Eclipse Swordfish — an Open Source SOA Runtime Framework for the Enterprise

Whitepaper

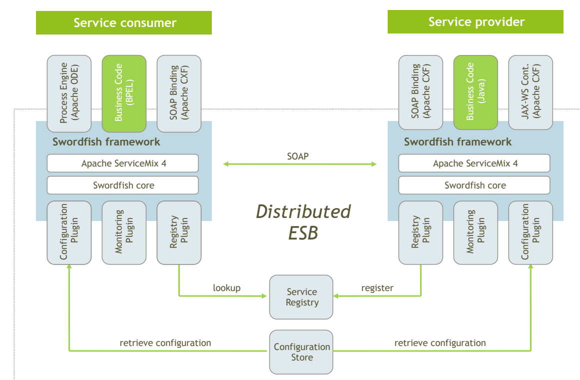
Oliver Wolf (SOPERA GmbH), February 2009
oliver.wolf@sopera.com

The Enterprise Service Bus (ESB) has always been the cornerstone of every vendor's SOA strategy. In the past, however, ESBs have often failed to help leverage on the full value of SOA. This was due to their large footprint, the centralized architecture, and the considerable effort required to properly integrate them into an existing application landscape.

Swordfish — the next-generation ESB

Swordfish takes on a vastly different approach. Built on proven open source components such as Apache ServiceMix and Apache CXF, Swordfish provides an extensible framework that allows application developers and system integrators alike to build their own ESB that can be tailor-made to their requirements. But Swordfish is more than just a framework. It follows the Eclipse tradition of providing "both extensible frameworks and exemplary plugins". The Swordfish project also aims at delivering Enterprise-grade plugins, turning it into a full-fledged open source ESB that takes the "E" in ESB seriously. In order to illustrate the advantages of Swordfish over other open source ESBs, we will highlight some of the features that have grown out of many years of SOA experience gained in real-world enterprise settings.

The first feature that we are going to take a closer look at involves integrating a Service Registry and dynamically binding services at runtime. What this basically means is that in Swordfish — in contrast to many other ESBs — the relationship between components using a service (consumers) and components providing a service (providers) need not be static. In fact, a consumer refers to a provider by means of a logical identifier that clearly signifies the service's interface together with a statement of the consumer's non-functional capabilities and requirements which is called a policy. Given these two pieces of information and its own database of existing service providers along with their respective policies, the Service Registry selects a matching provider and calculates an effective policy that governs all future communication between consumer and provider. The advantages are quite obvious: consumers and providers are even more loosely coupled than before and their non-functional characteristics can be easily changed without touching a single line of business application code.



Another notable point about Swordfish is that it is based on an architectural pattern that is often called "Distributed ESB" or "Federated ESB". What it basically means is that in order for two service participants to communicate with each other, no central components are needed after the initial communication set up. The advantage of this approach over „classical" Hub-and-Spoke integration architectures is quite obvious: There are no central components that are likely to become a performance bottleneck over time, so the whole system is much more scalable and grows with the users' requirements without forcing them to expand their hardware investment as they progress on their path to SOA adoption.

The potential downsides of a federated approach – increased administration and management complexity – are compensated for by a remote configuration mechanism that allows administrators to configure a large number of distributed Swordfish instances

conveniently with efficient and extensive monitoring capabilities.

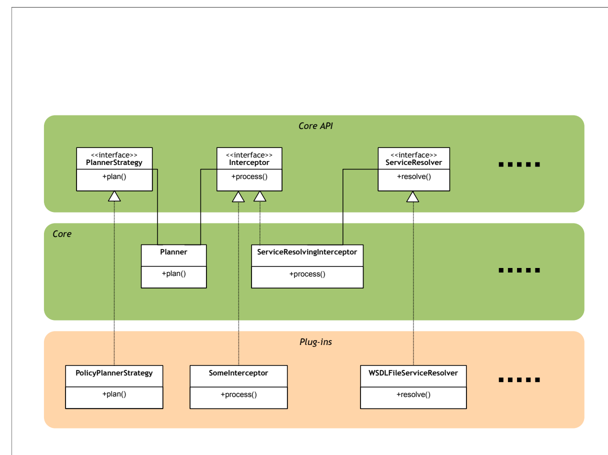
Monitoring is especially important in the context of Business Process Management (BPM), since fine-grained monitoring events are a prerequisite for a comprehensive Business Activity Monitoring (BAM), often based on a Complex Event Processing (CEP) approach.

OSGi and JBI inside

The Swordfish framework is based on standards that are relevant in the SOA space today. At the lowest level there is the OSGi, or more specifically, Equinox, the Eclipse Foundation's OSGi implementation. OSGi provides among other things the component model, a deployment mechanism for modules and a clean class loading system. On a higher level, Swordfish is uses a subset of the JBI standard for messaging abstraction and message routing between components. Originally, JBI 1.0 included its own component model and deployment mechanism, but this was dropped in favor of OSGi. This is very much in line with what used to be consensus in the JBI 2.0 working group and what has already been implemented in version 4 of Apache ServiceMix, Apache's JBI container, which serves as the core message routing engine for Swordfish. However, ServiceMix does not only provide routing; it also includes a large number of ready-made components that can be used out-of-the-box to interface to business logic, or transport channels, and protocols.

So, in a nutshell, Swordfish is a framework that builds on ServiceMix and extends it with features that are required for Enterprise SOA. The Swordfish core uses public extension points in ServiceMix in order to hook into the message flows inside the Normalized Message Router (or NMR).

The central concept behind hooking into the NMR lies with the Interceptor, which is a piece of code that intercepts message exchanges as they go through the NMR and operates on them in some way. The Java interface that represents an Interceptor is part of the Swordfish API and can be implemented using plugins that contribute Interceptors for special purposes, for example, message validation, transformation to name a few.



The order in which the available interceptors are actually applied to a message exchange is defined by a processing plan. This plan is calculated by a component of the Swordfish framework core called, the Planner. The exact method by which planning is performed is again something that should be user extensible. For this reason, there is an interface in the API called, PlannerStrategy that plugins can implement. One example of such a strategy could be an interface that knows how to evaluate a Policy that is included inside the message exchange. In addition to custom interceptors there are some special purpose interceptors, for example an interceptor that looks up a service in a Service Registry and re-routes the exchange accordingly. The basic (JBI-related) behavior of this interceptor is part of the core implementation, but the details of how exactly the look-up is performed can be supplied by a plugin that implements the ServiceResolver interface in the framework API. The same pattern is applied throughout all the public APIs.

Registry & Repository for Application Developers

We have already touched on the merits of using a Service Registry to resolve service endpoints at runtime in the previous passages. However, a Service Registry does

more than just that. It essentially provides a comprehensive overview of the services that exist within a Service-oriented Architecture and plays a key role in fostering service reuse, which is one of the main benefits of the SOA paradigm.

The real power of a Service Registry becomes obvious when services are composed into more complex services by means of a process orchestration facility. If the service interfaces have been carefully defined with reuse in mind, creating a complex composite service can be merely a matter of moving and connecting boxes in a graphical editing environment.

In the future, the Service Registry will be complemented by a Service Repository. This Service Repository allows all SOA-related artifacts that are created as part of the SOA artifact workflow (such as, service interface descriptions, policies, to list a few options) to be stored in a reliable and traceable manner. Features like versioning, the possibility to define approval workflows and analysis, as well as reporting capabilities enable enterprises to set up and enforce the governance processes that are often deemed crucial for the successful adoption of SOA.