SwiftScale: Technical Approach Document

Overview

This document outlines a technology embodiment of the SwiftScale application including the technology, deployment and application architectures.

Technology Architecture

This proposed architecture will be composed of three server side tiers and a client tier. The initial deployment of the client tier will be based on HTML5, and subsequently native applications will be considered.

Client Tier

Goals for the client tier include avoidance of OWASP vulnerabilities, a high degree of interactivity, solid support for separating design and logic implementation skill sets, and support for metadata driven UI elements. Both native applications and Single Page Apps will be deferred to start. Upon introduction of an SPA, an object-capability such as Google's Caja would be required to manage client side vulnerabilities. For the time being, HTML5 browser capabilities have improved dramatically enabling highly fluent user experience while the risk of managing device resident information in a secure manner has not yet sufficiently matured.

The server side tiers will be:

- Web Session / Interactivity Tier
- REST API
- Database

Interactivity Tier

The interactivity tier will be hosted by the Lift application server. Lift provides a high degree of interactivity through AJAX and Comet. Some benefits of Lift are provided on the Lift website here:

http://seventhings.liftweb.net/index

Benefits key to SwiftScale requirements include:

Security – "resistant" to OWASP Top 10

"Lift at its core seeks to abstract away the HTTP request/response cycle rather than placing object wrappers around the HTTP Request. At the practical level, this means that most any action that a user can take (submitting form elements, doing Ajax, etc.) is represented by a GUID in the browser and a function on the server. When the GUID is presented as part of the an HTTP request, the function is applied (called) with the supplied parameters. Because the GUIDs are hard to predict and session-

specific, replay attacks and many parameter tampering attacks are far more difficult with Lift than most other web frameworks, including Spring. It also means that developers are more productive because they are focusing on user actions and the business logic associated with user actions rather than the plumbing of packing and unpacking an HTTP request. “

“Lift's philosophy of GUID associated with function has the dual benefit of much better security and much better developer productivity. The GUID -> Function association has proven very durable... the same construct works for normal forms, ajax, comet, multi-page wizards, etc.”

“The next core piece of Lift is keeping the high level abstractions around for as long as possible. On the page generation side, that means building the page as XHTML elements and keeping the page as XHTML until just before streaming the response. The benefits are resistance to cross site scripting errors, the ability to move CSS tags to the head and scripts to the bottom of the page after the page has been composed, and the ability to rewrite the page based on the target browser. On the input side, URLs can be re-written to extract parameters (both query and path parameters) in a type-safe manner, high level, security checked data is available for processing very early in the request cycle.”

“The last part of Lift's security focus is SiteMap. It's a unified access control, site navigation, and menu system. The developer defines the access control rules for each page using Scala code (e.g. `If(User.loggedIn _)` or `If(User.superUser _)`) and those access control rules are applied before any page rendering starts.”

Very interactive – Lift provides a direct and simple method for supporting Comet and Ajax. Lift models small grain units of interaction called snippets rather than fitting interactivity under a coarser grain model of the HTTP protocol as is common in classic MVC based application servers.

Designer friendly – UX defined separately from coded functionality. Lift support HTML5 (and XHTML) templates and doesn’t require designers to learn a new tag or object language. Designers and developers have a clean contract to coordinate working together.

Lift will require sticky session load balancing so deployment and application architecture must take this into account. Scaling out will require greater deliberation. But in exchange, interaction context is managed by Lift and not the applications increasing the reliability of more sophisticated interactions.

Lift has reasonable built-in authentication allowing the introduction of centralized SSO capability (e.g. Apache Shiro) to be deferred. As the breadth of the solution increases supported by a loosely coupled architecture, this deferral will need to be revisited. Lift-Shiro integration projects should ease this transition.

API Tier

Scalatra is a light weight and fast framework very handy for supporting APIs. The Swagger API code generator can expedite the creation of the more basic API capabilities. Behind Scalatra, an actor model is supported through Akka. Actor models provide the very responsive, event driven message processing, the ability to implement capability based security models and the ability to expose the operational

semantics as a trace of message activity. Supervisory actor models provide the opportunity for higher level supervisor actors to intervene when threat activity is detected either on a policy or statistical basis. There is also the opportunity to "replay" actor sequences if a perceived potential threat event turns out to be harmless. To protect confidential information, access would be exposed as a set of capabilities based on the principle of least privilege through actors. The message trace of activity is exposed to supervisory actors who enforce policies across the trace and intervene according to policy specified actions when suspect activity is encountered. If the activity is interpreted as a threat incident, the activity will be aborted and higher level supervisor actors will clean up, report the incident and terminate the activity of the principal driving the access attempts while maintaining the trace history of activity to that point. There is the possibility of false positive incident detection. An additional layer of confirmation can request evidence that the principal's access was authorized and not a thread, and the activity preceding the declaration of the false threat incident to that point can be replayed, facilitating bringing the principal back to their last point of action.

An example of such monitoring policy would be to continuously examine the trace of activity looking for a pattern of information lookup failures that would be typical of intruder probing to find weakness at the early stage of an attack. A simple case would have the same principle issuing failing access requests in a short period of time.

DB Tier

MongoDB provides scalable, "document style" persistence that works well with "polystructured" data such as that defined through SwiftScale metamodel based Profiles. MongoDB's architecture strikes a balance between managing consistency of updates and scaling out to support large numbers of users.

MongoDB also has built-in support for file management through GridFS. This capability serves to manage due diligence evidence in the form the various kinds of file artifacts while keeping underlying evidence artifacts synchronized with the associated summarizing Profiles created by the reviewing Subject Matter Experts.

Much of the architectural tier infrastructure is based on Scala, a high level, very expressive language with direct access to the huge set of Java libraries. Scala runs on tried and true JVMs which provide a very efficient execution foundation. The strong and flexible typing facilitates security and cross cutting functionality can be encapsulated well in Scala traits, but support for Aspects work as well.

Casbah provide Object Document Mapping supporting access to MongoDB mapping Scala objects to MongoDB documents. Casbah also provides support for GridFS.

Since SwiftScale data is a high value target, all data should be encrypted at rest and in flight. At rest encryption can be provided by Gazzang which encrypts the data at the file system level.

Information Partitioning

As the volume of data stored in SwiftScale grows, a method to scale storage becomes essential. A common method to scale storage is to partition data across multiple storage servers and volumes. Requests for information are then dispatched to the appropriate partitions based on the values of some set of application information elements serving as the partition keys. To this end, it is helpful to select in the data architecture of SwiftScale which application data fields will serve as the partition keys.

SwiftScale has a natural fissure upon which a partitioning can be implemented. Each Organization manages a pool of information and information flows between Organizations tends to be restricted due to confidentiality needs. Though two Organizations may elect to become very intimate and extensively share information, that sharing doesn't tend to extend to other Organizations simultaneously.

Though Organization provides a partitioning element, the specific partitioning implementation entails a strategy which requires more general properties of the partitioning keys. By itself, Organization may not have high enough cardinality for the partitioning to scale adequately. MongoDB recommends monitoring the growth of the data store and after having identified the growth dimensions, refining the partitioning strategy.

Deployment

The following is a diagram of a sharding version of the deployment architecture.