Engage Server-Side Technology

Overview

This document outlines the specific technologies we propose to use in the Fluid Engage services layer. More details about the overall architecture of Engage can be found on the Engage Architecture page.

The Engage services layer aims to weave together the various resources and sources of information required to create a compelling and interactive exhibit experience. We'll talk to content and collections management systems, image stores and asset systems, as well as social networking services on the Web such as YouTube, Wikipedia, and Twitter. Engage will share these resources back out throughout the system ensuring that it's easy to create great user interfaces for both the Web and mobile devices using these services.

For database persistence and searching, we'll use Apache's CouchDB and Lucene products. CouchDB provides us with a Web-friendly, document-oriented database solution that won't force strict schema requirements on our users and implementers.

For services development, we'll create a new server-side JavaScript Web environment by assembling existing technologies into a lightweight framework called Kettle. This framework will initially be hosted inside the Java Virtual Machine using Rhino, and will eventually migrate to a fast next-generation rutime such as Google's V8 engine.

Goals & Criteria

- Broadly approachable
- Hit the ground running
- Interoperable
- Scalable and forward-looking
- Fits the approach of the Fluid community: functional, declarative, Web-oriented

A broadly approachable services layer technology lets us reach out to a large number of Web developers, allowing them to get involved in the Engage community and technology using skills and languages they may already be familiar with. Similarly, the Fluid community itself needs to hit the ground running with Engage, building the services layer in technologies that are familiar and will get us coding with minimum ramp-up time.

Interoperability is a primary goal of Engage, ensuring that our technologies will work with a wide range of existing authoring tools, content management systems, and databases commonly found in museums and on the Web. The Engage services layer needs to fit in well and leverage open standards wherever possible.

At the same time, our choice for server-side technology needs to be scalable and forward-looking, allowing museums to invest in Engage over the long run, without fear that our services will slow down impossibly when confronted with large collections. Engage services also need to be future-proof enough to not quickly become obsolete as the technology landscape evolves.

Lastly, the Fluid community has built up, over the years, a set of techniques and philosophies for writing software: accessibility, the Open Web, functional programming, and markup agnosticism are common themes throughout our technologies. As a community, we embrace diversity and a wide range of styles; shared patterns and techniques in code help make our solutions more coherent and consistent to develop with.

Some Background Information

We explored a variety of potential technologies for the Engage services layer. In each case, we considered both a programming language and an accompanying Web framework, recognizing that much of the advantage of a particular language comes from good tools we can reuse. We looked at solutions in Python, Ruby, PHP, and JavaScript running on the server.

Our aim was to find a language and accompanying framework that fit both the technological needs of our users and the culture of our development community. Throughout the process, we carefully examined both features and the wider context: the associated community, documentation, and support.

In-depth information about our approach to evaluating technologies, along with our observations, is available at the Fluid Engage Server-side Technology Notes page.

CouchDB and Lucene

Data Feeds and CouchDB

There is an incredible amount of diversity among museums in terms of how they each organize their collections and structure their data. Each museum's collection is different, and it is a daunting and risky task to attempt to force all types of collections into a single schema. Engage's technology needs to embrace this diversity. Rather than creating a "one size fits all" approach at the database level, we'll treat schemas as flexible.

Given that we can't pre-bake the data model completely, a standard SQL-based relational database is inappropriate for Engage. A new crop of document-oriented databases has emerged that are well-suited to handling dynamic schemas. Foremost among these is CouchDB, a highly scalable and Web-friendly database written in Erlang.

Couch specifically addresses the schema problem we face in Engage. Data is stored as documents in JSON format, making it particularly useful in client-side applications that make heavy use of JavaScript. CouchDB Views, which provide a means for filtering, aggregating, and retrieving data from documents in the database, are written in JavaScript and can be used to provide various data feeds from a collection of documents. This fits neatly within Engage's open Web architectural approach, and is approachable to a wide range of developers.

Data in CouchDB is queried and saved using a RESTful API: fetches use a standard HTTP GET request, while additions and updates use simple PUT or POST operations. This means that Couch can be used with any programming language, without requiring extra native database drivers. It's also an excellent fit for Engage's RESTful service-oriented architecture. For museums that already have their own databases or CMS systems, Couch can be replaced with a custom service that provides the same operations and feeds as the Couch Views that will ship out of the box with Engage.

Written in Erlang, Couch is extremely scalable and can meet the requirements of massive, complex collections and exhibit data.

Free Text Searching With Lucene

Users have come to expect natural, Google-like text searches from most modern Web experiences. As a result, Engage will need to include a free text search engine service.

Hands down, the best open source technology for this is Apache's Lucene. Today, it's the fastest and most reliable open source text searching software available, and it has few competitors. Lucene is a JVM-based tool, and should be relatively easy to install in most server environments. Lucene can be easily connected to CouchDB, allowing new documents in Couch to seamlessly be added to the indexes.

We will continue to assess Lucene's associated Solr project, which provides a RESTful API and JSON-based wire protocol ideal for Engage's architecture.

JavaScript on the Server

JavaScript is traditionally considered as a client-only language, running mostly in Web browsers. A number of runtimes do exist that enable JavaScript as a general-purpose programming language, and most modern browser engines can be compiled as standard libraries for use on the server.

At the same time, JavaScript is probably the most widely used and understood programming language today. Regardless of what technologies are used on the server-side, the majority of Web developers also write client-side code in JavaScript. In terms of the ubiquity angle, JavaScript is overwhelmingly familiar and accessible to programmers coming from diverse background and technology camps.

That said, Web development on the server with JavaScript is still largely uncharted territory. There are few production-grade environments available for writing JavaScript Web applications on the server. A multitude of lesser-known projects exist, but few have picked up the momentum and community of more popular environments such as Ruby or Python.

As richer and more complex application code is built in JavaScript on the client side, there is an increasing need to share code and development tools between the client and server. The Fluid community has built up an extremely productive toolset for developing usable and accessible interfaces with Infusion, our JavaScript application framework. As we extend our development efforts to Web services and server-side applications, a goal is to be able reuse as much of the infrastructure we've already got, lowering the cost and improving our productivity.

JavaScript on the server, while still on the bleeding edge, will provide us with a broadly approachable environment for writing server-side code using many of the existing tools offered by Fluid Infusion.

Rhino and the JVM

Currently, the most mature and reliable environment for running JavaScript on the server is Mozilla's Rhino. Running within a Java Virtual Machine, Rhino is used widely for embedded scripting in many Java applications, and also ships as part of the standard Sun Java 6 JDK. As a result, Rhino is widely tested and stable.

Rhino's JavaScript runtime is based on the SpiderMonkey engine that shipped in Firefox 2. As a result, it offers acceptable but uninspiring performance for most Web application tasks.

Rhino's deployment within the JVM brings with it a number of advantages. We can leverage the Servlet API to provide us with a proven interface to Apache and other Web servers. Since threading in Java is well-established, and the Servlet API offers clear semantics for concurrency, the JVM offers an easy way to establish a container and framework that is scalable and concurrency-safe.

Where necessary, we can also leverage other libraries that are available in Java. However, to be very clear, we will limit our use of Java within the Engage services layer to only situations requiring extremely fast performance (such as converting data in bulk) or to clear, easy-to-port algorithms.

Next Generation Runtimes

In the past, JavaScript has been regarded as a slow scripting language. With the browser wars back in full swing, new JavaScript runtimes have emerged in the latest releases of every major browser. Even more than any other scripting language, JavaScript VM optimisation is receiving very significant attention these days.

Google's V8 engine and Mozilla's TraceMonkey engines are both blazingly fast, bringing JavaScript performance at least up to levels comparable to other dynamic languages such as Ruby, Python, and PHP.

The long term roadmap for the Engage services layer will be to move from Rhino on the JVM to one of these next-generation JavaScript runtimes. While there are Web server adaptors for these runtimes available today—most prominently v8cgi—we expect it will take a number of months before these are production-ready. As soon as we're confident in the concurrency, cross-platform support, and reliability of using a next-generation runtime for server-side Web development, we'll switch. In the meantime, Rhino and the JVM will provide us with a stable and acceptable production environment.

Portability and JSGI

With this plan to build first on Rhino and the JVM, and then move to a next-generation runtime as soon as they're ready, the obvious problem is portability. If we commit to Rhino now, will be able to move to V8 or Tracemonkey-based environment later?

The JSGI spec provides a solution to the portability issue. Modeled after Python's WSGI and Ruby's Rack, JSGI provides a simple contract for Web containers, specifying how requests and responses should be handled. JSGI is well-suited to JavaScript's functional style, and is relatively straightforward to implement. Several server-side JavaScript environments, including v8cgi, already support JSGI today, and it's expected that others will adopt it. By coding to JSGI, the Engage server-side technology can be trivially ported from Rhino + Servlets to v8cgi or another next-generation runtime.

Env.js for Browser Compatibility

One of the goals of JavaScript on the server is to provide a natural environment for developers who are familiar with client-side JavaScript programming in browsers. With this, we want to reuse the existing Infusion components and framework without having to modify the code to run on the server.

John Resig's Env.js provides us with the ability to support common JavaScript browser features that are typically missing on the server, including the DOM, local and remote AJAX, and the use of libraries such as jQuery. Inevitably, only a subset of these features are relevant on the server, but Env.js ensures we can heavily reuse existing code, making it easy to start developing in this new server-side environment quickly.

Tying it all Together: Kettle

The Engage services layer will weave together a handful of existing technologies to create a new server-side Web development framework called Kettle:

- Rhino + Servlets for the Web container
- JSGI for portability
- Env.js for standard library features and browser compatibiltiy.
- Infusion for application development

Using Rhino and Servlets, we can easily establish a JavaScript-based Web container, and we'll follow the JSGI spec to ensure its portability to v8cgi and other environments. Env.js will provide us with an interface for standard tasks such as making HTTP requests and file manipulation on the server, all wrapped in a browser-like API familiar to most JavaScript developers. Fluid Infusion will provide us with a framework for most common application framework features such as data binding and events.

The only substantial new code we'll need write is a URL routing scheme, most likely inspired by CherryPy's approach. With this, Kettle will offer a full-featured framework for server-side development.

Summary

Using JavaScript on the server side to build the Engage services layer is unconventional approach and not without its risks. That said, it provides us with a number of compelling benefits. This architecture...

- is familiar and broadly approachable, both by our community and by the majority of Web developers
- allows us to reuse existing code and infrastructure such as Fluid Infusion, allowing us to get started fast
- enables us to deploy components and markup either on the client or the server, based on device or app requirements
- can scale to meet the needs of museums with large collections and many users
- is unlikely to become obsolete in the future