

ARCHITECTURAL IMPLICATIONS OF RIA

RETHINKING WEB-BASED APPLICATIONS

PRINTED: TUESDAY, MAY 12, 2009

© HILBERT COMPUTING, INC., 2009

ARCHITECTURAL IMPLICATIONS OF RIA

RETHINKING WEB-BASED APPLICATIONS

TABLE OF CONTENTS

INTRODUCTION.....	1
HOW WE GOT HERE.....	2
HTTP.....	2
Dealing with Statelessness.....	3
Synchronicity.....	5
Where We Are: A Redux.....	6
RICH INTERNET APPLICATIONS.....	7
Statelessness.....	7
Asynchronicity.....	8
RESTful Web Services.....	10
THE NECESSARY STATE?.....	11
THE ARCHITECTURAL RIPPLE.....	13
THE ARCHITECTURE OF CHOICE.....	14
APPENDIX A: UNMANAGED THREADS.....	15

ARCHITECTURAL IMPLICATIONS OF RIA

RETHINKING WEB-BASED APPLICATIONS

INTRODUCTION

When many developers first see a rich internet application (RIA), the reaction is typically astonishment at the eye candy that is possible with these emerging technologies. That appeal alone is enough to look seriously at a move to RIA applications. Many companies are looking at RIA to accommodate the desire for the end-customer to have a better user-interface experience.

When development begins in earnest with these RIA technologies, it becomes clear that there is something more significant than just visual appeal and a more interactive user interface. These technologies enable a shift back to a client/server architecture that is more similar to the architectures in the 1990s than the architectures embodied in the typical web application that were developed after the emergence of the world-wide web.

It's time to re-evaluate our approach to application architecture in light of these changes. In doing so, we might even kill a sacred cow or two.

HOW WE GOT HERE

There is a perception among a lot of younger IT professionals that networked applications began with the world-wide web in the mid-1990s. In fact, there was a lot of network-based applications being developed and deployed on mainframes, mini-computers and personal computers on local area networks.

Yes, we actually had network protocols that were not based on HTTP and, for the task at hand, they worked better than HTTP.

HTTP

The fundamental difference between the world-wide web and the previous networked applications is one of scale. Internally-developed applications had a finite and predictable set of customers. With the world-wide web, your audience was potentially millions across the globe. Those considerations of scale were a driving force in the protocols that were developed in support of this then-new vision of globally-networked applications. Enter HTTP.

HTTP was designed to handle a large number of stateless, short-term, synchronous transactions. Most implementations¹ accomplish this through a pool of network threads that are reused in service of an synchronous HTTP request/response cycle. As long as the response time of a transaction is relatively small, a large volume of transactions can be supported with a surprisingly small number of threads in the network thread pool.

The design of HTTP is absolutely appropriate to meet these requirements. Scale is best accomplished through statelessness and high numbers of very small transactions. Since the original goal of the world-wide web was to serve user-initiated requests for hyper-linked information, the synchronous request/response cycle was appropriate. While it is a good and appropriate design, it can be problematic for a lot of dynamic web application scenarios. Regardless of the problems with HTTP, given that our public-facing applications need to be routed through firewalls and typically HTTP and HTTPS are the only ports allowed, that is often the only networking protocol that is available.

¹ Since I spend most of my time in Java, I will use that as my basis for discussion. Other ecosystems may have a different implementation, although I suspect they are fundamentally similar.

ARCHITECTURAL IMPLICATIONS OF RIA

RETHINKING WEB-BASED APPLICATIONS

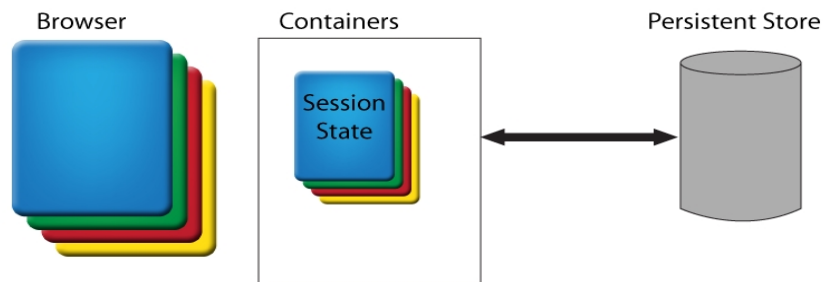
DEALING WITH STATELESSNESS

The first issue to arise when dynamic web applications were developed was the need to manage state on a stateless protocol. HTTP is a stateless network protocol whereas most prior network protocols were stateful.

An example of a prior network protocol is telnet. To create a terminal session with a remote Unix box, you would fire up your terminal program and use the `telnet` protocol to manage the life of the connection. Typically, the telnet server would spawn a thread pair or process pair - one to manage the inbound flow of data and the other to outbound flow of data. The state of the session had a one-to-one relationship with the lifecycle of the network connection. When the network connection was terminated, the session state was terminated at the same time. There wasn't the notion of "renting" a stateless thread for a brief network interaction. The network connection could last for literally for days.

Java programmers are familiar with the mostly-complete solution to the stateless nature of the HTTP protocol when writing web applications. There is a cookie sent to the browser that is a unique key to the session. That key is used to locate items that represent the transient state² of the application.

Simple Web Application



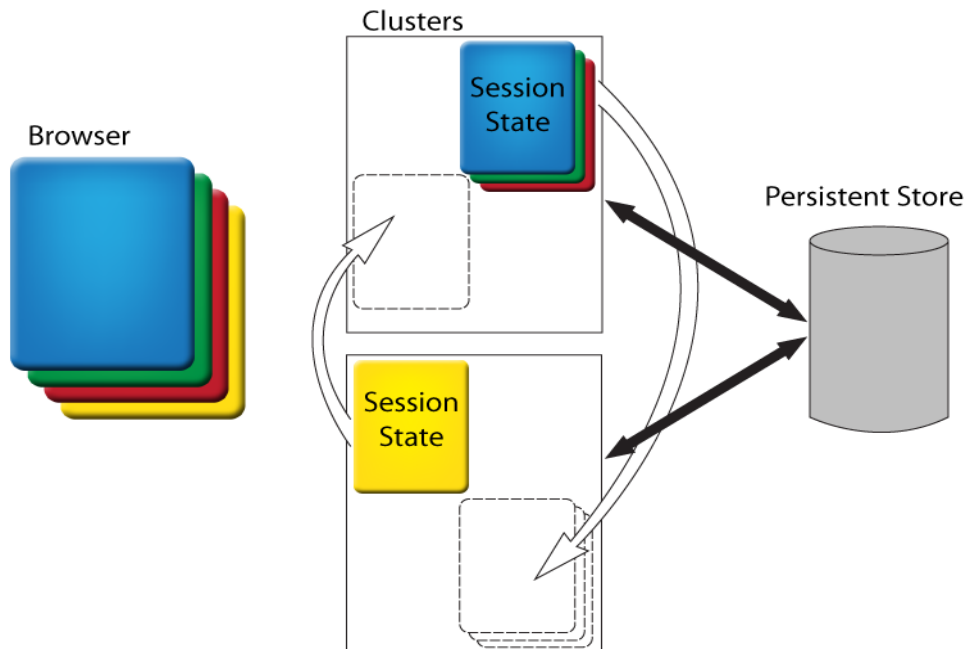
² Persistent state, of course, is kept typically in a database or in a filesystem on the server. The discussion of "state" in this case is the transient state of a logical session

ARCHITECTURAL IMPLICATIONS OF RIA

RETHINKING WEB-BASED APPLICATIONS

That works pretty well for the single-server applications, but the bottleneck that was avoided with the statelessness of the HTTP protocol just moved to the server. If we want to scale our application beyond the largest box, we need obviously need to scale horizontally. In Java application server terms, that means a container cluster. In a clustered environment, the session state is replicated to all of the machines in a cluster.

Clustered Web Application



If the application developers are not aware of the issues regarding clustered environments, then they may place large objects or a large number of objects in a session and create a throughput problem. If the developers are aware of the need to maintain a small amount of session data, then they will likely save transient session data in a persistent storage medium such as a relational database, where it can be accessed by all of the members in the cluster. While this is a workable solution, we now have the applications developers changing the design of their code to compensate for an architectural deficiency. This is not an ideal situation.

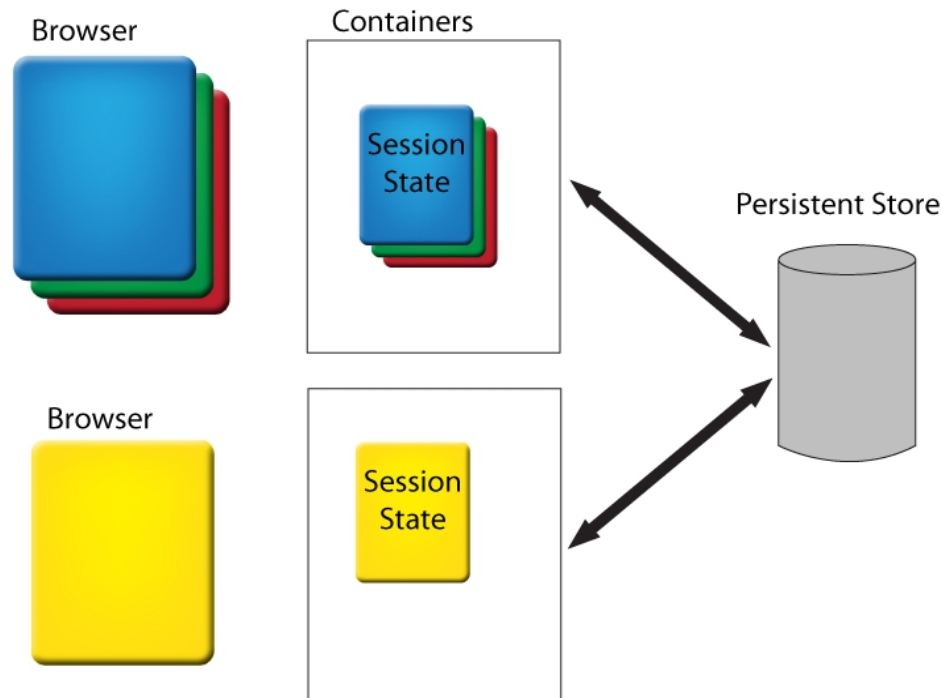
There are other approaches to dealing with scalability. It is possible to implement session affinity. Session affinity means that once a connection is made to a server in the cluster, all interactions within that session must be routed back to that server. This

ARCHITECTURAL IMPLICATIONS OF RIA

RETHINKING WEB-BASED APPLICATIONS

implies that there is additional state - the server to which I connected - in addition to the naturally occurring session state within the application. If that member of the cluster abnormally terminates, then all sessions with affinity to the cluster are terminated. Of course, that is no different than the older, stateful network protocols, but that approach to scalability eliminates the fault-tolerant aspects to the cluster technology.

Server Affinity



SYNCHRONICITY

The other architectural ramification to an HTTP-based protocol is the synchronous request/response model. As mentioned before, the original design intent was to accommodate user-initiated requests for information. Since the premise for web-based applications was originally this paradigm, the natural side effect is a single-threaded program-

ARCHITECTURAL IMPLICATIONS OF RIA

RETHINKING WEB-BASED APPLICATIONS

ming model³. An HTTP request causes the servlet container to borrow a thread from the thread pool and dispatch your request on it. Your code runs on this single thread⁴ until the transaction has completed and the HTTP request/response lifecycle is complete and the thread is returned to the pool.

Now a single-threaded programming model doesn't pose a lot of problems as long as the transactions are very short lived. Keep in mind that our applications "rent" a thread from the servlet's thread pool, so if the application takes too much time to process, we may deny service to other applications in the container or other requests against our application. We have limited our ability to scale by virtue of long-running transactions. Some developers have averted this problem by creating their own threads that are spawned from the servlet request for long running transactions. This violates the servlet spec and should be avoided. See the appendix in this paper for more on that topic.

WHERE WE ARE: A REDUX

Web applications serve a specific kind of application very well. Applications that have user-initiated, short-term, synchronous, single-threaded transactions work great. Everything else starts requires some creativity.

We can violate the specs and implement multi-threaded applications if we are careful to manage our thread's lifecycle with that of the servlet lifecycle. We can just run a long-running transaction and add more threads or container to compensate. We can increase the operational complexity by moving longer-running transactions out of the container through the use of message-oriented middleware or an enterprise service bus, but we often still need to rendezvous with the user-interface. All of these techniques have been done and all can be made to work reasonably well, but we have to realize these solutions are circumventing the original architectural premise of the world-wide web.

3 Of course, servlet containers are multi-threaded, but from an application programmers point-of-view, synchronization is necessary only for application-scope data which is typically read-only. Application programmers are not supposed to fork a thread in a servlet container.

4 Of course multiple simultaneous requests for your application may be going on concurrently within the servlet container, so it is multi-threaded in implementation, but it is a single-threaded **programming model**. That is, the programmer doesn't manage threads and programs as though there is only the thread they are running on at a moment in time.

ARCHITECTURAL IMPLICATIONS OF RIA

RETHINKING WEB-BASED APPLICATIONS

RICH INTERNET APPLICATIONS

At first glance, rich internet applications seem like regular web applications with a lot more visual appeal and user-interface effects. Of course, they *can* be implemented that way, but if we limit our thinking to that approach, there is a missed opportunity.

Many applications have a natural design that deviates from the user-initiated, short-transaction, synchronous model. Some of the business rules that we implement may have transactions times in the several-second range, even though they were triggered from a user-interface event. Many applications have to maintain a significant amount of session state during the lifetime of the application. Finally, many applications want to interact with the user when the user has not initiated a request/response cycle. These include time-based events and responses to longer running background tasks, such as complex business rule computation or interactions with other systems that may be slow or intermittently unavailable.

STATELESSNESS

Statelessness is one of the keys to horizontal scalability and fault tolerance in horizontally-scaled applications. Remember the discussion on the session state in the servlet model? The server has to maintain the state of each client across transactions that are carried out over the stateless HTTP protocol. That becomes a barrier to horizontal scalability since we have to replicate that session state to all of the members of the cluster.

What if the server didn't have to maintain session state? We would have unlimited horizontal scalability at the server level because there would be no session replication to create a bottleneck. We would not have to resort to server affinity since it no longer matters where the server-side transaction runs. We can do that with RIA applications⁵ because it supports a fully-featured object-oriented programming language. It is an application-hosting environment in the browser. It can contain session state in the browser's memory (and in the case of Adobe AIR applications, in the browser-side database or client-side filesystem).

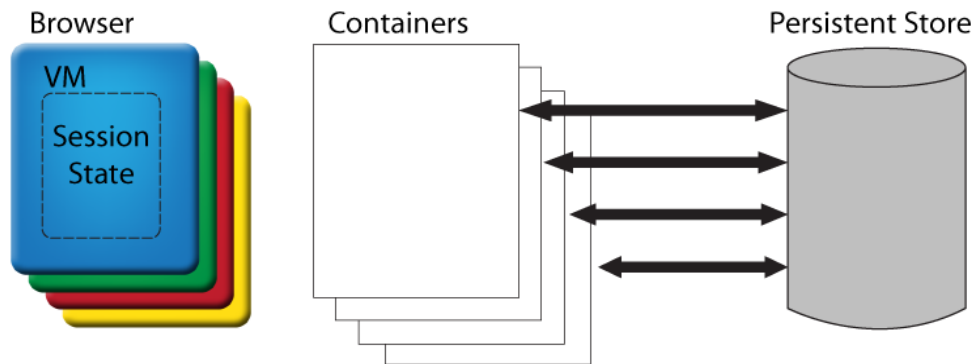
⁵ This discussion of RIA is based on Adobe Flex, since that is the RIA technology with which I am most familiar. I would expect the others - Silverlight, JavaFX, Swing applets, etc. to be similar in most regards

ARCHITECTURAL IMPLICATIONS OF RIA

RETHINKING WEB-BASED APPLICATIONS

That is a significant architectural opportunity. As applications developers, it is important that our designs ensure that the server-side is stateless with respect to transient session data. Obviously persistent state is still stored on the server side database or filesystem.

Rich Internet Application



ASYNCHRONICITY

As mentioned, many applications aren't strictly synchronous, user-initiated applications. There may be server-initiated events, time-based events or long-running transactions that don't fit well within the traditional web-model. This causes an application design that is asynchronous.

ARCHITECTURAL IMPLICATIONS OF RIA

RETHINKING WEB-BASED APPLICATIONS

For long-running client-side⁶ tasks, there isn't an issue. The lifecycle of the client part of the application is independent from that of the server. We aren't bound by the rules of synchronous HTTP or server-side thread pools. For time-based events, again, there is no issue since those aren't bound by the server-side rules that are driven from synchronous HTTP. If the client application needs to initiate transactions with the server on recurring time-intervals, that's no problem since that can happen independently of user-interaction. We are starting to introduce asynchronicity into our application design.

When we do interact with the server, it is most-likely going to be over HTTP, since that is the port that will be allowed through the firewall. That is still a synchronous protocol and if it is talking with a servlet container, we are still bound by the need for short transactions. (More on that in a bit).

However, we are liberated from user initiating the transaction. Since the client application code has the smarts to initiate the transaction on a time-basis we can make requests for information from the server in the background. That gets us closer to server-initiated push, but not quite.

There are multiple techniques that are used to simulate server-push in rich internet applications. One is to do polling on a regular basis. That is inefficient in terms of network and server resources, since a certain amount of polls will return no meaningful information. Another solution has been termed "long polling". In that case, the client begins the HTTP request/response cycle, expecting an eventual response from the server. When the server is ready to push information to the client, there is already an open request on which to place the response. In a traditional servlet environment, this violates the architectural contract for short-term transactions and would quickly exhaust the threads in the thread pool.

There is work being done in the Java community under JSR 315 to implement long polling in a standard way. Until then, there are technologies such as the Tomcat Comet Processor and Grizzly that decouples the threading model from the HTTP request/response model. There is still a risk of limiting scalability by holding too many socket handles open on the server.

⁶ "Client-side" of course means the code running in the browser. For Adobe Flex applications, this is running in the Flash virtual machine implemented as a browser plugin.

RESTFUL WEB SERVICES

When we use HTTP in an asynchronous manner and decouple the HTTP traffic from the user interface, it is helpful to design those HTTP transactions as RESTful services. REST stands for representational state transfer. That is, there is a transfer from the current persistent state as seen on the server to the transient session state of the client.

So much of the potential benefit of RIA applications is centered around shifting session state management to the client and deploying stateless servers. That is exactly what REST enables.

THE NECESSARY STATE?

If we have *a single datum* of transient state at all on the server, we lose the architectural opportunities afforded by rich internet applications. While it makes sense for the RIA application to maintain session state with regard to application-specific data, there seems to be a problem with that one datum that must be shared between the client and the server - authentication credentials.

If we are truly stateless in our interactions between the client and the server, we have to authenticate the client on *every* transaction. The obvious approach, which is to send the userid and password across the network on every transaction, has some problems. First, we most likely don't want in-the-clear authentication credentials flowing across the internet, so we would have to use HTTPS for all of the client/server interactions. HTTPS is more resource-consumptive than the plain-text HTTP protocol, so it would take more server resources to accommodate that approach.

If we send userid/password across the network, we also need to make a query against the persistent storage containing those credentials. That's most likely a relational database or LDAP call, which can be a relatively expensive operation.

The way to eliminate these problems is the use of self-validating authentication tokens. A token is generated on the server upon successful authentication of full credentials, typically a userid/password. That token contains sufficient information to ensure that the client is who they claim to be. The client doesn't have to do any processing of the token. The client merely ensures that it is sent back to the server on every interaction. The server decrypts the token and checks that it is valid. The token should not be validated through a call to a data store, but should contain redundant information that enables it to be self-checking.

This self-validating token can be placed in an HTTP header and exchanged between the RIA client and the server. The details of the implementation this is beyond the scope of this white paper, but you must ensure your implementation is secure. In my implementations, we incorporate the following key concepts into the code:

- The encryption/decryption keys on the server are periodically changed out. In one case, we rotated through a set of 12 keys that were based on the month of the year, although weekly change out would be appropriate for applications requiring more stringent security.
- The authentication tokens generated for the client had limited lifespans.

ARCHITECTURAL IMPLICATIONS OF RIA

RETHINKING WEB-BASED APPLICATIONS

- The authentication tokens were scrambled before they were encrypted. Tokens that always have consistent information at consistent places are easier to crack.

Of course, when a token expires or is otherwise invalid, the client would be challenged to provide the userid/password⁷ as they would be on their initial connection to the application.

⁷ As a side note, if your RIA technology can write persistent data in a secure way, as is the case with Adobe Flex, you can store the userid/password locally and the user can be automatically logged on.

THE ARCHITECTURAL RIPPLE

As noted above, there are some concerns with the threading and lifecycle model of the servlet.

Those concerns are outweighed by the advantages when developing traditional web applications. First and foremost, the servlet container provides the network daemon to listen for socket connections and it provides the implementation of the HTTP protocol. The servlet model transparently maintains the notion of a session across the stateless HTTP protocol. The servlet model provides the programming interfaces to maintain distinct request, session and application-scoped state. If we are using servlets in a clustered environment, there is a mechanism for automatically replicating that state throughout the cluster. That's all good, compelling stuff.

If we implement the right application architecture in a rich internet application, we have eliminated the need for session state on the server. All we really need is an implementation of an HTTP protocol, preferably one that doesn't couple the socket handle to a thread in a thread pool, so we can consider the use of long polling. We want the capability to implement multi-threaded server side application processes for greater asynchronicity without the risk of the servlet container taking us out of service while our specification-violating thread is running. We want to view HTTP transactions not as user-interface request/response transactions, but as requests for state transfer between the transient session state on the client and the persistent state maintained on the server.

By virtue of these newer server-side requirements, we don't really want servlets. We want network services more like those we wrote in the 1990s. This time, however, we will stick with the HTTP protocol. Its statelessness is not an issue, since our servers are stateless and it will scale to a much larger number of clients than the traditional telnet-like protocols of the 1990s.

THE ARCHITECTURE OF CHOICE

The most enjoyable part of being in the information technology business is the constant, often disruptive, change that occurs with the advances in technology and the increasing complexity of the business needs and user demands.

The architecture of choice for most applications - at least for the time being - is an RIA client who handles the user interface layer and the management of transient session state. In many cases, the client can host the implementation of many of the business rules in addition to the user interface.

The server side is an implementation of the HTTP stack that services RESTful service requests from the RIA client, but is not running in a servlet container. I am currently using the reference implementation of JAX-RS being serviced by Grizzly. Grizzly provides an NIO-based HTTP stack, so long polling is possible if absolutely necessary.

This enables the server-side implementation to handle the short-transactions in a traditional manner while allowing long running transactions to be run using threads that are waiting for requests using asynchronous messaging techniques⁸. If there is a relatively few number of long-running transactions, it may make sense to implement long polling for the responses in the client. Otherwise, the client can poll. Ideally, the polling could be based on a transfer of state from the server to the client indicating an estimated time when the results will be available⁹. That would eliminate excessive polling when there is no meaningful result.

The emergence of RIA is a game-changer. It is important that architects look past the visual appeal of RIA to fully exploit the architectural opportunities that are created.

8 This is similar in concept to JMS message queues, but it is thread-to-thread within the same JVM and it scales to very high transaction loads with very graceful degradation under transiently high loads

9 Ideally the server would maintain statistics of real response times so this value would be dynamic and self-tuning

APPENDIX A: UNMANAGED THREADS

This is an architectural paper on the effect of rich internet applications, so this discussion is a little out of scope. However, it is relevant to issues that surround the traditional web application development and some of the more arcane problems that arise from the limits in that architecture.

Web applications in general and the Java incarnation of that in the form of servlets were designed for short-lived, synchronous single-threaded applications. Many applications have portions of them that are not directly initiated from user input (the synchronous model) and some transactions would like to complete the user-interaction lifecycle, but continue to run background processing that takes a longer amount of time. In order to accomplish this an application that is not constraint by the servlet specification would implement a multi-threaded application. Servlet programmers incorrectly assume they can do the same. This can cause a very arcane problem.

The lifecycle of the servlet is managed by the servlet container. It can take the servlet container out of service at any time for any reason as long as all currently running transactions are completed. When the next HTTP requests hits the servlet, a new instance of the servlet is created. That also implies a new instance of the Java class loader for that servlet. Herein lies the problem.

A class is not an `instanceof` another class unless it has the some class definition and it was loaded by the same class loader. As long as the servlet container is allowed to manage the lifecycle of your application, that is not a problem. However, when you create a thread, you have to manage the lifecycle of that thread since it is out of the container's scope. If you have a thread running that was loaded by the first instance of the servlet's class loader, all of the objects running in that thread can no longer be assigned to what seem to be the same class type in the servlet loaded by the second instance of the class loader. Although they might have the same class definition, they aren't the same instance because they were loaded by different class loader instances. You will get `ClassCastException` objects being thrown and it will be very difficult to diagnose.

A similar situation can occur when using multi-threaded frameworks, such as the task-scheduler Quartz, in a container environment.