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 kernel 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:

* **Kernel:** bla
* **AppDomainBuilder:** Builds the application domains necessary for the different services

The main services are:

* **Core service:** This is used to allow the core elements (which are not services) to interact with the services on the service level.
* **Log service:** This 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.
* **Message service:** This 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).
* **License service:** This is used to validate the license keys and enable the other parts of the system to perform validity checks.
* **Persistence service:** This 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.
* **Timeline service:** This 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:** This is used to track the available components.
* **Project service:** This holds all the project related capabilities.
* **User interface service:** This 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.

All services will be 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 should allow crashes to be contained thus allowing the service to be taken down without affecting the application too much.

### Log service

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.

Q: what does a log message look like?

Q: what information can a log message hold?

### 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.

Q: How is a message defined?

Q: How is a message send?

Q: How do we know where to send a message?

Q: What happens when a message cannot be delivered? Do we throw an exception (in the senders domain)?

R: Messages can’t be extensible because then people could send non-primitive data across which would subsequently load the sending assembly into the receiving AppDomain which is what we were trying to prevent.

### License service

* Takes care of license key validation --> Which type do we use?
* Responds to requests from the other systems for validation checks

### Persistence service

### Timeline service

* 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

* 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

* 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).

### 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.
* ?

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?