Adopting SOA Best Practices and Lessons Learned

Salil Ahuja

July 15, 2009

Moving to a Service Oriented Architecture (SOA) brings many benefits to businesses, allowing for greater solution alignment and agility. Making this a smooth transition requires a special level of focus on quality and an awareness of the unique challenges related to testing within an SOA. Often, the adjustments that need to be made to testing competencies are not apparent or planned. Organizations need to understand the unique goals and challenges related to evolving services architecture and the implications of how testing should be performed. In this article we will discuss the quality assurance challenges to be addressed with a SOA adoption, recommended best practices and lessons learned.

Introduction

It is tempting to treat projects involving a Service Oriented Architecture or (SOA), just like any other previous project. However, by considering some of the unique testing challenges presented on projects, it is possible to optimize testing practices to better match the projects specific QA requirements. The first step is to make sure that development and test teams are familiar with all the aspects of the project that leads to unique challenges. These aspects can include:

1. Simple, easily consumed, complexity hiding interfaces
2. Open-ended future reuse
3. Technical complexity of implementing with each supporting system
Figure 1. Heterogeneous connections and complexity

5. Logic within service flows that can differ from the underlying application logic
6. Security needs of services differing from the supporting applications security

Figure 2. Logic and security in multiple places

7. Services as separately managed IT components

The challenges related to testing can arise as a result of each of these attributes, both in isolation and in combination. This article discusses some of the commonly encountered challenges, as well as the best-practices that can be used to alleviate the pain.
SOA Testing Unique Challenges

In SOA environments the test team goes beyond traditional application centric functionality and performance testing. SOA requires integration testing of interfaces and services that help collaborate and orchestrate diverse systems and platforms along with related security standards.

This presents unique challenges when it comes to testing these applications.

Services may not have a user interface

Traditionally applications have a GUI or related user interface that can be leveraged to perform functional test either manually or automatically. In an SOA, environment services may not always have an interface. The testers work with messages and network protocols of heterogeneous technologies such as SOAP and Web services. Further exposed services encapsulate the underlying process and hence ensuring sufficient test coverage can be a challenge.

We can use standard based test tools like SOAPUI to address challenge when working with Web services. These tools provide the ability to generate test clients based on the Web Service Description Language (WSDL) file of the service which then helps testers to understand the operations of the web service and invoke the service.

Data gathering

A single SOA environment can leverage different technologies and different data formats from different internal and external stakeholders. This then requires the QA team to validate the integration of all these technologies and data formats. A very critical component in this exercise is gathering the right data prior to deployment to execute all the test scenarios. If the test data is gathered before hand it ensures the QA life cycle will be not be disrupted with the possibility that data is unavailable for certain test cycles.

This requires more extensive design phase and SOA Governance with collaborative involvement from the QA team and related parties. SOA Governance is a central team that presides over the complete SOA effort to manage and monitor the effort. The test team should identify their test data needs and collaborate with involved stake holders to get data as the development of the service proceeds. SOA Governance process should ensure that needs of the QA team are met. These practices ensure that when the service is ready for integration testing the QA team will have the sufficient data they need.

Higher level of integration requires better planning and strategy to address/availability problems

In the more traditional approach a tester can expect the application or the business functionality be delivered through a single project on a single application server exposed through a Web interface. With an SOA, application logic is located within the middle tier, operating with any number of technologies, residing outside the department, or even outside the company. This makes end to end testing, even in the test environment, dependent on third parties. If a particular critical third party system in unavailable it has the potential to disrupt a QA cycle if the QA team is not prepared. The QA team must be prepared for all situations.
For the QA teams to run test cycles with such interruption, it is critical that all stakeholders are proactive in communicating and planning their downtimes. Tools such as SOAPUI and Rational Application Developer (RAD) are available to create mock placeholder services to help accommodate downtime and unavailability of certain services.

Mock services can enable QA teams to reduce dependency on stakeholders while running end to end test. The QA team can create placeholder services to behave similar to the actual services. These services are generated by using the actual WSDL and supported schemas of the original service. The mock service is then provided with a set of responses. These responses are used to respond to requests made to the mock service. The responses can be random, round robin or based upon certain rules or policies.

A mock service can then be used if the original service for any reason is unavailable. This helps in ensuring that QA cycles are not adversely affected by downtime of certain services. Testers can proceed to create services, once the service interfaces have been defined, to achieve the end to end solution without waiting for the actual service to be developed.

Mock services also help to isolate defects while debugging issues and investigating the source of the issue. As mock services are very controlled, they help create more controlled environments.
especially while diagnosing defects. In this way testers can drive different end-to-end scenarios and have lesser dependencies on possible issues at third party locations. We recommend a mock service be generated for each Web service in the project.

Security Challenges

Given the nature of SOA environments that leverage independent services, maintaining security is even more challenging. The security solution must address all of the typical security aspects - i.e. authentication, authorization, correct identification, message integrity, non-repudiation, confidentiality/privacy and auditing support. This requires a careful establishment of trust relationships, typically using the WS-Security standards. Awareness and skills with the standards, protocols and conventions used is critical to being able to test the security measures.

Also, web-services are web-based and, thus, require a similar diligence as any web solution. The vulnerabilities that tools such as Rational AppScan [5] test for can still apply.

General negative test cases, for security vulnerabilities, must be included. The list would include tests to ensure that:

1. replays are denied, i.e. a copied message that is resent later should not be accepted
2. man-in-the-middle attacks are not possible
3. messages are confidential and cannot be altered without notice
4. only legitimate users can access the services (including the denial of spoofing schemes)

Need to ensure excellent exception management

When different services interact together via a common layer it becomes essential to test validate exception conditions and ensure that the code uses standard based specifications to handle edge/negative process conditions. Business process flows help model sequence of business events collaborating underlying services. If any one service does not respond and fails to respond in a desired manner, the process should be robust to handle such conditions and provide notifications of such failures. As the process is tested, it is essential these edge conditions are identified and tested.

Reuse challenges

Reuse is one of the key benefits of an SOA. One single service can be used by several applications or maybe a few at the beginning of the adoption of SOA, but soon, it could happen that some services will be consumed by a dozen or more applications. If those services have not been designed and tested to perform under such load, some of the applications using them might suffer from unacceptable response times. And, in the worst case where services have not been thoroughly tested and fail in production, the impact of that failure can be catastrophic for the whole organization, as several applications would be affected at the same time.

Learning curve

Some of the technologies and standards such as SOAP, Web services, XML, BPEL and WS-Policy are continuing to evolve. Adoption of these technologies has been slow and therefore when
enterprises take on modernization efforts they need to account for the learning curve involved. The QA team members need a deep understanding of these technologies and their benefits that will enable them to effectively test SOA solutions.

**Best Practices**

**Perform SOA vision and technology training**

As described in the introduction, an awareness of the common aspects and expectations for SOA are important to help focus the QA effort. Thus, IT teams should be educated on the overall scope of the SOA environment, including what areas of the business are involved, what systems are involved in providing services and what systems are planned to be new service consumers. They should also be aware of the standards, guidelines and governance that have been put in place with the initiative.

As well as SOA initiative education, the teams should be comfortable with the Web service technologies required for testing. This would typically include XML (structure, namespaces and schemas) and WSDL.

Thus, plan for training and education in these areas is an ongoing part of the SOA initiative.

**Maintain automated service integration test suites**

Core strategic services have an expectation for easy consumption in conjunction with increased reuse over time. Their simple interfaces must hide the complexities of integration across the underlying backend systems. Thus, there is a higher risk to service quality that must be contained. A change within any layer of a service implementation stack could cause a regression in the service capabilities. In such situations, the regression should be caught early and should be isolated quickly to determine and implement the necessary adjustments prior to the next release of systems updates.

To manage each strategic service quality, it is recommended that an automated suite of tests be maintained along with the service (see [2] for specific guidelines on creating this suite with composite services). The suite needs to be executable as needed, with little or no setup time required. The suite should "hit" the main components within each layer of the service stack.

A related governance practice is to only certify a service for use once it has been verified that its automated integration test suite exists and is being maintained.

**Adopt a service testing framework**

To automate and maintain service integration test suites, there are some common capabilities that must be developed and reused. These include:

1. The ability to produce test harnesses in the absence of an application UI
2. Generation of test messages, based on the service description (WSDL)
3. Variation of inputs, using a data table
4. Data set-up and tear-down scripts
5. Output of test reports
6. Definition of expected results
7. Execution of tests against each integrated layer of the stack (usually via a unit test environment)
8. Emulation of external services (mocks)
9. Inspection and validation of service messages from consuming applications
10. Sending multiple test messages via separate threads

Such capabilities are packaged within an Integrated Test Framework (ITF). The framework is typically made up of commercial or open-source tools in combination with customisation or tailoring to meet the specific environment needs.

Rather than repeatedly implement these capabilities with each service, the ITF should be maintained as a separate reusable asset, containing the commonly used utilities, tools and scripts. It is recommended that you start with a Web service functional testing tool (see [3] and [4]) to base the framework on.

**Figure 5. Integrated Test Frameworks (ITF)**

![Integrated Test Framework Diagram](image)

**Use mock services to test consumers ahead of integration with providers**

Mocks allow for testing when the actual provider is not available. This helps with the service test development, but it can also help to isolate consumer tests prior to integration with the full service.

To allow for imperfect project timing, such as when a service is still under development and the consumer needs to proceed with other tests, mock services can be hosted in place of the real services. Producing mock services should be a capability of the ITF, for use in such situations.
Use test suites to adequately test services ahead of consumer integration

During service development, an evolving version of the automated service integration test suite should be developed. This will ease the effort necessary for problem determination and debugging. If this is not done, it becomes more difficult to determine if a problem is due to the consumer(s) or the provider(s).

Always plan for performance testing

Performance testing to gauge the scalability and responsiveness of new services should always be part of the plan. Performance testing with new consumers should also be considered, since different consumers can have different load characteristics.

The ITF should include the ability to scale up the number of parallel requests.

Early and ongoing measurements of service implementation performance should be performed. It cannot wait until the end, since design changes can be costly.

Allow adequate time for security setup and validation

With service security implementations, a number of lessons should be kept in mind:

1. It can be difficult to debug service handshakes, when the solution is secured from end-to-end, such as with token profiles
2. Negative case test scenarios are much more prevalent than positive case (such as message being recognized and processed)
3. With end-use UIs, develop test cases that validate what these users can do against the new solution vs. what they are allowed to do in the supporting applications
4. The ITF needs to be able to invoke secure services
5. Only a subset of the WS-Security specifications is interoperable across vendor and open-source framework implementations. Thus allow time for workarounds
6. Tests with external partners or consumers should plan to include consumer-side message and parameter logging, to narrow down debugging efforts
Design for instrumentation and logging capabilities

In order to quickly identify and narrow down on problems within SOA implementations, especially when involved in complex process flows, the use of logging is recommended. The logging should be switchable to multiple levels of detail and should enable correlation of tracing information across systems, such as via the logging of accurate timestamps or header identifiers.

Employ environment smoke testing

An SOA solution often relies on many integration components, such as Enterprise Service Bus (ESB) and gateways. Given the many moving parts, testing of these components, ahead of full deployment within each environment, should be business-as-usual testing. It becomes more difficult and time-consuming to narrow down issues if the environment and integration components have not been validated first.

Adopt Test Driven Development practices in general

Several of the practices outlined here can be categorised as general Test Driven Development (TDD) practices. TDD recommends beginning testing early and continuously testing, such as with each system build. As the systems evolve, there is the need for different forms of test harnesses, including the ability to mock or stub out parts that are not yet ready for prime time.

Given the complexity and coordination requirements found in many SOA projects, it is not uncommon to see incremental or iterative development practices in use. Using TDD principles is recommended in order to complement such projects by allowing them to get instant test-based feedback on the latest changes.

Carefully plan out the test strategy

The test strategy outlines project or release details such as the testing responsibilities and types of tests to perform. It should include plans for training, building out integrated service test suites, independent consumer testing, performance testing, security and dependency management (for the common scenarios where dependent components are not released in manageable steps).

To improve consistency, it is recommended that guidelines be used for the default practices in each potential scenario. This would include:

- Performance testing and measurement with each new service
- Building and executing integrated testing suites with each new service
- Executing the integrated testing suite with each new set of system updates
- Hosting early mock services or test versions of actual services with new consumers

Detailed test planning guidelines are also now integrated within current releases of SOMA[1], an IBM methodology.