1.5 Edge Component

The last pattern of the basic patterns is the Edge Component pattern. Unlike the other pattern which are basic just because they are very common, the Edge component is also basic because it is a platform for implementing other patterns. Because the Edge Component pattern is a step in implementing other patterns it is a little hard to come up with a concrete example that shows exactly why it is needed since the concrete examples fit the patterns the build on Edge Component. Instead I'll try to introduce few short examples and the communalities between them will lead us to the Edge Component.

1.5.1 The problem

Scenario 1

On one company I worked for we developed a Military Naval C4I platform. This platform had services that are reusable per solution. For example one of the core services provides a unified and centralized view of targets. The first implementation that was built on that platform used a messaging infrastructure using Tibco Rendezvous®. The next implementation has to use a different technology altogether (WSE 3.01). Both implementations had to use the same business logic but needed different technologies to access that logic.

Scenario 2

On another project (that is also mentioned in the Workflodize pattern) - a cellular carrier constantly wanted to introduce new usage plans and offerings such as friends and family, night rate and the like to a service that handled ordering . The service interface remained pretty stable as the changes in the details were part of the XML but the business logic kept changing and adapting to the new plans.

What we see here is the opposite of what we had in scenario 1, here the interface and technology are stable and the business logic changes

Scenario 3

The last scenario is is a common situation in many projects. You normally have more than one service in a system. Each of these services handles a different business aspect yet all of them have to perform common tasks like making sure a request is authorized before performing a request, saving an audit trail etc.

---

[1] Microsoft interim solution for the WS-* stack before Windows Communication Foundation

In this scenario we have a functionality that is not related directly to a single service and is mostly repetitive across services – as it takes pretty much the same code to log the request even it on service handles orders and the other handles customers

The commonality between these scenarios is that we have different concerns (business logic, technology, logging etc.) all bundles within each service. As we've seen in the different scenarios, each of these concerns can change independently the others depending on the circumstances – we need a way to enable that flexibility so our problem is

> **How do we allow the business aspects of the service, technological concerns and other cross-cutting concerns like security, logging etc. to evolve in their own pace and independently of each other?**

The simplest, not to say simplistic, option is not to do anything in particular. An example for this approach is taking a piece of logic and directly exposing it as a web-service[2] which by the way, is very common in on-line samples technology vendors such as Microsoft (WCF) or Sun (JAX-WS) provide on their tutorials. However, when the handling of the contract is directly intertwined with the business logic implementation, the maintainability of the code greatly suffers – for example it would have been very hard to support scenario 1 and replace technology using this approach.

We could try to solve the problem of replacing a technology for an existing service by making a new copy of the service and making the changes there . An approach also known as "own and clone". The problem is, however this creates a maintainability problem as you now have multiple copies of the same business logic lying around and you'd have to make changes to all copies not to mention that it doesn't solve problems such as the ones presented in scenario 3 of adding logging capabilities to several services.

If not doing anything and cloning don't work maybe we can go for separation of concerns

1.5.2 The solution

Separation of concerns is very known concept in the object oriented thinking. The root principle behind it is known as the "The Single Responsibility Principle" or SRP for short. SRP states that a class should have only one reason to change and that a responsibility is such a reason. We can apply the same principle within SOA and consider the business logic as one responsibility and the other concerns as another responsibility we get the following pattern:

> **Add Edge Component(s) to the service implementation to add flexibility and separate between the business logic and the other concerns like contacts, protocols, end point technology and additional cross-cutting concerns**

[2] Web service – is a method that is exposed over http. This is an unfortunate term since it overloads the word service for something that can easily be abused to create

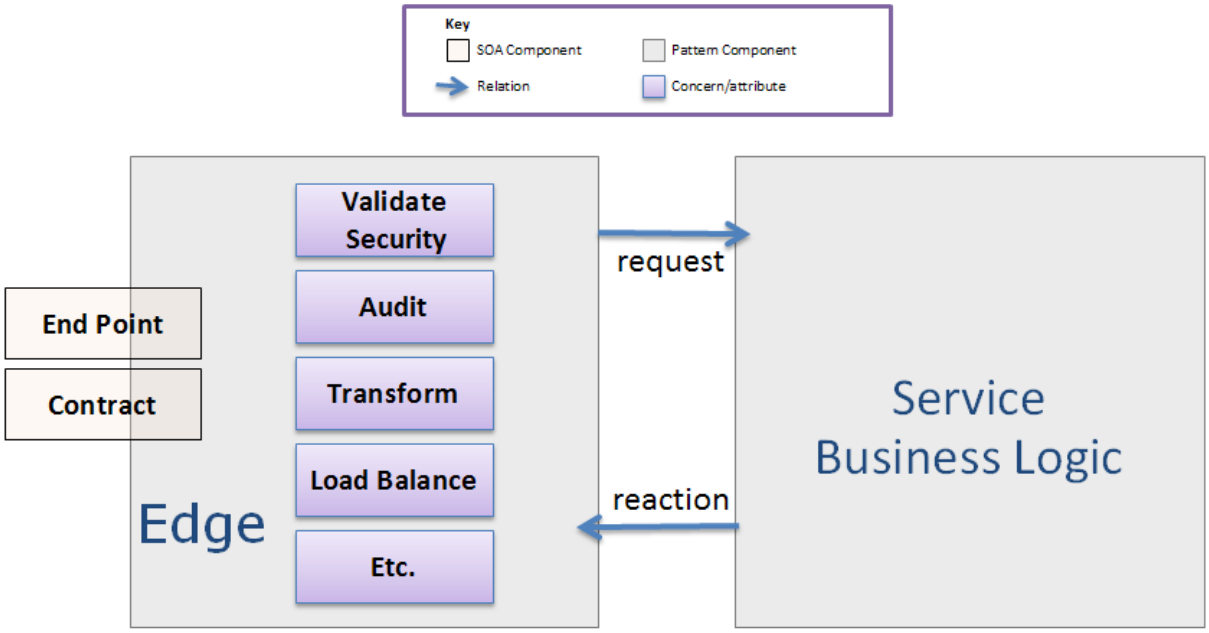

Figure 2.11 The Edge Component pattern. The pattern means adding a component in from the business itself and having that new component, called an Edge, handle cross-cutting concerns like load balancing, auditing etc. Adding an edge allows the service itself to focus on the business logic and not clutter it with unrelated stuff.The edge component delegates request and replies between the service and its consumers.

As Figure 2.11 above demonstrates, the main idea behind adding an Edge Component is separation of concerns. The Edge Component takes care of all the cross-cutting concerns and other concerns that are not in the core business of the service. These concerns can include areas such as load balancing, format transformations, auditing. The business logic of the service is then handled in another component that focuses solely on the business logic and remains free of the other concerns. This separation allows supporting all the scenarios mentioned above since the separation allows each component to evolve in its own pace. For example to support a new technology (scenario 1) you just add an additional edge component but the business logic doesn't have to change. When you change the behavior of the business logic and add a new usage plan (scenario 2) the Edge component doesn't change.

In a sense the Edge Component pattern can be used to provide a façade, proxy and AOP patterns for SOA.

We still have to show how the problem in scenario 3 of implementing cross cutting concerns across services can be takes care of. The best approach to do that is to take single responsibility principle further and remember that the Edge Component is, indeed, a component and it doesn't have to map to a single "monolith" class. For example you can apply a pipes and filters architectural style and chain several classes/component, each dealing with a specific concern, to create incoming or outgoing pipe-lines For example, the figure 2.12 below shows a sample Edge Component that applies a validation filter to make sure the message is correctly formatted. Then there's a transformation filter to translate an external contract format into an internal one. Lastly there's a routing filter to send the message to the correct component within the service. These sub components can be reused from service to service as needed as well as change and evolve independently.

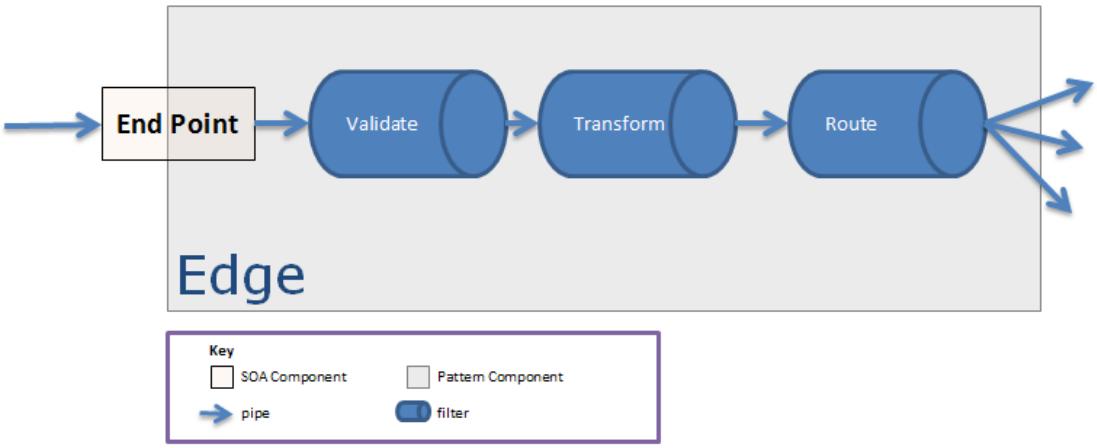

Figure 2.12 Sample Edge Component pipe-line. As an incoming message is received on the endpoint it goes through validation, transformation to an internal format and routing to the correct component within the service.

While it is tempting to define an inner contract between the Edge Component and the Service at the onset, there is no real reason to do that unless, maybe, you have to support multiple external contract (be weary of implementing a contract per consumer though – see PTP Integration anti-pattern). When the service evolves and newer versions of the contract are created like in scenario 1 of adding a new technology, you may have to add inner contract when you still want to support the older versions of the external one.

The edge component is very useful and I've introduced it in most of the SOA projects I architected. Many of the structural patterns mentioned in this book expand and build on the Edge component pattern.

Let's take a look at the technological aspects of the Edge Component pattern.

1.5.3 Technology Mapping

As mentioned in the patterns structure in Chapter 1, the technology mapping section takes a brief look at what does it mean to implement the pattern using existing technologies as well as mention places technologies implement the pattern.

There is no technology that will take care of all the concerns the Edge component can fulfill automatically. The up side is that no technology, I know of, prevents you from implementing the Edge Component pattern and some technologies will even handle some of the concerns for you.

For example, JAX-WS or Windows Communication Foundation (WCF) basically implement the Edge Component pattern for you, but they only handle lower level concerns, which they sometimes call bindings. The concerns handled by JAX-WS and WCF are those mentioned in the various WS* standards, for example, WCF can handle MTOM encoding or Security for you. However as I already mentioned,you still need to code high-level concerns like routing, contract translations etc. by yourself.

An interesting technology option is a Java engine called Restlet®. Restlet® has a few built in classes that sets it as a good example for implementing the Edge Component Pattern

Edge Component Example – the Restlet® Engine

The Restlet® engine by Noelios Consulting, which is a Java library for implementing RESTful services, has some built is classes like filter and router that allow for easily building an Edge Component. Consider the example in the diagram below:

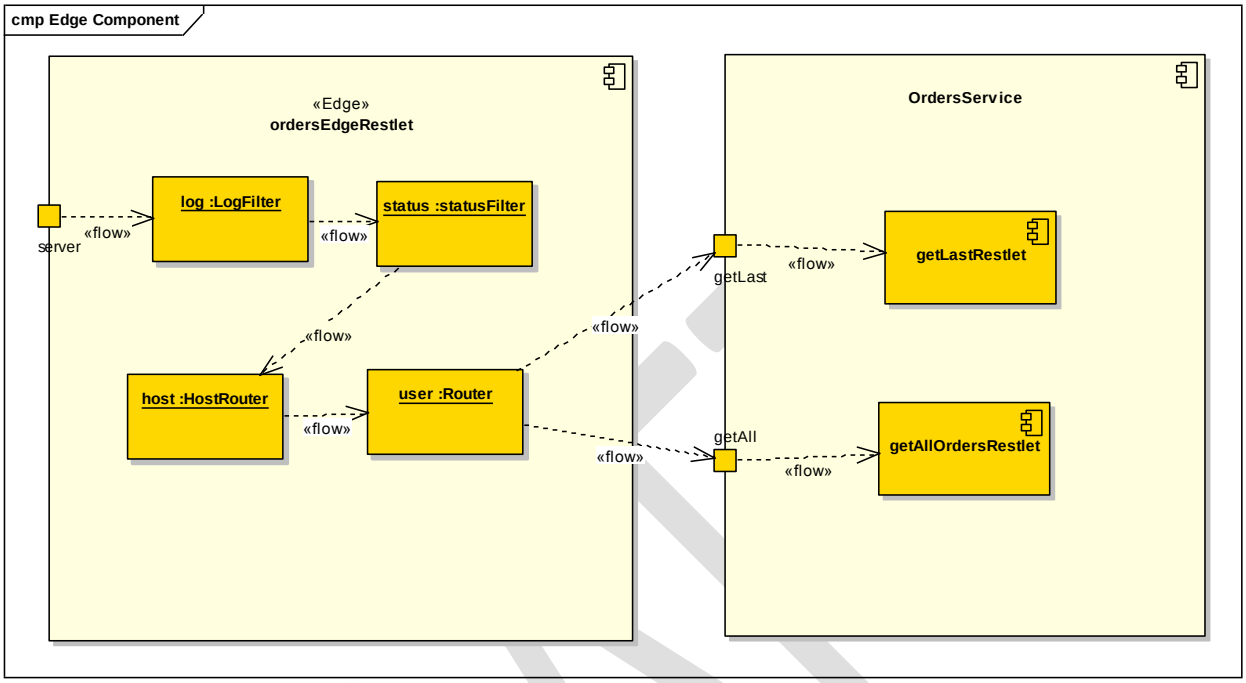

Figure 2.13 Implementing Edge Component in Restlet®. We see an Implementation of an edge component by using some of the Restlet® components such as HostRouter, Router, statusFilter and LogFilter. As a request is received it gets routed through and handled by the different components before it gets to the actual business service.

Figure 2.13 shows a possible Edge configuration on an Orders service whose contract has two operations: getLast which returns the last order and getAll which returns all the orders for a specific customer. However, before the call actually makes it to the business logic we have to log it, handle statuses or problems, make sure the correct host was used and finally route the call to the appropriate business functionality. Adding an Edge component lets us achieve all that without affecting the business logic which just processes the business requests.

Here is an excerpt of the above mentioned configuration in code :

```
Builders.buildContainer()

        .addServer(Protocol.HTTP, portNumeber)

        .attachLog("Log Entry")

          .attachStatus(true, "webmaster@mysite.org", "http://www.mysite.org")

                  .attachHost(portNumber)
                          .attachRouter("/orders/[+")
                                .attach("/getAll$", getAllRestlet).owner().start();
                                .attach("/getLast$", getLastOrderRestlet).owner().start();
```

Code excerpt 2.2 : except of an instantiaion code for defining an edge component using Restlet®

As we've seen The Edge Component pattern is supported by all current technologies and even implemented internally by some of them. You can take a look at the Further reading section at the end of the chapter for links to resources that expand on the technologies mentioned in this section.

1.5.4 Quality Attribute Scenarios

The quality attribute scenarios section talks about the architectural benefits of utilizing patterns from the requirements perspective. As was mentioned in chapter 1, most of the architectural requirement are described from the perspective of quality attributes (scalability, flexibility, performance et.) – through the use of scenarios where these attributes are manifested. The scenarios can also be used as reference for situations where the pattern is applicable.

The Edge Component pattern can be associated with a lot of quality attributes. Most of these attributes however are the result of applying the Edge Component Pattern along with another pattern. There are however 2 quality attributes that are directly related to the Edge Component Pattern. The first is Flexibility – making it easy to change and enhance the external properties of the service without affecting the business logic. The second is maintainability – separation of concerns makes it easy to understand what each component is doing. Recall the three sample scenarios – adding a new technology to an existing service, changing business behavior without changing the contract and solving cross cutting concerns only once - with the Edge Component in place, we were able to solve the problem without affecting the rest of the solution or at least by minimizing it. Table 2.2 below shows a couple of sample scenarios that can direct us for using the Edge Component.

Quality Attribute (level1)	Quality Attribute (level2)	Sample Scenario
Maintainability	Backward Compatibility	As contracts evolve, the services should be able to support consumers using older versions of the contract (at least the last 2 revisions)
Flexibility	Extension points	Within the next year the customer is expecting to need SOX compliance and add auditing across the board

Table 2.6 Edge Component quality attributes scenarios. The architectural scenarios that can make us think about using the Edge Component pattern.

The Edge Pattern is the last of the basic structural patterns for SOA. To wrap this chapter up, let's take a look at the patterns that were covered - before we take a look at additional structural pattern for availability and scalability patterns.