Everis Design Document

For the attention of DG EMPL

Contacts

|  |  |
| --- | --- |
| Name: | Stijn Moreels |
| Job Title: | Product Developer |
| Mail: | stijn.moreels@codit.eu |
| Mobile: | +32 48 697 167 |

Table of Contents

[1 Purpose 4](#_Toc459786913)

[2 Layered Architecture 5](#_Toc459786914)

[3 Infrastructure Layer 6](#_Toc459786915)

[4 Application Layer 7](#_Toc459786916)

[5 Presentation Layer 8](#_Toc459786917)

[6 Frameworks 10](#_Toc459786918)

# Purpose

The purpose of this document is to describe the global technical design of the Everis project (a.k.a. AS4 project). Some critical core design points are explained to have a global awareness of the software design of the project.

# Layered Architecture

The AS4 application is separated in several layers. Since the Core is the essential part of the application, it could also be seen as the **Domain Layer** of the project. Like any layered architecture, all the dependencies go in one direction only.

* The **Infrastructure Layer** contains core functionality and provides the services (and interfaces) to the upper layers. The core definitions of the AS4 protocol are also describes in here.
* The **Application Layer** is actually the layer that uses the core to provide a system for the presentation layer. This layer contains besides the delegation of functionality also the mappings that’s needed to provide a connection form the Business Application to the MSH.
* The **Extension Layer** contains all the agents that are directly coupled to the Business Applications. The agents can be extended to have a personalized AS4 adapter for your communication. The Steps and Receivers are exposed interfaces and used steps inside the core that can be extended by the user.



# Infrastructure Layer

In the Infrastructure Layer, the core functionality of the AS4 protocol is described. What’s really important is the **Serialization** of the message so it can be send and retrieved from other parties. The message itself (*AS4Message*) has two methods to perform the serialization/deserialization can be called from the *Send Agent*. The message itself asks at the *Global Registry* to retrieve the right *Serializer* (all have to be registered in this Global Registry). These serializers are registered with the Content Type by which they perform the serialization (MIME or SOAP), this way the client (in this case the Send Agent) has only to know which content type the incoming AS4Message has.

The functionality described in the serializers itself calls somethings external parties (like the *Soap Envelope Builder* or the Mapping functionality described inside the AutoMapper Profiles, See 6 Frameworks) to have the right functionality to perform the serialization.

**C:\Users\stijn\Downloads\EVERIS(3).png**

The *Global Context* is responsible for the context in which the application run, this includes the **PMode** configuration, the **Settings** file… Since we use only this single context, there’s only a single truth at a single place where these items can be retrieved for those who needed.

The *DataStoreContext* uses it for the connection string, the *Transformers* uses it for the right pmode, the *Receivers* uses it for the configuration settings of the receivers (and all these items are found in the settings file or pmode files).

# Application Layer

The application layer is all about delegation and knowing where the functionality is placed instead of having the functionality described itself. That’s why you’d find only the responsibility to create *IAgent* implementations from the settings file (where the different types are described).

So this layer contains the startup and the responsibility of the creation of the Agents.



# Extension Layer

The whole development process of the project has the *Extendibility* in mind, and this layer for sure. Almost every *Service* has to be extendable for the users: Agents, Transformers, Receivers, Steps…

The following diagram shows the different relationships between each component. All the interfaces are exposed by the Core and are not part (of course) of the presentation layer, but for readability purposes these are included.

**

When you look at this diagram you see that the *Agents* are actually the central point of the system and wraps all the needed functionality. The *Agent Base* is an abstract class to have a consistent *Open()* method for all its child’s. When clients want to write an *Agent* from scratch, they can use the *IAgent* interface (this interface is used everywhere and not the abstract class).

Because messages can contain *Attachments*from different locations (in the same message) we have to retrieve these *Attachments* in different ways. The only problem is that we don’t know the exact way where these are stored and the way they must be retrieved up front.

The solution is a combination of two patterns: *Strategy* and *Registry Pattern*.



In this scheme we have two important concepts that are combined together: the *Strategy* interface uses the *IRegisterEntry* interface; this interface has one method to get the key of the *Entry*. This way each *Strategy* is responsible of the way it’s get registered. The other concept is the *Registry* itself which wraps all the *Entries* and exposes/register these *Entries*. With this approach we could allow clients to write their own implementations of the *IRegisterEntry* interface and exposes them when they need them.

# Frameworks

The following frameworks are used to outsource certain tasks:

* Theirs need for several mappings between objects. When messages arrive at the agents, when the messages leave the agents… **AutoMapper** is used to perform these mappings in a clean way. All *Profiles* (mapping schema for a mapping between two types) are using the same naming convention: [name of the mapping]Map.
* The project contains several Exception Strategies so logging is also a required functionality. All the loggings are centralized by the **NLog** framework. This framework is called by the *Singleton* that the framework provides.
* When messages arrive or leave, some validations are required. To have a solid and future-proof system, we’ll use **Fluent** to validate each object. This framework requires and apart class to perform the validation and so the naming convention [name of the type]Validator.