JJ's Reference Architecture

Author: Jan-Joost van Zon
Date: December 2014 – July 2017
[Under Construction]

Service Architecture

Contents

Introduction

What has been described so far is the *application architecture*. A second part of the software architecture is the *service architecture*, which is mainly about linking systems together. This section is an addition to the documentation with regards to the service architecture. Currently the services are programmed using WCF.

The ESB Concept

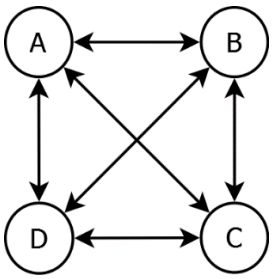
ESB stands for Enterprise Service Bus, which is a system for exchanging data between different systems of different organizations in different formats with different protocols. Central components are used to make integration between these systems more manageable. One important concept is the canonical model.

Canonical Model

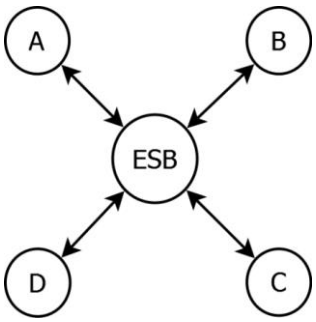
The canonical model helps us exchange data between systems. Data can be retrieved from multiple systems and is converted to a canonical form, so that the same code may be reused for data that comes from various systems. The canonical model should be as pure and general as possible, so indeed information of any system can fit into it with very little modification.

Less Integration Code

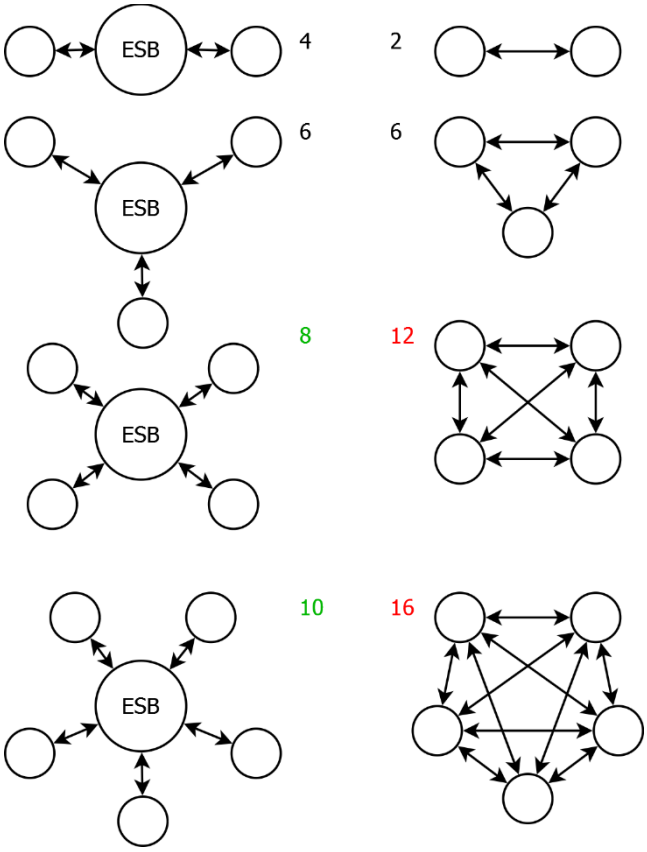
Say you have 4 systems: A, B, C and D and you want to connect all 4 of them together. Theoretically you would have to write 12 different message conversion as you can see from the arrows in the diagram below:



By connecting a system to the ESB, instead of connecting individual systems together, you have to implement only 8 different message conversions as you can see from the arrows below:



You just saved yourself 33% of the work!
With every system you add to your ESB it gets better as you can see from the numbers below that indicate the amount of message conversions.



The first integration between 2 systems you program using your ESB you actually program more message conversions, but with the next system it is already a tie between ESB and no ESB. The 4th integration you introduce you will have saved 33% of the overall work.

It gets better with each system you introduce in your ESB. When messages from a system are converted to and from the canonical model, you can automatically connect it to all the other systems.

Clearer Integration Code

But it gets better. You save yourself even more work. The conversion code from message to canonical model is often easier than converting from one system's format to the other system's format, because instead of converting from one quirky format to another quirky format, which is really difficult to do, you convert from one quirky format to a very clear format, which is much easier to program.

In Practice

In practice not every system sends every type of message back and forth to every other system. And sometimes the messaging is not bidirectional but one way only. But the benefits of the ESB still hold and you will link systems together with less code and less effort than custom programming every integration between two systems.

Standard ESB vs Custom ESB

There are standard Enterprise Service Bus software packages available. Yet, we choose to build a custom one ourselves. The concepts are not that hard to implement. And generic ESB's are really complex and have a steep learning curve, require training, specialists. This all while you are going to have to custom program much of the message conversion code yourself anyway, and design your own canonical model, which is basically all of the work. Therefore we build it ourselves.

ESB Model

On top of a canonical model, we need more facilities. The ESB model will offer a model for administrating connection settings and register enterprises that can log in to our system to get access to our services.

Next will be listed the main entities of this model.

Enterprises

Every enterprise involved in our service architecture is registered in our ESB database. Some of these enterprises will actually log into our system. Those will get an associated User entity with (encrypted) credentials stored in it.

ConnectionTypes

Every type of connection between systems is registered in a table of ConnectionTypes. Each ConnectionType is a very specific way of integrating with a system, with a specific messaging protocol, message format and implementation.

Connections

Every individual connection between two parties is registered in the Connection table with the connection settings stored with it. Each connection has an associated ConnectionType that indicates what type of integration it is. Note that some connections are not between parties but involve only one party. Connections do not have to be complete messaging implementations. Sometimes they are simply database connection settings or even the path of a network folder.

Keys

Often systems have different identifiers for e.g. orders or other objects. There is often a need to map a reference number from one system to the reference number of another system. The ESB model has entities and logic to manage these key mappings.

Transmissions

Optionally you can log the transferred messages that went over a connection. Do note that logging all messages can significantly impact performance and storage requirements so use it sparsely.

Service Implementations

The implementation of a service involves mostly message transformation and transmission. Data is received through some communication protocol, the message format is parsed and

then converted to a canonical model. Conversely, canonical models are converted back to a specific message format and the sent over a communication protocol.

Multi-Dispatch

The content of a canonical model can determine what service to send it to. For instance, one canonical Order might have to be sent to one supplier using their own specific integration protocol, another order might simply be e-mailed to the supplier. The service architecture enables you to retrieve a message from one system, for instance an order, and then send that message to an arbitrary other system. That is part of the power of the canonical model, where multiple systems' messages being converted to canonical model, enables all those systems to communicate with eachother.

Namespaces

These namespaces use a hypothetical Ordering system as an example.

JJ.Services	Root namespace for web services / WCF services
JJ.LocalServices	Root namespace for windows services. (Not part of the service architecture, but this is where that other type of service goes.)
JJ.Data.Canonical	Where are canonical entity models are defined.
JJ.Data.Esb	Entity model that stores Enterprises, Users, ConnectionTypes, Connections, etc. Basically the configuration settings of the architecture.
JJ.Data.Esb.NHibernate	Stores the Esb entity model using NHibernate.
JJ.Data.Esb.SqlClient	SQL queries for working with the stored Esb entity model.
JJ.Business.Canonical	Some shared logic that operates on canonical models.
JJ.Business.Esb	Business logic for managing the Esb model.
JJ.Services.Ordering.Interface	Defines interfaces (the C# kind) that abstract the way messages are sent between different ordering system. These interfaces use the canonical models.
JJ.Services.Ordering.Dispatcher	Makes sure messages (orders, price updates) are received from and sent to the right system depending on message content.
JJ.Services.Ordering.Email	A specific implementation of an ordering interface, behind which we send the order by e-mail.

JJ.Services.Ordering.SuperAwesomeProtocol	A specific implementation of an ordering interface, behind which we implement the hypothetical 'super awesome protocol' for sending orders.
JJ.Services.Ordering.Wcf	A WCF service that allows you to communicate with the multi-dispatch ordering system.
JJ.Services.Ordering.Wcf.Interface	Defines the interface of the WCF service that allows you to communicate with the multi-dispatch ordering system. This service interface can is used by both service and client.
JJ.Services.Ordering.Wcf.Client	Allows code to connect to the WCF service using the strongly typed service interface.
JJ.Services.Ordering.JsonRest	Exposes the multi-dispatch ordering service using the Json and Rest protocols.
JJ.Services.Ordering.WebApi	There is no reason Web API should not be involved in this service architecture, in fact, the idea of WCF being the default for service, might not be a very long-lived.
JJ.Presentation.Shop.AppService.Wcf	A special kind of service is an app service, that exposes presentation logic instead of business logic and returns ViewModels.

Service-Related Patterns

Facade

An interface behind which a lot of other interfaces and classes are used, with the goal of simplifying working with these systems.

This concept is used in this architecture to give a service interface an even simpler interface than the underlying business logic has. It may hide interactions with multiple systems, and hide infrastructural setup.

Hidden Infrastructure

Not so much a pattern, but a difference in handling infrastructure setup between the application architecture and the service architecture. Iin the application architecture the infrastructural context is determined by the top-level project and passed down to the deeper layers as for instance repository interfaces or interfaces on security, while in the service architecture the infrastructural context is determined by the bottom-level project. At least in case of multi-dispatch this seems necessary. A bottom-level project, for instance JJ.Services.Ordering.Email does not expose that there will be smpt server setup. You cannot see that from the constructor or interface at all. The service will handle all that internally.

<TODO: Code example.>

TODO

<TODO: Service Architecture: Three-Stage KeyMapping vs. Custom KeyMapping is something to write about in Architecture Details. It explains why you do not need an even more generic Key mapping, for intance:

- CustomerOrderNumber
- SupplierOrderNumber
- InternalID

Cannot remember how and why it worked exactly.>