification of the license. This verification can either be requested by another service whenever a special action is taken, e.g. when data is written to the hard drive. Regular verification also takes place on a timed basis. The license service has one or more timers running which validate the license each time they fire.

The second verification path is upon a request from one of the services for an encrypted code block. Each service can store one or more code blocks in the encrypted store which is controlled by the license service. When the service needs access to the code block it will request the code block from the license service. The code will be made decrypted if the license key is valid. Once decrypted the code can be transported (as actual code, not IL) to the AppDomain in question and then compiled and executed.

### Persistence service

TO BE DESIGNED

What about this one? It’s important to get it right. Versioning must play a big role in it.

* Persistence service can read files
* Persistence service is there to write the following files:
  + Configuration files
  + Project files
  + Command log files (keep track of the order of the commands and inputs)
* Files will be written according to a 'template'. e.g. configuration files always follow a specific format. Note that this template may only describe the way to put the data together, or the way to put data streams together.

Do we want this to live in its own AppDomain? Crossing domain boundaries is a heavy operation and can slow things down quite a lot. Maybe the persistence service should be virtual, allowing it create writers on request?

Problem with this is that we then need the permissions to do this. This is easy for the core but hard for the plug-in / project AppDomains

### Log service

TO BE DESIGNED

The log service provides logging capabilities to the other services in the kernel of the Apollo application. Services can log information directly by sending a message to the log service with the information that must be logged.

Should this be virtual? We expect quite a lot of messages, especially if we don't filter out the ones that we don't write first. Maybe we should have a log proxy in each service domain. The proxy handles the message creation and determines if the message should be send. If yes then it forwards it to the actual log service, which then writes the message.

Q: what does a log message look like?

Q: what information can a log message hold?

Q: where do log messages get written too?

Q: do we have different log levels? --> If so then we’ll need to figure out how to deal with creating a message that never gets logged. Want to make this very efficient but it all has to go across AppDomain boundaries...

Logger does not go through the persistence system. Instead it will do its own writing. For that we'll need permissions in the appdomain to write to a specific log directory ...

The pipeline will work the same way as the other services. This may mean that the pipeline has to be able to send messages the official way.

The logger should be able to handle multiple different message streams. e.g. there could be a debug log, a command log and other logs. Each log stream may have its own log format and destination.

### Timeline service

TO BE DESIGNED

* Stores information about the timeline followed by the application.
* The timeline describes in which order data was altered / generated or commands were executed.
* History information is tracked individually by the different participating parts. These are:
  + Plugin repository: indicates when a history breaking changes occur, due to changes in the plug-in assemblies
  + User interface: Marks time steps, tracks history of user elements
  + Project: Tracks history of the different data sets. Each data set holds their own history, project provides proxies to the timeline service
* The UI will indicate when a new time step is taken (thus collecting user actions).
* The data sets in the project hold their own data history.
* The timeline service allows running undo’s and redo’s based on the individual time steps
* To roll-back or roll-forward the timeline service sends a message indicating to which points the change should be made. The history proxies then calculate to which point each of the histories should return.

### Plug-in service

TO BE DESIGNED

* Tracks plugings
* Based on MEF(?)
* Allows installing plugins, but no run-time replacement
* Installation goes through our own system(?)
* Allows multiple plugin locations

### The project system

TO BE DESIGNED

* Based on data flow programming where the data ‘flows’ through different boxes that manipulate it.
* Have different generators for different types of data generation. Initially specify the simulation and experimental data generators
* A single experiment consists of data + generator + post-processing
* A single project consists of a tree of experiments. There are different ways of creating DAG nodes. Always have a single top-level node(?)
* Features and components are contained in the generator? How about data?
* Should data only specify how/which data is written to storage and how/which data is retrieved. Then the storage can be our custom implementation.
* Data should be easily retrievable (see data storage spec).
* Each project runs in its own AppDomain. Ditto for data-sets

### Ideas

Updating is done by detecting if there is an update and then linking them to the website to download the new installer. This simplifies greatly because:

* There’s no problem installing the new version if we’re running inside rhino because the installer wasn’t started by the app
* We can simply keep on using the MSI system which gives us all the advantages of MSI / windows installer
* We can still apply MD5 hashes + signing to detect corrupt / evil downloads?

Questions:

* Do we want everything to be a plug-in? If so how far do we go in this? Are we making the services plug-ins too?
  + We probably want most of it to be plug-ins however at the moment there is no good reason to make the services full plug-ins. It only complicates matters (e.g. how do we find the plug-ins when the plug-in repository is a plug-in itself). On the other hand it would be very useful if we could add services easily, aka the services need to be very loosely coupled and have a minimum of configuration.

Processes

Startup

Shutdown

A shutdown message is send to the core.

The core determines if this is a normal shutdown or a crash shutdown

If crash shutdown then write to the log and kill everything (shutdown all threads and kill the appdomains)

If normal shutdown then

* Send message to all services requesting actions for shutdown. This does not start the shutdown but just allows services to indicate if they can shutdown
* If all services can shutdown then send shutdown messages in proper order
* If not all services can shutdown then send message of cancelation to orginator