



From: [www.cio.com](http://www.cio.com)

## How To Develop a Cloud Computing Error Logging Strategy

– David Taber, CIO

**June 07, 2011**

Error handling is such a pedestrian issue, you take it for granted. Same thing with transaction and trace logs — these are just so obvious. In a simple virtualization cloud (e.g., a MySQL server) you can take low-level error logging for granted at several layers of the software stack.

But that doesn't mean you can take persistence of those logs for granted. Your logs may be wiped out if you didn't configure the VM properly, or if you didn't pay your cloud hosting provider to persist the logs if the VM crashes. Without those server-side logs, your troubleshooting is set back several hours, if not days. In unattended systems, the lack of persistent logs becomes a serious issue, and there are good forensic reasons why you'll want to archive logs for long periods. So don't skip here.

The situation is even more interesting with cloud-based applications. Some of them don't have any server side logging at all, and even the best of them do full-scale logging only for a while — turning the logging off without any explicit notification. The best of them have a good story for debugging. But for reconstructing the crime of error conditions that occurred several hours ago...well, if you're lucky, you'll get an e-mail with the error stack trace. Goody.

So in the cloud (and particularly across clouds), you can't really count on server- (or service-) side error logging. Guess what: if you want to know what's really going on with server transactions, you'll have to write your own server-side error-throwing code. Generally speaking, you want to supplement the server cloud's native error logs with calls to a centralized error-logging service like Splunk, Exceptional, or SysLog based offerings. These things are cheap in comparison to an untrapped error.

Further, you need to develop a cloud logging strategy that focuses on the requester (or client) side for persisting the errors. Why do you need to develop this logging code?

- Only on the requester side can you know the application-level context under which the error occurred. On the server side, all the service knows for certain is a requester's URL/IP address, the call parameters, and a time stamp. Of course you could include a client-side thread identifier with each

server call, but why not just do logging on both sides?

- Only the requester can log an error of "server not responding." And of course, only the requester can launch a retry or transaction-recovery strategy if the server's gone silent.
- The requester knows (or can determine) exactly which transaction(s), user(s), and record(s) are affected by the error. The requester's error-handling code may be able to execute a strategy that satisfies the application's needs (e.g., by putting the transaction in an "on hold" state) without the server response. At the very least it can put the application data in a self-consistent state, so that the server error doesn't propagate into an application-wide crash. Further, the requester can put as much transaction context and history as it wants into local log files, to speed troubleshooting and recovery.

Unfortunately, client-side error handling and logging seem to be a "new requirement" for many developers. Just like code comments and documentation. Developers focus on satisfying the positive case, rather than dealing with the inevitable negative case (or even the null case of "nothing happened"). Cloud application developers need to be encouraged to develop better habits of error handling, since they often can't depend on the server logs for the dirty work.

In developing the requester / client side error management, it's a good idea to think about where the logs should be kept. The traditional "throw a message into the text file" strategy doesn't work all that well in a virtualized environment, as that text file may be on another VM, with a file location that might be tough to track down a month or so later. (And yes, your strategy needs to assume that you may not know about the need to troubleshoot/debug for several months after the fact, so the error logs will need to support some level of forensics.)

What about storing the detailed error information in with the record that was involved? With a big stack trace (think J2EE), this can gobble up room. But disk space is disk space, and a few BLOBs won't really slow down your database much. The problem with this strategy is "what happens on the second (perhaps related) error involving a single record?" Your developers are smart enough to figure out how to archive past errors — give it to them as a programming challenge.

A final area of error logging is notification: how is someone supposed to know to go looking for a problem? Many cloud environments don't have a strong feature set for this, which means either a continuing accumulation of error conditions or a periodic chore for your admins and developers. Whether you want to fire off a message to a Twitter feed or turn on the USB port that powers on lava lamps, appropriate real-time notification is the first step toward error containment and rapid recovery.

*David Taber is the author of the new Prentice Hall book, "[Salesforce.com Secrets of Success](#)" and is the CEO of [SalesLogistix](#), a certified Salesforce.com consultancy focused on business process improvement through use of CRM systems. SalesLogistix clients are in North America, Europe, Israel, and India, and David has over 25 years experience in high tech, including 10 years at the VP level or above.*

**Follow everything from CIO.com on Twitter [@CIOonline](#).**

© 2010 CXO Media Inc.