SPRING IN PRODUCTION

by Adrian Colyer, CTO, SpringSource // October 2007



The Spring Framework and Spring Portfolio products are used in production in mission-critical applications all over the world and in every kind of industry.

The Spring programming model and configuration model is well understood and documented and used by hundreds of thousands of developers worldwide.

This white paper was written for operations teams who manage those applications. It explains Spring from a different perspective: the Spring runtime and the critical role that it plays in the execution of enterprise applications.

Copyright 2007, SpringSource. Copying, publishing, or distributing without express written permission is prohibited.

Introduction

The Spring Framework and Spring Portfolio products are used in production in mission-critical applications all over the world and in every kind of industry.

For example, almost all of the inter-bank transfers in the United Kingdom are handled by a company called Voca. They process over 80 million items on a peak day, handle bill payments for over 70% of the UK population, and handle salary payments for over 90%. Voca are proud of the fact that they have never lost a single payment. These transfers are powered by Spring.

For example, HSBC recently took out a support contract with SpringSource to support their mission-critical use of Spring in production applications.

For example, 25% of the world's cargo movement through port and intermodal facilities is managed through a Spring-powered application built by Navis.

The Spring programming and configuration models are well understood and documented and used by hundreds of thousands of developers worldwide. This white paper was written for operations teams who manage those applications. It explains Spring from a different perspective: the Spring runtime and the critical role that it plays in the execution of enterprise applications.

The Spring runtime is extensive and covers a broad range of application services. In this white paper we will focus on the core runtime components used by the vast majority of Spring-powered applications, and how to tune a Springpowered application.

Spring's runtime has a deserved reputation for being of high quality and is battle-tested in production. For the benefits of operations teams supporting Spring-powered applications, I describe the the kinds of symptoms you would expect to see if there were to be a failure in one of the runtime components. We hope you never encounter such a situation!

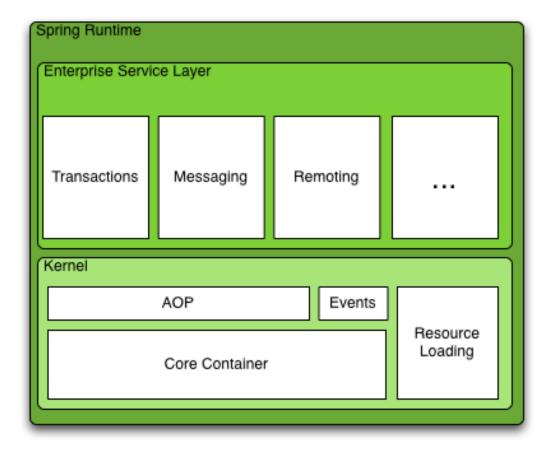
Spring Runtime Overview

The Spring runtime plays a vital role in the execution of production applications. At its core, the Spring runtime is responsible for the creation and management of the application components themselves, and for managing the execution of every single request serviced by those components. On top of this foundation the Spring runtime manages critical enterprise services such as access to resources, transactions, security, messaging, scheduling and more.

At the heart of the Spring runtime is the core Spring container. On top of this core the AOP (aspect-oriented programming) and resource loading runtime components complete the kernel of the Spring runtime. The kernel instantiates and configures application components and manages the invocation pipeline for every request serviced by those components.

Plugged into the kernel, via Spring extension mechanisms such as namespace handlers and post-processors, are the runtime components that provide the vital services that enterprise applications depend on. These components form the enterprise service layer. They ensure for example that business operations occur within transactions, manage exceptions, dispatch inbound messages to application components, establish security contexts, and so forth.

Figure 1. Spring Runtime Overview



Spring Runtime: Kernel

The Spring runtime kernel consists of the core container, which plays the fundamental role of managing application components; the AOP runtime, which manages the invocation of component operations; the resource loading manager; and an eventing subsystem.

MANAGING APPLICATION COMPONENTS

The kernel needs to be bootstrapped when an enterprise application is started. Once up and running, the main responsibilities of the kernel are to:

- determine the application components that are needed for the application
- establish the configuration information for those components
- analyze the component dependency graph and instantiate the components
- assemble the components into a coherent whole by connecting them together
- decorate the components to set up the required request dispatching pipeline
- manage the lifecycle and scope of the application components themselves.

These responsibilities are discussed further below.

Bootstrapping

A Spring *application context* provides the runtime context for an application. Before any application components can be created, an application context must be created. In some applications the application construct is explicitly created through user code. For the vast majority of Spring-powered applications the application context is created automatically through one of Spring's bootstrapping mechanisms. For example, in a web application Spring's DispatcherServlet provides the entry point. In an OSGi[™]-based application, Spring's *extender bundle* provides the same function.

When the application context is created it adapts itself to the runtime environment by installing a suitable resource loading implementation and by discovering and registering any Spring namespace handlers. Namespace handlers enable the core container to understand Spring configuration written using XML schema, and to translate the elements and attributes in the configuration blueprints into runtime components that satisfy the declared application requirements.

Any failure during the bootstrapping phase will render the application inoperable. A failure to detect and register a namespace handler means that if elements from that namespace are used in the application configuration then the creation of the application context will fail.

Determining component configuration

Once bootstrapped, the next critical tasks are to determine the set of application components that need to be created; how those components are to be configured; and what supporting services they need.

The configuration information can come from many different sources. These sources are identified during the bootstrapping phase, and can be supplemented by additional configuration metadata sources specified in configuration files themselves. For example, configuration information is often sourced from multiple XML files. One of these configuration files may enable the component scanning facility introduced in Spring 2.5 which sources additional configuration metadata by scanning resources on the classpath.

The registered namespace handlers and other metadata handlers process the configuration sources to determine the set of application and infrastructure components to be instantiated. They produce a set of component definition objects that represent the desired application configuration. In addition to information on the required components, this runtime representation includes information on the component dependencies and configuration values.

Services from the enterprise services layer (and user application configuration) may have registered one or more post processors that iterate over the component definitions to augment them.

This crucial phase of application startup ensures that the application blueprint (the required components and their configuration) is successfully created in memory, ready for realization as fully functioning, configured, application objects in the next phase.

Failures during this blueprint discovery phase can lead to missing or misconfigured components. It is imperative that the application blueprint specified by the application designers be faithfully captured at this stage.

Instantiation, Configuration, and Assembly

The next key runtime responsibility of the kernel is to analyze the blueprint and begin instantiating the needed components. The container must take into account dependencies between components and any specified ordering constraints so that components can be created in a 'safe' order.

The configuration blueprint specifies how the component instances themselves are to be obtained. The construction process can be complex – including for example the need to look up objects in a directory or service registry, create objects via factories, or create references to remote services. The new components are configured using configuration data that can come from a variety of sources, including XML, properties files, evaluation of placeholder expressions and so on. The container provides sophisticated support for converting string-based data into the types required by the components, and for generating rich sets of configuration data using structures such as lists, maps and sets. A component is also configured with its collaborators (other application components that it needs to interact with). In the simplest case the collaborators are identified by name, but the container may also need to automatically deduce collaborators using a variety of strategies such as looking for matches by name and by type.

Failures during context creation and configuration prevent the application context from initializing successfully, and hence leave the application inoperable. An "Exception during initialization" message will appear in the log file in this case.

Component Post Processing

Services from the enterprise service layer may have registered one or more component (bean) post processors with the container. It is also possible for additional user-level post processors to be declared in configuration files. The runtime orders these post processors and then invokes them after creating the component instances.

Many of the post processors registered by the enterprise service layer decorate the component as described in the following section.

Decoration

Spring presents a simple declarative model for specifying behaviors that crosscut component application logic; behaviors specified for example by elements in the transaction namespace, or declared as aspects using the AOP namespace or annotated classes. The runtime machinery that translates these declarations into runtime artifacts implementing the semantics of those declarations is much more complex.

The basic approach is first to qualify component operations to see if any crosscutting logic could potentially need to be executed whenever the component operation in question is invoked. For example, is it required to demarcate transactions? There are a variety of matching strategies, but the most common is to evaluate an AspectJ pointcut expression. This involves parsing the expression and matching the terms against the signature of the operation in question. After the qualifying phase, if it has been determined that one or more of the component operations need to be advised by one or more aspects¹, then Spring generates a runtime proxy that ensures the advice is invoked at the appropriate time when the operation is invoked (before, after, around etc.). It is possible that multiple pieces of advice need to run for the same operation, and the kernel must obey ordering constraints here. Proxy generation is done using either JDK dynamic proxies or the runtime generation of subclasses. The container can also be configured to integrate with AspectJ load-time weaving, in which case on-the-fly byte code instrumentation of the application classes occurs in order to provide a very efficient implementation of the required behavior.

In some cases it is not possible to determine statically whether the execution of a given component operation should or should not be advised. in this case, the kernel determines the minimal runtime test needed each time the operation is invoked (this could be, for example, testing whether an argument is an instance of a given type).

At the end of the decoration phase, any components that require crosscutting services have been proxied, and the proxy objects themselves are registered in place of the application objects.

The decoration phase is the foundation for critical enterprise services such as transaction and security management.

¹ or Spring "advisors"

Managing component life cycle and scope

The container's job is not over once the initial set of components have been instantiated, configured, and decorated. Components may have different life cycles which must be managed – for example a new instance of a requestscoped component must be instantiated for every request in which the component is accessed. For any non-singleton component in the container, the container is also responsible for destroying the component instance(s) at the end of their life.

REQUEST DISPATCHING

The kernel is intimately involved in the execution pipeline for every decorated or scoped component. Whenever an operation is invoked on such a component the Spring runtime first invokes any interceptors that were configured by the AOP subsystem during the decoration phase. Execution of some of the interceptors may be conditional depending on runtime tests determined earlier. These tests are executed and the interceptors are only invoked if they pass. Spring implements the AspectJ pointcut and advice model with fully typed advice. This means that the runtime needs to extract contextual information from the request and bind it to parameters in the advice method dispatch.

If the advice execution indicates that computation should proceed then the runtime selects the target component instance. In simple cases the target component instance is always the same, but for scoped beans such as a request scoped bean, the runtime must select the instance being managed in the current scope.

After executing the component operation the AOP runtime also needs to invoke the interceptor chain before returning to the caller. This can again involve executing runtime tests and parameter binding. Interceptor execution may for example be conditional on return values or exceptions thrown.

The decoration phase ensures that the AOP subsystem is correctly configured, but it is the runtime management of the invocation of operations on components that ensures that enterprise services are correctly applied each and every time.

RESOURCE LOADING

The resource loading component is a relatively small but still important part of

the kernel. It is responsible for loading resources wherever the container needs them. It can support resource loading from the class path, file system, input streams, byte arrays, and so on.

FVFNTS

The kernel contains an event subsystem that supports the publication of application context events, and the ability for an application component to listen to those events. By default event delivery is synchronous and happens within the transaction context of the publisher. It is possible to customize this behavior. This event system is used by Spring Security for example, to generate security events. If using Spring in a clustered environment, many vendors in the clustering and caching space provide support for distributing application context events across a cluster.

Spring Runtime: Enterprise Service Layer

The enterprise services support provided by the Spring runtime builds on top of the kernel. Spring supports a wide range of enterprise services, here we will focus on some of the most widely used components.

TRANSACTION MANAGEMENT

Runtime transaction management support consists of two phases. During setup, the transaction subsystem interprets @Transactional annotations and elements in the "tx" namespace to decorate application components that require transaction support. In the execution phase the transaction interceptor registered during set-up is invoked before and after the execution of every transactional operation. The interceptor ensures that transactions are begun, committed, suspended, resumed, and rolled-back according to the transaction propagation attribute specified in the transaction metadata. The platform transaction manager ensures that isolation levels and read-only behavior are passed on to the underlying transaction manager.

When an exception is thrown from a transactional method, the transaction subsystem interprets rollback rules based on the exception type to decide

whether or not to roll-back the transaction.

Clearly, any failure in the transaction subsystem can have serious consequences leading to non-atomic updates, dirty reads, and other undesirable effects.

DATA ACCESS

Spring's runtime support for data access consists of multiple independent subsystems, depending on the data access technology used (e.g. JDBC, JPA, iBATIS, Hibernate). It also supports mixed-mode data access, managing for example mixed usage of JDBC and Hibernate in the same transaction. The Spring runtime manages persistence sessions, and is involved in the execution of every data access operation to ensure that exceptions are translated into Spring's canonical DataAccessException hierarchy as required.

For JDBC based data access, the Spring runtime is intimately involved with data operations – managing the iteration over result sets, coordinating the translation of response data into domain objects, invoking stored procedures and so on.

Data access, and the integration between data access and the transaction management subsystem is a critical part of virtually all enterprise applications.

MESSAGING

The central runtime component in Spring's messaging support are the message listener containers. These act as the intermediaries between message-driven application components and messaging providers. The containers are responsible for registering with the message provider to receive messages, managing threads for message processing, and dispatching messages to application component operations. The messaging runtime is also responsible for transaction participation, resource acquisition and release, and exception translation.

Failures in this subsystem may result in missing or incorrectly delivered messages, loss of task execution threads, or non-atomic message processing.

SECURITY MANAGEMENT

When using Spring Security to provide security management the security runtime is once again involved in the processing pipeline of many application components. Filters in the web layer drive authentication and authorization. The runtime is also responsible for security management in other application layers. It registers interceptors during the decoration phase and then intercepts the invocation of any "secured" operation. The runtime is responsible for managing role-based access, and also for ACL-based security. Pre-invocation interceptors validate the right of the caller to invoke a given operation, postinvocation interceptors check that the caller can see the object(s) returned. Collection filtering may be performed to remove objects the caller is not intended to see.

Failures in the security subsystem may manifest as unprotected access to secured resources and operations, denied permissions when they should be granted, and authentication failures.

JMX INTEGRATION

Spring makes it very easy to expose any MBean, or even any application component, to JMXTM. The JMX subsystem can register any existing MBean or MXBean with an MBeanServer. It also makes it simple to register application components that do not implement any management interfaces at all. The JMX subsystem will build ModelMBeans on the fly for such application components. Making good use of Spring's JMX support to expose application information through JMX is an excellent way of providing operational visibility into the workings of an application.

WEB REQUESTS

The involvement of the Spring runtime in processing web requests depends on the web framework used. Spring always manages request and session-scoped components – ensuring that when an an operation is invoked on such a component, the correct component instance is instantiated (if needed) and the request dispatched to it. The Spring runtime is often involved in data binding (moving data from form submissions into object fields) and validation. Many modern web applications are built on the Spring Web Flow engine. This has a significant runtime component that manages the state of flows and drives transitions between flow states.

Failures in the web runtime may result in accessing the wrong or stale state, lost state, a failure to successfully translate form input into application domain values, and similar scenarios.

Spring Runtime: Tuning

How can you tune a Spring-powered application to improve performance?

Before making any changes to a Spring-powered application, the first thing to do is to measure to find out where the hot-spots are and to ensure that you can quantify the benefits of any proposed change. Optimizations then fall into two major categories: establishing an effective blueprint (tuning your configuration), and making effective use of runtime facilities (optimizing your application design). Start off with the cleanest and clearest design, making full use of the facilities that Spring offers, and only deviate from this where the numbers show real benefit.

MEASURE FIRST

The starting point for any tuning exercise is to measure. Tools such as Apache JMeter [1], Selenium [3], and a profiler are useful here. SpringSource consultants have had good success combining JAMon [2] with Spring AOP or Aspect to profile component operations and request processing paths.

Measure by layer so that you know for example how much time is spent in and below the application layer versus the amount of time you spend rendering in the web laver.

ESTABLISH AN EFFECTIVE BLUEPRINT

The secret to establishing an effective blueprint is to take full advantage of your deployment platform. Because Spring keeps environmental dependencies out of your application code this becomes much easier to do.

For database connection pools, if you are running on an application server then use a pool configured through the administration console and configure Spring to look the reference up via JNDI. For example, use

```
<jee:jndi-lookup id="dataSource"
    jndi-name="jdbc/MyDataSource"/>
```

Instead of

<bean id="dataSource"</pre>

```
class="com.mchange.v2.c3p0.ComboPooledDataSource"
 destroy-method="close">
 cproperty name="jdbcUrl" value="${jdbc.url}"/>
 cproperty name="password" value="${jdbc.password}"/>
</bean>
```

This gives you two advantages: one is that it becomes easier for the operations team to administer the application through the application server console, and the second is that the application server vendor will have optimized the connection pool for use on their platform.

In the same vein, use JNDI to obtain an application-server configured JMS ConnectionFactory and Destinations when working with JMS. Additionally, the application-server ConnectionFactory not only will pool JMS connections but also other intermediate JMS resources such as sessions and message producers. By pooling these other JMS resources you can come closer to the maximum throughput of the JMS provider. In a non-managed JavaTM EE environment using Spring's SingleConnectionFactory will reuse the same connection instance. If your vendor provides a pooling ConnectionFactory adapter then take advantage of it. This will remove the largest performance bottleneck when sending messages with JmsTemplate.

For transaction management, if Spring has a custom platform transaction manager for your deployment platform then be sure to use it. For example, Spring provides the WebLogicJtaTransactionManager that takes full advantage of a WebLogic[™]-managed transaction environment, the WebSphereUowTransactionManager for WebSphere[™], and the Oc4jJtaTransactionManager for OC4J, these should all be used in preference to the standard JtaTransactionManager when deploying to one of these platforms. The <tx:jta-transaction-manager/> tag introduced in Spring 2.5 will automatically detect the underlying platform and select the appropriate implementation.

When exposing application resources via Spring's JMX support, you'll want to integrate with the MBeanServer provided by your production platform. For WebLogic look up the MBeanServer from JNDI ("java:comp/env/jmx/runtime"), and for WebSphere use the WebSphereMBeanServerFactoryBean. For other environments the default discovery mechanisms work just fine. In Spring 2.5,

the <context:mbean-server ... /> tag will automatically detect the appropriate MBeanServer for your platform.

Using server-configured resources looked up via JNDI enables you to take full advantage of your deployment platform facilities. However, you still want to be able to run integration tests outside of the application server. For example, you may want to run some basic messaging integration tests using ActiveMQ, but deploy in production using IBM[™] MQSeries[™]. Such configurations are easy to achieve by taking advantage of Spring's support for multiple configuration files. We recommend separating out all environment-dependent configuration from the core application configuration. A best practice is to maintain a configuration file per application module. In addition to these you might define for example an integration-test.xml configuration and a production.xml configuration. When constructing integration tests you simply build an application context using the module configuration files for the modules under test in conjunction with the integration-test.xml file, and for production and system testing replace this with the production.xml file.

On a related note, Spring's PropertyPlaceholderConfigurer is excellent for externalizing configuration settings that may need to be changed by an operations team. One convention that SpringSource consultants have successfully used is to chain properties files as follows:

- 1. classpath*:*.properties.local: These property files are not checked into the source code control system, and allow for per-developer overrides.
- 2. classpath*:META-INF/*.properties.default: These property files are included in the application artifacts produced by the build and contain default configuration settings. Sometimes this level may be omitted, depending on the project requirements.
- 3. classpath*:*.properties: These property files are outside of any application artifact and used to allow easy modification by an operations team.

```
<context:property-placeholder</pre>
   location="classpath*:META-INF/*.properties.default,
             classpath*:*.properties,
             classpath*:*.properties.local"/>
```

A good tip here is to use Spring's JMX export capabilities to define an MBean that exposes all of the configuration values via JMX. This enables you to connect to a running application and easily see the configuration values it is currently using.

Here are some additional things to think about when creating an effective blueprint:

- Experiment to find the best number of connections for your JDBC connection pool. Make sure to use realistic production scenarios when testing this.
- When using Spring's TaskExecutor abstraction with a thread pool implementation, choose an appropriate thread pool size. For computation-bound tasks a good starting point is to set the number of threads equal to your number of CPUs. Add one thread if your tasks use local file i/o. Add "several" threads for network-based i/o. When using the TaskExecutor abstraction on WebSphere or WebLogic, be sure to use CommonJ WorkManagerTaskExecutor implementation to take advantage of the underlying platform optimizations (for example, self-tuning pools) and integration.
- Use the read-only attribute for read-only transactions when configuring transaction demarcation. This can give a real performance boost to transactions using Hibernate that read in a lot of objects from the database but don't change anything. It causes the Hibernate session to be set to FlushMode.NEVER, which means that Hibernate will no longer iterate over all of the objects in the session checking for dirtiness when the session is flushed.
- Consider using a local transaction manager instead of JTA if you don't need two-phase commit (have only one resource manager involved). Spring can easily mix and match JDBC and Hibernate data access in the same transaction using a HibernateTransactionManager :- there may be two different access modes, but there is only one resource manager involved in the transaction.
- Consider using native JMS transactions for message listener containers by setting 'acknowledge="transacted"'.

TAKE ADVANTAGE OF RUNTIME OPTIMIZATIONS

Many enterprise application performance problems can be tracked down to the

persistence layer. Good performance here is often a function of sound design choices. Some useful design tips include:

- When using an ORM tool, strive for the right balance between eager and lazy loading strategies. By default use lazy loading, and then use fetch-joins to tune specific cases that will benefit from eager loading. When tuning queries use a data set that matches your "1 year from now in production" projections.
- Use the facilities provided by your ORM tool or database to show the SQL statements in the logs. This enables you to very easily detect when too many queries are being issued.
- When working with Hibernate, make use of the Hibernate Statistics object to understand what is happening at runtime. You can either access the statistics programmatically, or use Spring to export the Hibernate Statistics MBean to your MBean server. You can combine programmatic use of the statistics object with JUnit tests to assert how many queries you expect to be issued, or specify a tolerance for the number of queries allowed.
- For batch style operations, bulk updates, or inserts, and stored procedures it's normally best to use JDBC (via Spring JDBC) than an ORM tool. Spring makes it easy to mix and match, for example, Hibernate and JDBC data access in the same transaction – just be sure to remember to flush the Hibernate session before working with JDBC directly if working on the same tables.
- Make the most of the features that your database offers.
 - In an application that had to read a (very large) Excel spreadsheet, perform a simple transformation, and then insert rows into a SQL Server table, the operation was taking three hours, despite best attempts at tuning. By taking advantage of SQL Server linked queries to treat the spreadsheet as a database accessed via ODBC, and moving the logic into a stored procedure, the whole operation took only 17 seconds.
 - An application that needed to traverse a tree of data to an arbitrary depth was written using Hibernate. Despite best efforts at tuning the query would take too long (hours), or go out of memory. Moving to an Oracle[™] stored procedure and taking advantage of Oracle's hierarchical query support enabled the operation to easily complete in under 5 seconds.

 If you need to load a flat file into an Oracle database, use Oracle SQL Loader to load the data into a staging table, then use e.g. a stored procedure to transform and copy the data into the desired target table(s).

- If you have an operation that contains entirely persistence logic (no business logic), consider moving it into the database as a stored procedure and invoke it via Spring JDBC.
- Read-only reference data can be kept in a cache in memory

Batch applications bring additional consideration since memory usage becomes very important – you want to avoid consuming large amounts of memory and causing costly garbage collection pauses. Stream-based algorithms are the best approach here. Use iterators rather than collections for example. When working with files, if you need to split lines then use character based splitting rather than String-based. Using this approach we were able to read a 2.5 million line file, parse and process it, in under 4 seconds and using only 102K of memory.

Use streaming for batch applications working with XML too. One application we encountered needed to process 100,000 complex XML events captured in a 280 megabyte file. This took 2.5 hours to process using a DOM-based approach with one DOM tree per-event, and garbage collection pause of up to 9 minutes. Switching to an XML pull-parsing based approach enabled the whole input to be processed in 3 seconds, using only 200K of memory.

Another good tip here is to use the JVM statistics available through the java.lang.management package in unit and integration tests. This allows you to make assertions about CPU and garbage collection time and so on.

A final piece of advice relating to tuning your data layer: every team benefits from access to a good DBA.

A collection of other runtime tuning and optimization tips gathered from some of SpringSource's consultants:

- The retry support in the Spring Batch project can be used to retry failing operations (for example, an operation that has failed on an individual cluster node in an Oracle RAC). This can ease the operational burden by reducing the number of failures that bubble up to end users.
- Don't underestimate the cost of web content rendering. You definitely want to do this outside of a transaction.
- Don't instantiate an application context per-request (a mistake

sometimes encountered when teams migrate legacy applications to Spring).

- Consider exploiting Spring's asynchronous task executors to reduce user wait time for tasks that can be run in the background.
- Choose an appropriate remoting protocol. If you don't need SOAP interoperability, a simple scheme such as Spring's HttpInvoker support will be simpler and faster.
- Consider using AspectJ in place of Spring AOP for aspects that impact large portions of your application.

Some resources that SpringSource consultants have found to be helpful in tuning include:

- Thomas Kyte's "Runstats.sql" test harness [4]
- "Effective Oracle by Design" (Thomas Kyte) [6]
- "Java Performance Tuning" (Jack Shirazi) [5]
- Sun's Java Performance Guides [7]

Summary

The Spring runtime provides an extensive set of services to application components and plays a critical role in the execution of applications in production. These services include core application component management, transactions, security, data access and more. This white paper provides operations teams with a guide to the Spring runtime, to aid in understanding of the runtime behavior of Spring applications and to assist in troubleshooting. When it comes to tuning a Spring-powered application the key is first to measure, then to establish an effective blueprint, and to take advantage of runtime optimizations through effective design.

In recognition of the vital nature of the runtime services to the correct functioning (or even functioning at all) of production applications, Spring is engineered to high standards and has a strong reputation for quality. The many thousands of Spring-powered applications in production around the world are a testament to this.

ACKNOWLEDGMENTS

Thanks to my many SpringSource colleagues who contributed their tuning hints and tips for this white paper.

Java and JMX are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

OSGi is a registered trademark of the OSGi Alliance in the United States, other countries, or both.

IBM, MQSeries, and WebSphere are trademarks or registered trademarks of IBM in the United States, other countries, or both.

WebLogic is a registered trademark of BEA Systems, Inc. in the United States, other countries, or both.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates.

REFERENCES

- [1] Apache JMeter - http://jakarta.apache.org/jmeter/
- [2] JAMon - http://jamonapi.sourceforge.net/
- [3] Selenium - http://www.openga.org/selenium/
- [4] Runstats.sql (Thomas Kyte) http://asktom.oracle.com/tkyte/runstats.html
- [5] "Java Performance Tuning", Jack Shirazi
- "Effective Oracle by Design", Thomas Kyte [6]
- Java performance guides http://java.sun.com/docs/performance/ [7]