Samantha Niroshan Kahadawa

Telephone : 0714-311669
Address : No. 26, Horagollagama, Nittambuwa, Sri Lanka

.

E-mail : samanthan@inbox.com

++

Event-Driven Architecture: Achieving Architectural Agility

Event Driven Architecture (EDA) can seem quite esoteric, but it provides unique benefits and capabilities that can enable business systems to become more agile and responsive. What's even better is that you don't have to abandon any of your hard-learned SOA lessons since EDA and SOA can not only happily coexist but also thrive by leveraging each other's strengths.

Event-based systems are systems in which the constituent modules (or components) have no direct connection to each other. I agree that this statement by itself is quite vague. So, let's explore it further using a typical service-oriented, N-tiered Web application.

An end user submits an order form using a Web browser, which results in a boundary service on the server side being invoked. This boundary service then finds and invokes the appropriate business components, which in turn may invoke a series of other business and data components, and so on. Each component in this system, including those on the client side, knows exactly which component (or set of components) it needs to invoke next, and furthermore, this logic is built into the application itself rather rigidly. Changing this logic in most cases entails a developer making a change to the application code.

The reason for this is simple: components in such a system are directly connected (or coupled) to each other. There are two generally accepted ways of establishing this connection between the two components. The more common approach is through the use of a factory pattern. (There is actually a third way to establish the connection. The "caller" component could simply create the "callee" component itself. This is the least desirable way of the three and should be avoided in all except the simplest of systems.)

What Sets An Event-Based System Apart

An event-based system is different from the system described above in that the components never really know about each other. More specifically, the components don't communicate by invoking methods on each other. There are no "caller" and "callee" components as was the case in the scenario described above. Rather, in an event-based system the components communicate by publishing and subscribing events. So in an event-based system we have "publisher" and "subscriber" components instead of "caller" and "callee" components. And just as in the scenario described above where the same component could be a "caller" and a "callee", the same component in an event-based system can be both a publisher and subscriber of events. Finally, multiple components may publish an event of the same type just as multiple components can subscribe to an event of the same type.

A more technical description of the difference between the system described above and an event- based system can be offered in terms of the degree of coupling between the constituent components within the system. In the system described above, the degree of coupling between the components varies from being very tight to very loose depending on which one of the three strategies was used to form the connections between the components (with the least desirable, third strategy creating the tightest coupling). In an event-based system, the components are completely decoupled from each other, since they communicate only through the publication and subscription of events and never directly with each other.

A system that is architected from ground up to be event-based is said to have an Event Driven Architecture (EDA). At its core, an EDA based system is implemented on a special class of middleware that supports at least the following three functions:

--Allows a component to register the events that it can publish.

--Allows a component to see what events are being published by other components and subscribe to the event it is interested in.

--Provides efficient event handling. For example, there is no need to broadcast an event published by a component that no other component is interested in.

The middleware may also support other features, such as transactions that span multiple components that are publishing and consuming an event or allowing event publishers to attach security criteria/requirements to their publications that subscribers must fulfill to subscribe to the event. The middleware would ensure that all subscribers to such an event satisfied the security requirements specified by the publisher.

Common Misconceptions

As is the case with the popular acronym SOA, EDA has a number of misconceptions associated with it too. One prevalent myth is that there is an EDA product. EDA is *not* a product that you just go out and buy on the open market. Rather, EDA is an architecture pattern that describes a style for implementing solutions.

Another common myth is that EDA and SOA are opposites. That is not true either. Despite the complexity of a properly implemented SOA, at its heart, an SOA based system is a set of business services with each service having a well-defined interface contract. However, an SOA imposes no restrictions on how these services get invoked. That is, there is nothing preventing these services from waiting for certain events to occur within the system and as the events occur the services execute their business logic. Thus, each node (i.e. publishing or subscribing component) in an EDA can be implemented as an SOA.

The Benefits of EDA And SOA Together

So, what are the benefits of combining EDA and SOA? A combination of the two architectural styles continues to provide all the commonly touted benefits of an SOA, such as increased levels of reusability, higher interoperability between systems, and improved scalability. In addition to these benefits, an EDA allows dynamic reconfiguration of business logic by the addition of new or the replacement of existing event subscribers. For example, if a new federal regulation requires that all banking transactions over $1M be recorded in a new federal registry that is monitored by the Dept. of Homeland Security, then a new subscriber that satisfies this need can be added to the system that subscribes to the events that are published whenever a transaction occurs. Code changes to the existing system are not required and it continues to function in the same way as it did before. This is possible because, as I described previously, the application logic in an EDA is not programmed as a series of connections; rather it is implemented by publishing and subscribing of decoupled events. Thus, an EDA by its very nature supports many-to-many connections in multiple dynamic, parallel, and asynchronous paths through the system.