

PARASOFT LEARNING SERIES

Load Testing Web Services

Automate SOA Performance Testing



Load Testing Web Services

Automate SOA Performance Testing



Parasoft Learning Series

Introduction.....	1
Choosing a performance testing approach	1
The "traditional" or "leave it till later" approach	1
The "test early, test often" approach	2
The "performance test automation" approach.....	2
Creating an automated performance testing infrastructure.....	3
Automating the build-test process.....	3
Automating performance test results analysis	4
Collecting historical data.....	5
Creating performance test scenarios: general guidelines.....	5
Improving the diagnostic ability of performance tests	6
Increasing performance tests' coverage	6
Type of Use	7
Emulation Mode	8
Content Type	8
Type of Load.....	8
Creating a real-world value mix.....	10
Wrap up.....	10
About Parasoft	11
About Parasoft SOAtest and Load Test.....	11
Contacting Parasoft.....	12

Introduction

The successful development of scalable Web services requires thorough performance testing. The traditional performance testing approach—where one or more load tests are run near the end of the application development cycle—cannot guarantee the appropriate level of performance in a complex, multi-layered, rapidly-changing Web services environment. Because of the complexity of Web services applications and an increasing variety of ways they can be used and misused, an effective Web services performance testing solution will have to run a number of tests to cover multiple use case scenarios that the application might encounter. These tests need to run regularly as the application is evolving so that performance problems can be rapidly identified and resolved. To achieve these goals, Web services performance tests have to be automated. Applying a well-designed, consistent approach to performance testing automation throughout the development lifecycle is key to satisfying a Web services application's performance requirements.

This guide describes strategies for successful automation of Web services performance testing and provides a methodology for creating test scenarios that reflect the real-world environment. To help you apply these strategies, it introduces best practices for organizing and executing automated load tests and suggests how these practices fit into a Web services application's development life cycle.

Choosing a performance testing approach

Performance testing approaches can be generally divided into three categories: the "traditional" or "leave it till later" approach, the "test early test often" approach, and the "test automation" approach. The order in which they are listed is usually the order in which they are implemented in organizations. It is also the order in which they emerged historically.

The "traditional" or "leave it till later" approach

Traditionally, comprehensive performance testing is left until the later stages of the application development cycle, with the possible exception of some spontaneous performance evaluations by the development team. Usually, a performance testing team works with a development team only during the testing stage when both teams work in a "find problem – fix problem" mode.

Such an approach to performance testing has a major flaw: it leaves the question of whether the application meets its performance requirements unanswered for most of the development cycle. Unaware of the application's current performance profile, developers are at risk of making unwise design and architecture decisions that could be too significant to correct at the later stages of application development. The more complex the application, the greater the risk of such design mistakes, and the higher the cost of straightening things out. Significant performance problems discovered close to release time usually result in panic of various degrees of intensity, followed by hiring application performance consultants, last-minute purchase of extra hardware (which has to be shipped overnight, of course), as well as performance analysis software.

The resolution of a performance problem is often a patchwork of fixes to make things run by the deadline. The realization of the problems with the "leave it till later" load testing practice led to the emergence of the "test early, test often" slogan.

The "test early, test often" approach

This approach was an intuitive step forward towards resolving significant shortcomings of the "traditional" approach. Basically, it works by reducing the uncertainty of application performance during all stages of development and by catching performance problems before they get rooted too deep into the fabric of the application. This approach promoted starting load testing as early as application prototyping and continuing it through the entire application lifecycle.

However, although this approach promoted early and continuous testing, it did not specify the means of enforcing what it was promoting. Performance testing still remained the process of manually opening a load testing application, running tests, looking at the results, and deciding whether the report table entries or peaks and valleys on the performance graphs meant that the test succeeded or failed.

This approach is too subjective to be consistently reliable: its success largely depends on the personal discipline to run load tests consistently as well as the knowledge and qualification to evaluate performance test results correctly and reliably. Although the "test early, test often" approach is a step forward, it falls short of reaching the logical conclusion: the automation of application performance testing.

The "performance test automation" approach

The performance test automation approach provides the means to enforce regular test execution. It requires that performance tests should run automatically as a scheduled task: most commonly as a part of the automated daily application "build-test" process.

In order to take the full advantage of automated performance testing, however, regular test execution is not enough. An automated test results evaluation mechanism should be put into action to simplify daily report analysis and to bring consistency to load test results evaluation.

A properly-implemented automated performance test solution can bring the following benefits:

- You are constantly aware of the application's performance profile.
- Performance problems get detected soon after they are introduced due to regular and frequent test execution.
- Test execution and result analysis automation makes test management very efficient. Because of this efficiency gain, the number of performance tests can be significantly increased. This allows you to:
 - Run more use case scenarios to increase tests coverage.
 - Performance test sub-systems and components of your application in isolation to improve the diagnostic potential of the tests.
- Automated test report analysis makes test results more consistent.
- Your performance testing solution is less vulnerable to the personnel changes in your organization since both performance tests and tests success criteria of the existing tests are automated.

Of course, implementing performance test automation has its costs. Use common sense in determining which tests should be automated first, and which come later. In the beginning, you may find that some tests can be too time- or resource-consuming to run regularly. Hopefully, you will return to them as you observe the benefits of performance test automation in practice.

Once you've made a decision to completely or partially automate Web services load testing in your organization, it is time to consider the principles of how your performance test infrastructure will function.

Creating an automated performance testing infrastructure

Automating the build-test process

A continuous or periodic daily/nightly build process is common in forward-looking development organizations. If you want to automate your performance tests, implementing such a process is a prerequisite. Figure 1 shows the typical organization of a development environment in terms of how source code, tests, and test results flow through the automated build-test infrastructure.

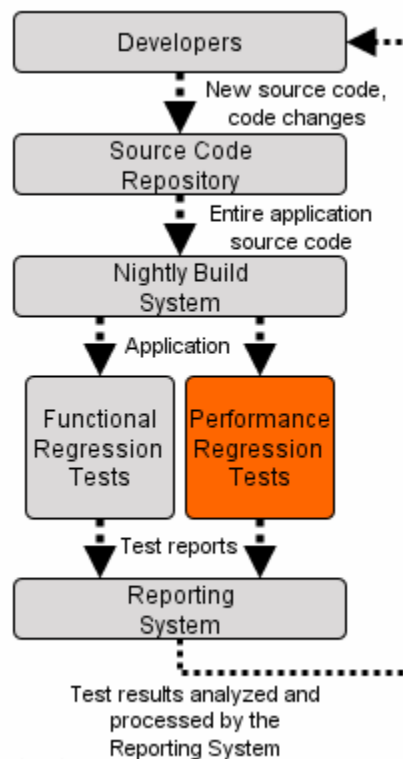


Fig 1. Automated performance testing as a part of the application development cycle.

It makes sense to schedule the automated build and performance test process to run after hours—when the code base is stable and when idling developer or QA machines can be utilized to create high-volume distributed loads. If there were failures in the nightly performance tests, analyzing the logs of your source control repository in the morning will help you isolate the parts of the code that were changed (and which likely caused the performance degradation). It is possible that the failure was caused by some hardware configuration changes; for this reason, keeping a hardware maintenance log in the

source control repository will help to pinpoint the problem. Periodic endurance tests that take more than 12 hours to complete could be scheduled to run during the weekend.

Automating performance test results analysis

In a traditional manual performance testing environment, a quality assurance (QA) analyst would open a load test report and examine the data that was collected during the load test run. Based on system requirements knowledge, he or she would determine whether the test succeeded or failed. For instance, if the CPU utilization of the application server was greater than 90% on average at a certain hit per second rate, the test should be declared as failed.

This type of decision making is not applicable to automated performance testing. The results of each load test run must be analyzed automatically and reduced to a success or failure answer. If this is not done, daily analysis of load test reports would become a time-consuming and tedious task. Eventually, it would either be ignored or become an obstacle to increasing the number of tests, improving coverage, and detecting problems.

To start the process of automating load test report analysis and reducing results to a success/failure answer, it is helpful to break down each load test report analysis into sub-reports commonly called quality of service (QoS) metrics. Each metric analyzes the report from a specific perspective and provides a success or failure answer. A load test succeeds if all its metrics succeed. Consequently, the success of the entire performance test batch depends on the success of every test in the batch:

- Performance test batch succeeds if
 - Each performance test scenario succeeds if
 - Each QoS metric of each scenario succeeds

It is convenient to use performance test report QoS metrics because they have a direct analogy in the realm of Web services requirements and policies. Using scripts or tools available in your preferred load test application, QoS metrics can be applied to the report upon the completion of the load test. Another advantage of QoS metrics is that they can be reused. For instance, a metric that checks the load test report for SOAP Fault errors, the average CPU utilization of the server, or the average response time of a Web service can be reused in many load tests. A section of a sample load test report that utilizes QoS metrics is shown in Figure 2.





<u>Test</u>	Reports	QoS		
		Metric Status	Status Details	Graphs
CheckReservation	History	CPU Average	CPU Average = 55 %	
	HTML	HPS Average	HitPerSecond Average = 126	
	Binary			
PlaceReservation	History	CPU Average	CPU utilization too high. CPU Average = 92 %	
	HTML	HPS Average	HitPerSecond Average = 36	
	Binary			

Fig 2. Quality of service metrics determine test success (green) or failure (red).

Collecting historical data

Load test report analysis automation creates a foundation for historical analysis of performance reports. Historical analysis can reveal subtle changes that might be unnoticeable in daily reports and provides insight into the application's performance tendencies. As new functionality is introduced every day, the change in performance may be small from one day to the next, but build up to significant differences over a long period of time. Some performance degradations may not be big enough to trigger a QoS metric to fail, but can be revealed in performance history reports. Figure 3 shows an example of a QoS metric performance history report.

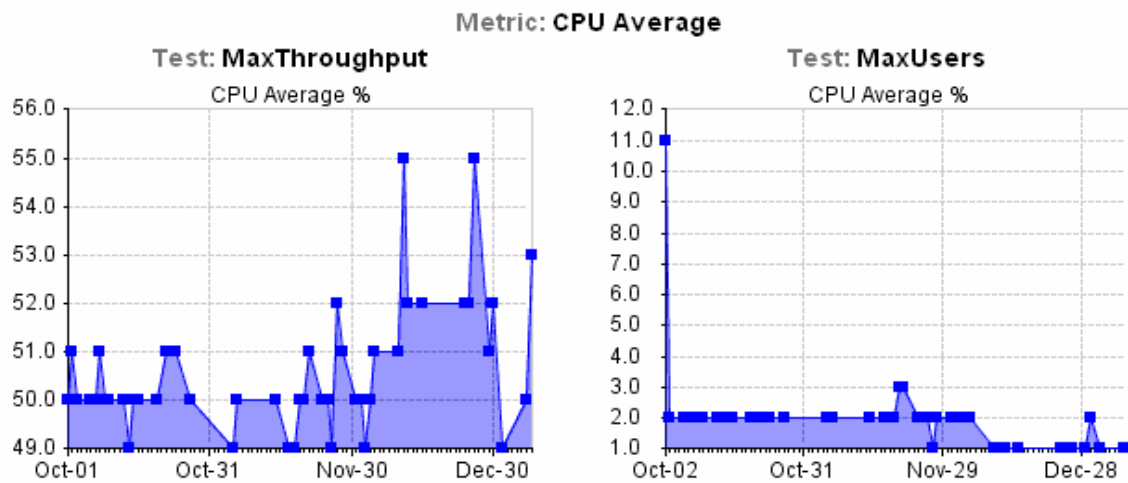


Fig 3. Evaluating performance history.

Once you have established an automated testing infrastructure, it is time to start creating load test scenarios that will evaluate the performance of your system.

Creating performance test scenarios

Performance test scenarios should be created in step with the development of the application functionality to ensure that the application's performance profile is continuously evaluated as new features are added. To satisfy this requirement, the QA team should work in close coordination with the development team over the entire application life cycle. Alternatively, the development team can be made responsible for performance test automation of its own code. The practice of creating a unit test or other appropriate functional test for every feature or bug fix is becoming more and more common in forward-looking software development organizations. The same practice could be successfully applied to performance tests as well.

The best way to build performance tests is to reuse the functional application tests in the load test scenarios. With this approach, the virtual users of the load testing application run complete functional test suites or parts of functional test suites based on the virtual user profile role. When creating load test scenarios from functional tests, make sure that the virtual users running functional tests do not share resources that they would not share in the real world (such as TCP Sockets, SSL connections, HTTP sessions, SAML tokens etc.).

Following either the traditional or the test early, test often performance testing approach usually results in the creation of a small number of performance tests that are designed to test as much as possible in as few load test runs as possible. Why? The tests are run and analyzed manually, and the fewer load tests there are, the more manageable the testing solution is. The downside of this approach is that load test scenarios which try to test everything in a single run usually generate results that are hard to analyze.

If performance testing is automated, the situation is different: you can create a greater number of tests without the risk of making the entire performance testing solution unmanageable. You can take advantage of this in two ways:

- Extend high-level Web services performance tests with sub-system or even component tests to help isolate performance problems and improve the diagnostic ability of the tests.
- Increase the number of tests to improve performance test coverage.

Improving performance tests' diagnostic ability

As a rule, more generic tests have greater coverage. However, they are also less adept at identifying the specific place in the system that is responsible for a performance problem. Metaphorically speaking, such tests have greater breadth, but less depth. More isolated tests, on the other hand, provide less coverage, but are better at pointing to the exact location of a problem in the system internals. In other words, because they concentrate on a specific part of the system, they have greater depth but less breadth. An effective set of performance tests would contain both generic high-level (breadth) tests and specific low-level (depth) tests that complement each other in improving the overall diagnostic potential of a performance test batch.

For instance, a high-level performance test that invokes a Web service via its HTTP access point might reveal that the service is responding too slowly. A more isolated performance test on an EJB component or an SQL query that is being invoked as a result of the Web service call would more precisely identify the part of the application stack that is slowing down the service. With the automated performance testing system in place, you can easily increase the number of tests and augment the high-level tests that invoke your Web services via their access points with more isolated, low-level tests that target the performance of the underlying tiers, components, sub-systems, internal Web services, or other resources your application might depend on.

In practice, you don't have to create low-level isolated tests for all components and all tiers to complement the high-level tests. Depending on the available time and resources, you can limit yourself to the most important ones and build up isolated performance tests as problems arise. For example: while investigating a high-level Web service test failure, a performance problem is discovered in an SQL query. Once the problem is resolved in the source code, secure this fix by adding an SQL query performance test that checks for the regression you just fixed. This way, your performance test library will grow "organically" in response to the arising needs.

Increasing performance tests' coverage

The usefulness of the performance tests is directly related to how closely they emulate request streams that the Web services application will encounter once it is deployed in the production environment. In a complex Web services environment, it is of the essence to choose a systematic approach in order to achieve adequate performance test coverage. Such an approach should include a wide range of use case scenarios that your application may encounter.

One such approach is to develop load test categories that can describe various sides of the expected stream of requests. Such categories can describe request types, sequences, and intensities with varying degrees of accuracy. An example of such a category breakdown is shown in Figure 4.

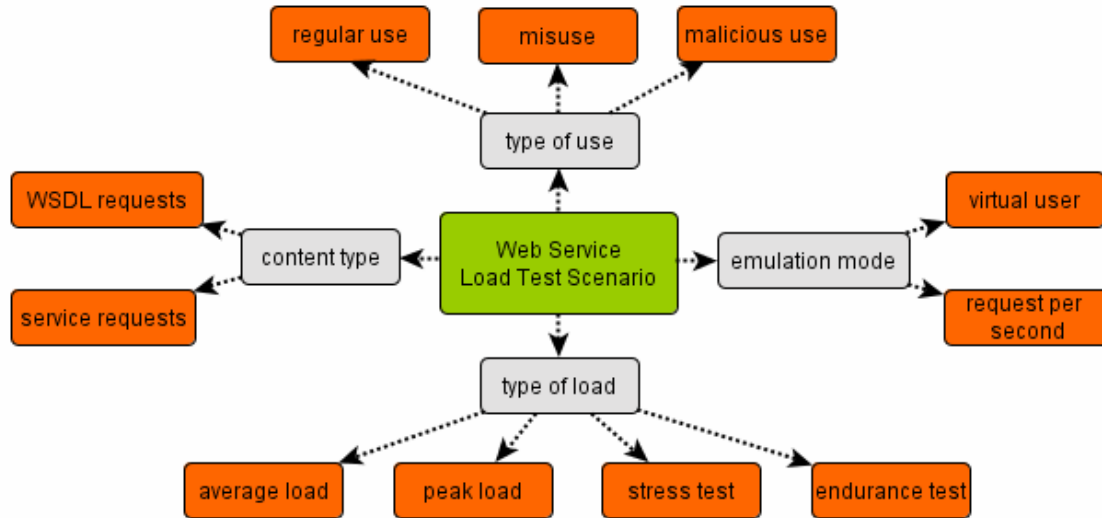


Fig 4. Load Test scenario categories.

Let's consider these categories in more detail. (The load type category analysis of your Web service can obviously include other categories as well as extend the ones shown in the Figure 4).

Type of use

Depending on the type of deployment, your Web services can be exposed to various types of SOAP clients. These clients may produce unexpected, erroneous, and even malicious requests. Your load test scenarios should include profiles that emulate such users. The more your Web service is exposed to the outside world (as opposed to being for internal consumption), the greater the probability of non-regular usage. The misuse and malicious use categories may include invalid SOAP requests as well as valid requests with unusual or unexpected values of request sizes. For example, if your service uses an array of complex types, examine your WSDL and create load test scenarios that emulate requests with expected, average, and maximum possible element counts, as well as element counts that exceed the allowed maximum.

```

<xsd:complexType name="IntArray">
  <xsd:sequence>
    <xsd:element name="arg" type="xsd:int" maxOccurs="100"/>
  </xsd:sequence>
</xsd:complexType>
  
```

Measure service performance with various sizes of client requests and server responses. If the expected request sizes and their probabilities are known (for example, based on log analysis), then create the request mix accordingly. If such data is unavailable, test with the best-, average-, and worst-case scenarios to cover the full performance spectrum.

Emulation mode

A Web service may or may not support the notion of a user. More generically, it may be stateful or stateless. Your decision to use either virtual user or request per second emulation mode should be based on this criteria. For example, the load of a stateless search engine exposed as a Web service is best expressed in terms of a number of requests per second because the notion of a virtual user is not well-defined in this case. A counter example of a stateful Web service is one that supports customer login, such as a ticket reservation service. In this context, it makes more sense to use virtual user emulation mode. If your service is stateless and you have chosen the request per second approach, make sure that you select a test tool which supports this mode. If a load test tool can sustain only the scheduled number of users, the effective request injection rate may vary substantially based on server response times. Such a tool will not be able to accurately emulate the desired request sequence. If the number of users is constant, the request injection rate will be inversely proportionate to the server processing time. It will also be likely to fluctuate, sometimes dramatically, during the test.

When load testing stateful Web services, such as services that support the notion of a user, make sure that you are applying appropriate intensity and concurrency loads. Load intensity can be expressed in request arrival rate; it affects system resources required to transfer and process client requests, such as CPU and network resources. Load concurrency, on the other hand, affects system resources required to keep the data associated with logged-in users or other stateful entities such as session object in memory, open connections, or used disk space. A concurrent load of appropriate intensity could expose synchronization errors in your Web service application. You can control the ratio between load intensity and concurrency by changing the virtual user think time in your load test tool.

Content type

When load testing Web services, it is easy to overlook the fact that SOAP clients may periodically refresh the WSDL, which describes the service, to get updates of the service parameters it is about to invoke. The probability of such updates may vary depending on the circumstances. Some SOAP clients refresh the WSDL every time they make a call. The test team can analyze access logs or make reasonable predictions based on the nature of the service.

If the WSDL access factor (the probability of WSDL access per service invocation) is high and WSDL size is compatible with the combined average size of request and response, then network utilization will be noticeably higher in this scenario, as compared to the one without the WSDL refresh. If your Web services WSDLs are generated dynamically, the high WSDL access factor will affect server utilization as well. On the other hand, if your WSDLs are static, you can offload your application server by moving the WSDL files to a separate Web server optimized for serving static pages. Such a move will create increased capacity for processing Web service requests.

Type of load

To ensure that your Web services application can handle the challenges it will face once it is deployed in production, you test its performance with various load intensities and durations. Performance requirement specifications should include metrics for both expected average and peak loads. After you run average and peak load scenarios, conduct a stress test. A stress test should reveal the Web services application's behavior under extreme circumstances, which would cause your

application to start running out of resources such as database connections or disk space. Your application should not crash under this stress.

It is important to keep in mind that simply pushing the load generator throttle to the floor is not enough to thoroughly stress test a Web services application. Be explicit in what part of the system you are stressing. While some parts of the system may be running out of resources, others may be comfortably underutilized. Ask yourself: When this application is deployed in the production environment, will the resource utilization profile be the same? Can you be sure that the parts of the system which were not stressed during the test will not experience resource starvation in the production environment?

For instance, assume that performance tests on the staging environment revealed that the application bottleneck was the CPU of the database server. However, you know that you have a high performance database server cluster in production. In this case, it is likely that the production system bottleneck will be somewhere else and the system will respond differently under stress. In such a situation, it would make sense to change the parts of your database access code with code stubs that emulate access to the database. The system bottleneck will shift to some other resource, and the test will better emulate production system behavior.

Applying this code stubbing approach to other parts of the system (as described above) will allow you to shift bottlenecks to the parts of the system that you want to put under stress and thus more thoroughly test your application. Keeping a table of system behavior under stress, as shown in Figure 5, will help you approach stress testing in a more systematic manner.

Test	Resource under stress	Max. resource utilization under stress	System behavior under stress	System behavior after stress load is removed
1.	Application Server CPU	98%	Response time increased to 3 sec. on average. Timeouts in 10% of requests.	Returned to normal performance - success
2.	App. Server thread pool	100% - all threads busy	Request timeouts followed by OutOfMemoryError(s) printed in sever console.	Up to 50% errors after stress load is removed - failure
3.	App. Server Network connections	100% - running out of sockets	Connection refused in 40% of requests.	Returned to normal performance - success
...

Fig 5. System stress test evaluation.

Performance degradation—even dramatic degradation—is acceptable in this context, but the application should return to normal after the load has been reduced to average. If the application does not crash under stress, verify that the resources utilized during the stress have been released. A comprehensive performance testing plan will also include an endurance test that verifies the application's ability to run for hours or days, and could reveal slow resource leaks that are not noticeable during regular tests. Slow memory leaks are among the most common. If they are present in a Java environment, these leaks could lead to a java.lang.OutOfMemoryError and the crash of the application server instance.

Creating a real-world value mix

To better verify the robustness of your Web service, use your load test tool to generate a wide variety of values inside SOAP requests. This mix can be achieved, for example, by using multiple value data sources (such as spreadsheets or databases), or by having the values of the desired range dynamically-generated (scripted) and then passed to virtual users that simulate SOAP clients. By using this approach in load tests of sufficient duration and intensity, you can test your Web service with an extended range and mix of argument values that will augment your functional testing.

Depending on the circumstances, it may be advisable to run the mixed request load test after all known concurrency issues have been resolved. If errors start occurring after the variable request mix has been introduced, inspect error details and create functional tests using the values that caused your Web service to fail during load testing. These newly-created functional tests should become part of your functional test suite.

Wrap up

By implementing an automated performance testing process in your software development organization, you can reduce the number and severity of potential performance problems in your Web services application and improve its overall quality.

The strategies discussed in this guide can be applied and automated using Parasoft Load Test, which is available with Parasoft SOAtest in the [Parasoft Functional and Load Test solution](#). Load Test not only monitors the server's response rate with the specified number and mixture of simultaneous requests, but also verifies whether the test loads cause functionality problems.

Load Test allows you to load test SOAtest end-to-end functional test suites (which may span from the web interface, through services, to the database) according to preset or customized load test scenarios. Preset scenarios can be used to verify robustness, scalability and durability. You can easily customize these scenarios to use different test cases, load levels, load distributions, and so on. You can also distribute virtual users across remote server machines to simulate extreme loads and/or test from different locations.

Support is also provided for load testing non-Parasoft components such as JUnits or lightweight socket-based components. This provides teams an integrated solution for their various load testing needs.

About Parasoft

For 21 years, Parasoft has investigated how and why software defects are introduced into applications. Our solutions leverage this research to dramatically improve SDLC productivity and application quality. Through an optimal combination of quality tools, configurable workflow, and automated infrastructure, Parasoft seamlessly integrates into your development environment to drive SDLC tasks to a predictable outcome. Whether you are delivering code, evolving and integrating business systems, or improving business processes—draw on our expertise and award-winning products to ensure that quality software can be delivered consistently and efficiently. For more information, visit <http://www.parasoft.com>.

About Parasoft SOAtest and Load Test

Parasoft SOAtest automates web application testing, message/protocol testing, cloud testing, security testing, and behavior virtualization. Parasoft SOAtest and Parasoft Load Test (packaged together) ensure secure, reliable, compliant business processes and seamlessly integrate with Parasoft language products (e.g., Parasoft Jtest) to help teams prevent and detect application-layer defects from the start of the SDLC.

Parasoft SOAtest provides is an integrated solution for:

- End-to-end testing: To continuously validate all critical aspects of complex transactions, which may extend beyond the message layer through a web interface, ESBs, databases, and everything in between.
- Environment management: To reduce the complexity of testing in today's heterogeneous environments—with limited visibility/control of distributed components or vendor-specific technologies.
- Quality governance: To continuously measure how each service conforms to the often dynamic expectations defined by both your own organization and your partners.
- Process visibility and control: To establish a sustainable workflow that helps the entire team efficiently develop, share, and manage the evolution of quality assets throughout the lifecycle.

Parasoft Load Test allows you to load test your SOAtest tests to verify functionality and performance under load. Support is also provided for load testing non-Parasoft components such as JUnits or lightweight socket-based components, and for detecting concurrency issues.

Parasoft's customers, including 58% of the Fortune 500, rely on SOAtest and Load Test for:

- Ensuring the reliability, security, and compliance of SOA and web applications
- Reducing the time and effort required to construct and maintain automated tests
- Automatically and continuously validating complex business scenarios
- Facilitating testing in incomplete and/or evolving environments
- Validating performance and functionality expectations under load
- Rapidly diagnosing problems directly from the test environment



Contacting Parasoft

USA

101 E. Huntington Drive, 2nd Floor
Monrovia, CA 91016
Toll Free: (888) 305-0041
Tel: (626) 305-0041
Fax: (626) 305-3036
Email: info@parasoft.com
URL: www.parasoft.com

Europe

France: Tel: +33 (1) 64 89 26 00
UK: Tel: + 44 (0)208 263 6005
Germany: Tel: +49 731 880309-0
Email: info-europe@parasoft.com

Asia

Tel: +886 2 6636-8090
Email: info-psa@parasoft.com

Other Locations

See <http://www.parasoft.com/contacts>

© 2010 Parasoft Corporation

All rights reserved. Parasoft and all Parasoft products and services listed within are trademarks or registered trademarks of Parasoft Corporation. All other products, services, and companies are trademarks, registered trademarks, or servicemarks of their respective holders in the US and/or other countries.