Service Oriented Architectures and the Universal Application Interface: a World Wide Web for Applications

Universal Front End Interface enables true component based integration

dr. Arjen Schoneveld
drs. Matthijs Philip
Dexels
October 2003

Introduction

The popularity of Service Oriented Architectures (SOA) has increased strongly in the past few years, mainly due to the rise of web services. In this paper we briefly address the essentials of SOA by looking at its evolution. To do that, we simply have to look at the challenges developers have faced over the past few decades and observe the solutions that have been proposed to solve their problems.

In the stone age of software development, programmers realized that writing software was becoming more and more complex. They needed a better way to reuse some of the code they were rewriting. When researchers took notice of this, they introduced the concept of modular design. With modular design principles, programmers could write subroutines and functions and reuse their code. This was great for a while. Later, developers started to see that they were cutting and pasting their modules into other applications and that this started to create a maintenance nightmare; when a bug was discovered in some function, they had to track down all of the applications that used the function and modify the code to reflect the fix. Both developers and project sponsors didn't like that; a higher level of abstraction was needed.

Researchers proposed classes and object-oriented software to solve this, and many more, new problems were introduced. Again, as software complexity grew, developers started to see that developing and maintaining large software projects was complex and they wanted a way to reuse and maintain functionality, not just code. Researchers offered yet another abstraction layer to handle this complexity: component based software. Component-based software is/was a good solution for reuse and maintenance, but it doesn't address all of the complexities developers are faced with today. Today, we face complex issues like distributed software, application integration, varying platforms, varying protocols, various devices, the Internet, etc. Today's software has to be equipped to handle all of those requirements.

The rise of the Internet with the appearance of the World Wide Web has given us the opportunity to use remote applications with a a wide variety of technical signatures in a uniform matter. The browser became a *Universal User Interface* (*UUI*) resulting in a *user-centric* integration environment between different applications on different platforms. The enabling technology was HTML and the underlying HTTP transport protocol. The technological path could have continued along the same way as the integration of mainframe applications did, that is through the use of screen scraping techniques or complicated HTML connectors. These hypothetical historical events would have resulted in a huge complicated and inscrutable integration piramid with high a very risk of having far from desirable performance. However, now that we still have a choice, we want application integration technology to go in another direction where only the *concept* of the Universal User Interface is reused instead of the entire technological protocol stack. What is needed is a *Universal Application Interface* (*UAI*), such that (front-end) applications instead of people can easily and uniformly access other applications. We want to *lower* the Universal Interface from the end-user to the end-user application, enabling, among other benefits, a clear separation of the great variety of *user interface logic,* for all kinds of devices, and the actual *functional logic* offered by a multitude of application providers.

A Service Oriented Architecture seems to be the right framework for our Universal Application Interface (UAI). A SOA, by definition, is service-centric and a such the ideal abstraction layer between the user interface logic and the functional logic. Just as HTML has been the enabling technology for the WWW, the enabling technology that has emerged for a SOA is called XML. By adopting a SOA, you should be released of the headaches of different protocol and platforms and your applications *could* integrate seamlessly. That is you pass these headaches to others that have to deal purely with connecting to legacy applications, different database engines, EJB's, .NET, work flow packages, or any other software component/application. However, we are far from a true UAI just by adopting XML and some Service Oriented Architecture. Before we continue to discuss the requirements for a Universal Application Interface let's define a SOA in a bit more detail.

The Service Oriented Architecture

The first step in learning something new is to understand its vocabulary. In the context of SOA, we to define the following terms: *service*, *message*, *dynamic discovery*, and *web services*. Each of these plays an essential role in SOA.

Service

A service in SOA is an exposed piece of functionality with three properties:

- The *interface contract* to the service is platform-independent. *Basically the interface definition is our key player in realizing a true UAI layer!*
- The service could be *dynamically located and invoked*.
- The service is *self-contained*. That is, the service maintains its own state.

A platform-independent interface contract implies that a client from anywhere, on any OS, and in any language, can consume the service. Dynamic discovery hints that a discovery service (e.g., a directory service) is available. The directory service enables a look-up mechanism where consumers can go to find a service based on some criteria. For example, if an application was looking for a credit-card authorization service, it might query the directory service to find a list of service providers that could authorize a credit card for a given fee. The fact that services are self-contained reflects the fact that a service is *independent from other entities.*

Message

Service providers and consumers communicate via messages. Services expose an interface contract. This contract defines the behavior of the service and the messages they accept and return. Because the interface contract is platform- and language-independent, the technology used to define messages must also be agnostic to any specific platform/language. Therefore, messages are typically constructed using XML documents that conform to XML *schemas*. XML provides all of the functionality, granularity, and scalability required by messages. That is, for consumers and providers to effectively communicate, they need a non-restrictive type of system to clearly define messages; XML provides this. XML does not provide a generic definition of a message, just a structure space to define such a definition. The industry standard that has emerged for such a definition is called *SOAP*.
However, also SOAP does not provide a clean cut definition of a UAI interface contract, it is not restrictive enough to unambiguously define such a contract. Also note that a proper definition of a UAI interface would basically render SOAP obsolete due to the fact that a uniform UAI definition could also include basic header definition to define requested function and some sender identification.

Dynamic Description and Discovery

At a high level, SOA is composed of three core pieces: service providers, service consumers, and the dynamic directory service.
The role of providers and consumers is apparent, but the role of the directory service needs some explanation. The directory service is an intermediary between providers and consumers. Providers register with the directory service and consumers query the directory service to find service providers. Most directory services typically organize services based on criteria and categorize them. Consumers can then use the directory services' search capabilities to find providers. Embedding a directory service within SOA accomplishes the following:

- Scalability of services; you can add services incrementally.
- Decouples consumers from providers.
- Allows for hot updates of services.
- Provides a look-up service for consumers.
- Allows consumers to choose between providers at runtime rather than hard-coding a single provider.

Web Service

Although the concepts behind SOA were established long before web services came along, web services play a major role in a SOA. This is because web services are built on top of well-known and platform-independent protocols. These protocols include HTTP and XML. It is the combination of these protocols that make web services so attractive. Moreover, it is these protocols that fulfill the key requirements of a SOA. That is, a SOA requires that a service be dynamically discoverable and invokeable. This requirement is fulfilled by UDDI, WSDL, and SOAP. SOA requires that a service have a platform-independent interface contract. This requirement is fulfilled by XML. SOA stresses interoperability. This requirement is fulfilled by HTTP. This is why web services lie at the heart of SOA.

Defining a Universal Application Interface in Web Service Oriented Architecture

The Universal Application Interface *layer* is solely responsible for the orchestration of services and for the "*presentation*" and "*processing*" of Front End applications requests. An application submits a request to perform an operation to the UAI layer which processes the requests and generates the response. Be aware of the fact that the difference between a Remote Procedure mechanism and the UAI is determined by the granularity of the function and the ease of use due to a different abstraction layer resulting in much simpler semantics.

Basically exactly the same concept is used as a human user requesting an HTML page and submitting HTML forms resulting in yet another HTML page. Note the duality between a request (GET) and a form submittal (POST) which are merely technically different implementations of the same concept. A clean orthogonal implementation of the UAI layer should implement this as a single implementation in the UAI layer. A UAI layer implementation is conceptually similar to a classic web server serving *server side scripting pages* (JSP, PHP, ASP) that process the request and generate the response. However, these techniques implement a Universal *User* Interface which has different requirements, among others the need to define layout and style. Whereas a UAI implementation is basically layout and style independent and compared to the HTML interface language, a UAI interface request/response language should be based on a minimal generic interface that is simple to use and enables flexible methods of processing a request and generating a response. The UAI layer logic that we aim for could best be described as document-centric. The analogy of paper documents processed by some office clerk upon request of a client and the generated response in the form of another (semi)-complete document is in place here. Or even more explicit it should be able to represent a complete but abstract notion of some sort of (user) interface.

We identify four major components that a SOA UAI layer should implement:

- *a UAI message language* (or UAI documents)
- *a UAI to UI transformation language*
- *a UAI to Other Applications transformation/processing language*
- a set of *supporting aspects*

A generic UAI language

An important requirement is that we want the level of abstraction of the UAI language, which implements the ***interface,*** between the UAI layer and the front end applications as closely related as possible to the actual User Interface that is presented to the end user. A UAI document describing a service request or response should be close to a one-to-one mapping to a user interface component.

A self-contained electronic form has the following semantic requirements:

- *Platform independent*
- *Easy to understand*
- *Grouping* of data
- *Naming* and describing data items
- Definition of *input* and *output* data.
- A basic notion of *data types* (strings, integers, dates, but also selections or enumerations).
- Possible *actions*, defined as subsequent web services to support an (interactive) work flow.

A UAI to UI mapping language

Since the level of abstraction of our UAI is close to that of a normal User Interface, it enables us to augment our generic UAI SOA with yet another layer that maps the UAI to an actual user interface instance (which again could be HTML).
Such a generic and easy to use language should have the following characteristics:

- *Platform independent*
- *UI independent* (i.e. Java Swing, HTML, Flash, etc.)
- *Extensible* with user defined UI components
- Support for *multiple threads* to improve responsiveness
- Support for *event listeners* and accompanying *actions*
- *Conditional logic* with support for user defined functions
- And of course, ability to easily produce and consume UAI web services

General UAI architecture aspects

We identify the following aspects in our UAI SOA architecture:

- *Protocol Listeners.* A request to the UAI SOA could be sent over different protocols ranging from raw data sockets, RMI to SMTP. However, most commonly used protocol will probably be HTTP (especially in firewall rich environments).

- *Authentication*. The architecture should support multiple user defined authentication methods, e.g., username/password, tokens, certificates.
- *Authorization*. Authorization somehow links “users” (i.e. front end applications) to UAI web services. That is, it specifies whether some web service may be accessed. As for authentication, an open modular way of using any authorization method is mandatory.
- *Validation*. Besides simple XML validity and correctness the content should be validated against (possibly) user specific *rules* (a set of conditions that has to be met in order for the request to be valid). The business rules complement the *logic* that is implemented in subsequent layers.
- *Document augmentation*. The request can be enriched using (possibly) user specific data necessary for further processing.
- *Caching*. Static or slowly varying function responses are could candidates for caching. Caching of a response parameterized by the request should be part of our architecture.
- *Logging*. Access information should be logged as error exception logging.
- *Testability.* The UAI web services should support automated testing by means of some UAI web services unit testing framework, specifically enabling easy regression testing.
- *Documentation.* Some sort of automated documentation mechanism is vital for reportability on- and transferability of the implemented web services.

A UAI processing language or a UAI to Other mapping language

Our aimed UAI processing language should be able to process requests and generate a response as easy and flexible as possible. Any UIA web service is implemented using this language.
Typically the processing language should have access to any application and data source through some connector based concept. Furthermore, such a language would preferrably also be defined in XML for the sake of uniformness, the availability of XML editors and for automated generation of some sort of WSDL document that could “describe” our function and “open” it up to the *general standardized* SOA world.

- Support for user defined *connectors*
- Conditional logic with support for user defined *functions*
- *Iteration*
- A *stateless* execution model (at least in principle). The absence of client side state enables trivial scalability of the architecture. The phrase “in principle” is deliberately used because it should be interpreted loosely in the case of:
- Support for *Asynchronous* execution. Semantics for asynchronicity are required for example for long lasting transactions.

Note that we require a minimal language that is not necessarily universal in the computational sense.

Conclusion

Even though we have had decades of experience in software development, we have yet to solve the mysteries of software complexity. As complexity grows, researchers find more innovative ways to answer the call. SOA, in combination with web services and a Universal Application Interface (SOA UAI), is the latest answer. Application integration is one of the major issues companies face today; a SOA UAI can build on that. System availability, reliability, scalability continue to be major issues for IT dependent companies; SOA UAI addresses these issues. Business application presence over multiple-channels is another such issue, SOA UAI with it's generic messaging concept lays a foundation for multi-channel strategies. Given today's requirements, the multiple layers of a SOA UAI is the best scalable solution for modern application architectures.

At *Dexels* we have implemented such a SOA UAI framework: *Navajo*. The entire scope of the Navajo product lines covers just about everything that is discussed in this position paper. It's two main constituents are:

1. The *Navajo kernel*, implementing a SOA UAI server side framework application that is completely implemented in Java. It contains a number of standard protocol listeners for HTTP and SMTP. Furthermore, a set of basic connectors is included for accessing databases (Oracle, Sybase, SQL server), mail and some common files like .XLS and CSV formats.
2. *Navajo Tipi*, which implements a generic UAI to UI mapping using a special XML language TipiML for defining user interfaces and it's connection to Navajo web services.

For more information about the Navajo framework we refer to “Navajo developer Documentation”.