Server re-architecture

Progress report September 2012

1. Re-defined goals

During the month of September, the remaining Sakai OAE project team has started to execute on the alternative scenario that was agreed upon by the Steering Group at the end of August. This plan recommended the retirement of the existing back-end technology, Nakamura, and its associated technical debt, performance and production problems that have accumulated over the years.

The back-end will be re-built from the ground up using a multi-tenant, cloud compatible architecture that can scale to the needs of many institutions combined, albeit with a reduced initial scope focussing on a strong content creation, collaboration and networking platform. This multi-tenant, cloud compatible architecture is attempting to anticipate a scenario where groups of institutions will host such an application together on the one hand, and many institutions will use the application as a SaaS solution on the other hand.

The existing Sakai OAE UI and the technologies backing it will be retained as much as possible, and it is assumed that sufficient non-implemented design work is available to support feature and refinement work in the next 6-7 months. The widget technology, as well as the Widget SDK, will be retained and improved along the way and easy end-user contribution will remain a strong focus. Application data will still be served as RESTful JSON feeds, using endpoints that conform to the re-designed REST endpoint specification (see section 3).

This report aims to provide an overview of the decisions that have been taken, the suggested architecture, output of the work that has been done so far, the work that hasn't been done yet and next steps to be taken. For each decision, a set of pros and cons will be listed that reflect the team's thinking and decision process.

2. Investigated approaches

In order to get the re-architecture process started, the team gathered face-to-face in Cambridge and selected 3 options that would be considered for this architecture. There was a strong desire to decide between these various option as fast as possible, so the team could focus on validating that approach, failing quickly and revisiting the decision if necessary.

In terms of storage solution, the team decided to follow the OmniTI recommendation to only try and support a single storage solution and design and tune for that solution. The team quickly settled on Apache Cassandra, an open source NoSQL distributed database management system, designed to handle very large amounts of writes and data spread out across multiple commodity servers while providing a high-availability service with no single point of failure.

The reasons for selecting this technology were:

- Initial investigation of our performance requirements and data model pointed to strategy where data is published in a de-normalized fashion that conforms with how the data will be read, making access to the data very efficient. Apache Cassandra is optimized for this.
- Apache Cassandra is designed to run in the cloud across multiple nodes and even multiple data centers. It is easy to add additional Cassandra nodes on the fly, and has a built-in administrative console that can be leveraged (see section 3).
- If used the way it's intended to be used, Apache Cassandra has very good reports in terms of scalability and performance, which is proven by its use by Netflix, Reddit, Twitter, etc. Read and write throughput increases linearly as new machines are added.
- Apache Cassandra came strongly recommended by the rSmart team.

Some of the downsides of this decision are:

- Both the team and the Sakai community in general have almost no experience with developing for Apache Cassandra and running it in production. Luckily, OmniTI has a fair amount of NoSQL experience that it can share, although they have tended to work with Riak instead of Apache Cassandra.
- Running Apache Cassandra in production is more expensive for small scale deployments than running a traditional RDBMS. It is recommended to have at least 3 Apache Cassandra nodes in a cluster, whilst a traditional RDBMS could be run with just 2 instances (master-slave). However, given the deployment model, the required scale and the expected publishing model, this felt as a reasonable compromise.

2.1. Option 1: Apache Felix + Apache Shiro + Cassandra

Option 1 is based on a Java stack that is not very different from the existing Nakamura stack. It exists out of basic OSGi components running in an Apache Felix container on top of Pax-web. Simple servlets could be registered using JAX-RS which takes care of all data serialization, endpoint registering and parameter parsing. The Hector driver is used to interact with Cassandra and Apache Shiro could be used for authentication and authorization purposes.

Pros:

- Architecture that encourages a good Service Oriented Architecture
- Straightforward for creating REST endpoint
- Apache Felix provides out-of-the-box global administration and configuration abilities (no tenant adminstration though)
- Java-based, just like the Sakai community

Cons:

- The OSGi learning curve is quite steep
- Embedding 3rd party dependencies in OSGi is not very well supported yet in the Java community
- Limited to Jetty
- Compiling is required for code changes

2.2. Option 2: Django

Option 2 is based on Django, which is a high-level framework on top of Python that follows the model-view-controller architectural pattern. It uses django-piston for creating REST endoints, and Django-cassandra-backed for interacting with the Cassandra database.

Pros:

- Allows for good reusability and plug-ability of components
- Fast development cycle
- Very active community and breath of 3rd party components
- Built-in administrative interface

Cons:

- Prone to scaling problems at very large scale
- Can be hard when trying to do something that's not in line with the framework
- Python is not common in the Sakai Community

2.3. Option 3: Node.js + Cassandra

Option 3 is based on Node.js, a Javascript based technology using event-driven asynchronous I/O to minimize overhead and maximize scalability. It used Express.js for creating REST end points, and interfaces with Cassandra using the Helenus module. Authentication is done through integration with the Passport.js module

Pros:

- Fast development cycle with minimum reload or rebuild time.
- Very active developer community with many available modules
- Javascript based, offering the opportunity to re-use code in both UI and server
- Has inherent support for push, allowing for pushing data to the UI in real time (chat, concurrent editing, etc.)

Cons:

- Needs careful application design and testing to avoid blocking code
- Node.js is not common in the Sakai Community, even though Javascript is widely used in Sakai OAE

2.4. Decision

The team has made the assumption that it is very unlikely that the framework used is going to be the bottleneck in terms of performance. It is much more likely that the database and search will be the real bottleneck

After setting up various Proof of Concept project for all of these different options, the team fairly quickly settled on option 3, which was very much a pragmatic decision. Setting up the stack for Option 1 was a painfully slow process as the team quickly found itself in OSGi dependency hell. Only half of the reduced team felt comfortable with this stack, which would impose on the initial development rate of back-end features. Option 2 was considered to be a much more viable solution, but was not chosen because none of the team members had any real experience in Python or in creating a Django application. Although Node.js is not well known in the Sakai Community, it's the best known technology in the accumulation of the project team.

On top of that, option 3 came recommend by OmniTI, who use Node.js heavily in production themselves and can thus provide valuable production experience. It generally gets really good reports for performance and scalability, and is driving high traffic applications like the LinkedIn mobile application, etc. and the inherent support for push technology is really important factor as well. Combined with a very active community, a rich ecosystem of 3rd party modules and based on the team's skill set and previous experience, option 3 was chosen as the path forwards.

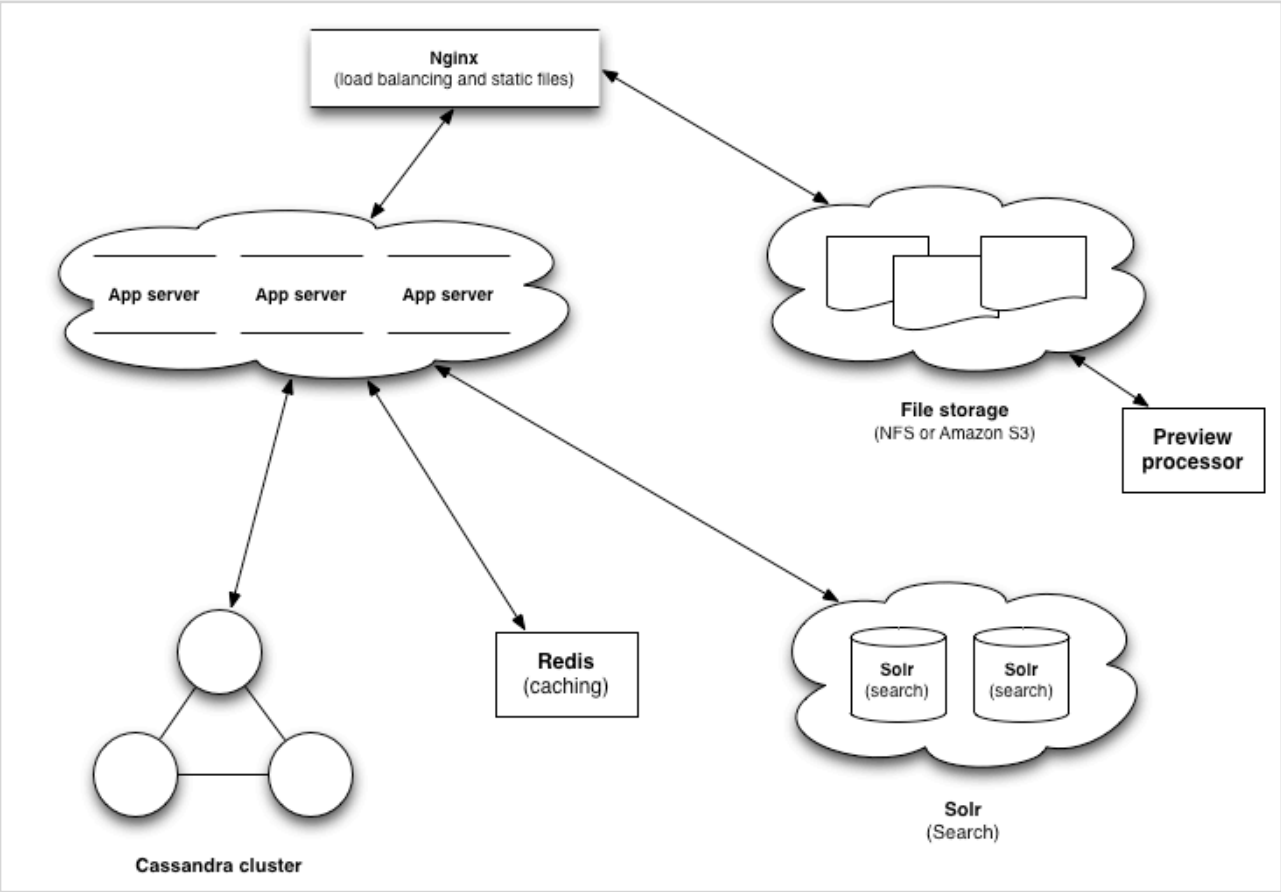
Note that a scalable solution could probably be built on top of all suggested options, the team has just made a pragmatic decision based on what it is most comfortable with and where OmniTI can provide the most help.

3. Output

Over the last month, the team has tried very hard to resist the temptation of re-implementing too much at once. Despite the time pressure at hand, we have attempted to follow a qualitative regime of code review, high unit test coverage and continuous benchmarking and performance testing, which is absolutely necessary given our lack of QA resources.

3.1. Architecture

The thinking around how the full architecture and a full (cloud) deployment would look like is still evolving, and will likely be evolving further as testing and implementation progresses. The current working assumption for a production environment looks like this:



It consists of the following components:

- **Nginx:** The Nginx server would be used to load balance incoming requests to the various app servers and serve static files (UI) as well as content (uploaded files and thumbnails) from the file storage solution. This front-end is recommended for use with Node.js and is used widely on the internet (15% of all websites on the web).
- **App servers:** A number of Node.js app servers that serve all REST requests and talk to the Cassandra database cluster and the Redis caching server. These app servers could be located in various data centers across the world to minimize latency between the end user and the application. A minimum of 2 app servers would be recommended in a production environment.

- **Cassandra cluster:** A ring of Cassandra nodes that store all of the user data, other than fine content. This allows usage of Datastax OpsCentre to monitor the health of the cluster and makes it easy to add new nodes on the fly. A minimum of 3 cassandra nodes is recommended. OpsCentre for our cloud deployment can be found at http://ec2-184-72-16-232.us-west-1.compute.amazonaws.com:8888/.
- **Redis:** Used for caching commonly accessed data and all of the user session information.
- **Solr:** Used for running search queries. Compared to Nakamura, Solr will be offloaded quite a bit as most requests will be served by Cassandra. Solr will now only be used for actual search queries.
- **File storage:** Detected storage for user uploaded files and thumbnails for those files. OmniTI recommends that cloud deployments use Amazon S3 for this service, whilst smaller deployment could use NFS storage.
- **Preview processor:** Dedicated server used for generating thumbnails, PDFs, etc. for user uploaded files. This server could also host video conversion software like Kaltura.

The advantage of this stack is that all of the bits included in it are quite common in modern web applications. Specializing individual components based on a specific task that needs to be executed has been an early OmniTI recommendation and is necessary when running a system at very high scale. The Node.js + Redis + Nginx combination is the most common Node.js architecture around and has thus been sufficiently tested in real-world deployments.

The downside of this stack is that setting up a new Sakai OAE installation is going to require more infrastructure and more tems to be set up initially. Obviously, the installation would run at a much higher scale and could be serving multiple schools at the same time, pointing at larger cloud deployments as a default scenario. Once the infrastructure is in place, configuration should be a lot easier due to the administrative UI that is being built, and additional tenants can be spun up at will.

The team aims to maintain its own reference environment using a cloud provider and use this environment to ensure deployability and running of all of the performance and load tests.

Overall, some of these technologies are new to the team and deployers in the community, but so far the team has found Cassandra (CQL), Nginx and Redis to be quite straightforward to work with.

3.2. Multi-tenancy

Sakai OAE multi-tenancy is envisioned in such a way that multiple institutions/schools share the same physical instance and each of the tenants have got their own hostname that can be used to access that tenant.

Currently, the system provides global and tenant administrators. A global administrative User Interface will be created that provides an overview of all running tenants, and allows for starting and stopping these, as well as spinning up a new tenant on the fly. The global admin UI also allows for setting the different configuration values overall or for each of the tenants specifically, and allows for specifying which configuration values can and cannot be overridden by tenant administrators.

Various authentication methods will be supported, like Shibboleth, CAS, Facebook, Google, Twitter, etc. and any combination of these can be used for each of the tenants.

Sakai OAE multi-tenancy is designed in such a way that allows for permeability between various tenants. If permeability is turned on in the configuration, it allows for users, content and groups to be accessed across tenants.

Per tenant skinning will be provided, although the exact mechanism still needs to be figured out. Obviously, because of the shared code and multi-tenant nature, per-tenant customization to the system will have be scoped more tightly, even though being able to provide functionality to a subset of tenants is built into the design as well.

We do not currently allow for multi-homed users where a specific user belongs to multiple tenants at the same time, both because of technical and UX concerns around this.

3.3. Implemented functionality

This section provides an overview of the actual work that has been implemented by the team in the past month:

- **API specification:** First of all, the Sakai OAE API specification has been redesigned to provide a more optimal and usable set of REST endpoints that attempt to be less generic than the Nakamura ones and aimed more at specific tasks. This will make these endpoints quite a bit easier to use for the UI, deployers and widget developers. The endpoints have been geared towards the new Single Page Application design in anticipation of the move towards that. A copy of the API specification document has been attached.
- **Multi-tenancy:** Multi-tenancy as described in section 3.2 has been designed into the core of the application. Multiple tenants can exist and will run next to each other, tenants can be stopped and tenants can be created on the fly. Data can be shared between tenants when configured that way.
- **REST endpoints for users:** user functionality and their REST endpoints have been implemented as specified in the API document. This includes account creation, getting the me feed, looking at user's basic profiles, user permissions, etc. It also implements the user privacy strategy that has been discussed at the URG.
- **REST endpoints for user authentication:** using the Node.js Passport module, per-tenant user authentication is currently supported for Internal Login, Google, Facebook and Twitter. Shibboleth authentication is currently in progress as well.
- **REST endpoints for user profiles:** profile functionality and their REST endpoints have been implemented as specified in the API document. This includes updating sections, reading section and section permissions as specified in the privacy strategy.
- **REST endpoints for groups:** group functionality and their REST endpoints have been implemented as specified in the API document. This includes creating simple groups, adding and removing managers and members, updating a group's basic information, listing group members. It also replicates the group-based permissions system, where permissions can be propagated through group memberships.
- **REST endpoints for content:** content functionality and a number of their REST endpoints have been implemented as specified in the API document. This includes creating content, getting content profiles, adding and removing managers and members, sharing content, updating content metadata, user libraries and group libraries.
- **Cloud environment:** A reference cloud environment has been set up using Amazon EC2, containing an Nginx front-end, 2 Node.js app servers, a Redis caching server and a 3-node Cassandra cluster, which will be used for all future testing. Because of the DTrace capabilities on SmartOS, we might switch over to Joyent for testing.
- **Performance testing:** Various micro benchmarks and Tsung performance tests have been run. More information about this can be found in section 4.
- **Administrative UI:** We have commenced building a global and tenant administrative UI. More information about this can be found in section 3.4.
- **Unit testing:** Special attention has been paid to providing very thorough unit testing coverage for our codebase, which allows for verification that the code is working. This is crucially important as we currently don't have any QA resource available. The team is using Mocha for its unit tests, and code coverage is currently at 89%. A screencast showcasing this can be found at http://screencast.com/t/UulsASO48.

- **Widget development:** A strategy has been designed in which Sakai OAE widgets can be wrapped inside of official node modules. At that point, the native NPM (Node Package Manager) can be used to easily distribute and install these widgets. It also makes it possible, although still optional, for widgets to start defining their own REST endpoints if that would be necessary.
- **Microsite:** The team has finished the Drupal implementation of the Sakai OAE microsite, which can be linked to from the Sakai Foundation website once it has passed review. The microsite can be found at the following temporary URL: http://ec2-23-20-61-158.compute-1.amazonaws.com/.

Most of this work is not very visual in nature, although a lot of REST endpoints are now available for hooking up a UI. We do expect the nature of the team's output to become more and more visual over time.

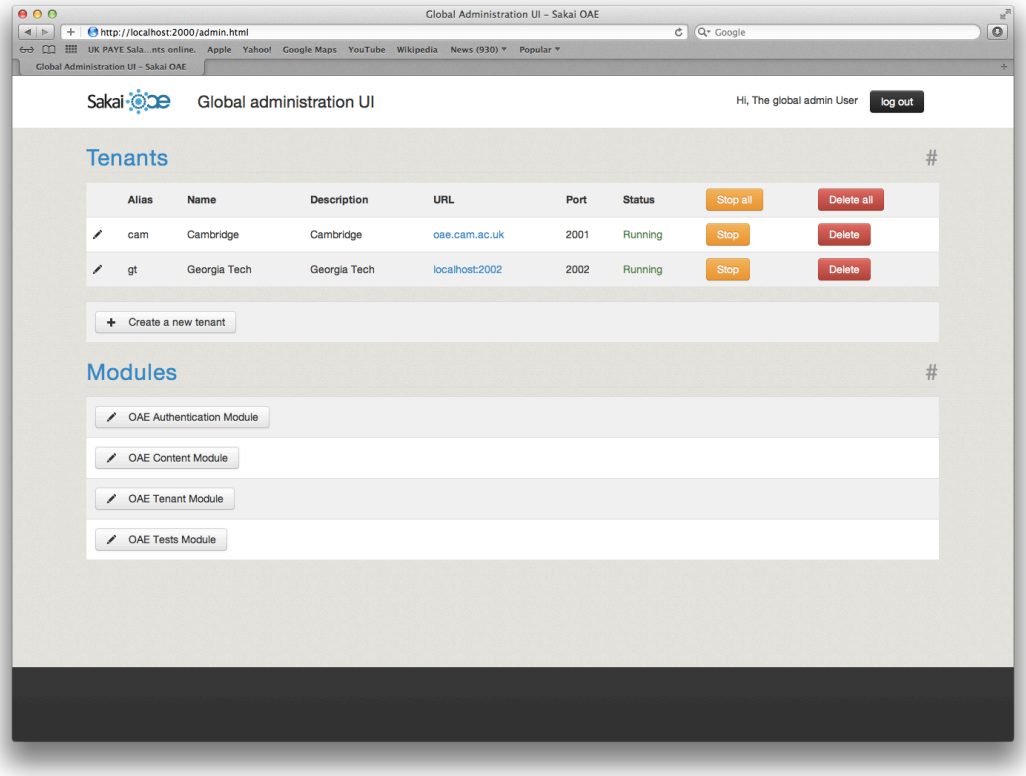
The following things have not been implemented yet:

- No search capabilities have been implemented yet, in order to keep the initial scope as focussed as possible. However, a lot of the data that is now being served from Cassandra using the new REST endpoints used to be served by Solr in Nakamura.

  The current thinking is that Solr would still be used for searching, but we would limit its use to what it was designed to do. OmniTI does not anticipate any problems with this, although incorporating content privacy into search results is currently the biggest challenge on the horizon.
- In order to keep the initial scope as focussed as possible, creating content does not yet store the actual file content. It is currently limited to storing the content metadata. Related to that, it does not yet store any version history for uploaded content. Sakai Documents are not yet supported either.
- The Sakai OAE User Interface has not yet been hooked up to the new back-end, although this work will be started soon. The team has deliberately chosen to first hook up the existing Sakai OAE UI to the new REST endpoints and not yet implement the newly designed SPA, to try and keep as much focus as possible on providing a solid back-end foundation. However, the team wants to move to implementing the new SPA designs as soon as it thinks that it can deal with that distraction.

3.4. Administrative UI

Work has started on the implementation of an administrative UI:



There is a global administrative UI that lists all of the available tenants, and allows the global administrator the stop, start or delete these tenants. New tenants can be started on the fly as well, and will be immediately available for usage.

Every Sakai OAE module can specify how it can be configured. The global administrator can modify that configuration for all of the tenants or for some tenants specifically. The global administrator can also determine which configuration can be overridden by the tenant administrators.

Tenant administrators can visit the tenant administration UI and change the configuration values that they are allowed to change.

Currently, the only setting that has been made fully configurable is which login strategies that should be offered on each of the tenants. More configurations options and types will be made available soon.

A screencast showcasing the administrative UI and tenant functionality can be found at http://screencast.com/t/cYlHxQgdKwTN.

4. Performance testing results

A variety of performance and scalability related tests have been run so far.

4.1 Micro benchmarks

First of all, various micro benchmarks have been run for each of the modules. These micro benchmarks hammers on one specific area of functionality as fast as they can, and do this by going through the service layer rather than the REST end points. These tests are not very meaningful in themselves, but do provide a rough idea of the performance and throughput on that area of functionality.

Overall, these benchmarks have shown very promising results, with a maximum number of permissions checks of about 1878 per second and user/group libraries at about 900 per second using the cloud environment. More of these will be created and will be fed into the performance testing reporting UI.

4.2 Data loading

Next, a full data load using a model generated by the Model Loader and using a concurrency of 5, 100,000 users, 100,000 groups and 100,000 content items have been loaded onto the cloud environment. This performed 670,000 REST requests, and completed in roughly 3 hours, equating to about 60 REST requests per second. This includes all of the publishing of things like user libraries, which should be optimized for fast reading.

The team will be doing more instrumentation and measurements around this dataload to better understand its current bottlenecks, but it is orders of magnitude better than anything we've been able to push into Nakamura in recent efforts. However, we should keep in mind that the new back-end is not yet feature complete and so that is not a fair comparison.

4.3 Tsung performance tests

Some basic Tsung performance tests have been created as well, simulating users hitting the system. For most areas of the system, very encouraging results were found. For example, both user libraries as well as user profiles were comfortably served in around 50ms during the test

These tests did expose 2 clear bottlenecks. The first one is related to the BCrypt encryption algorithm that's used to encrypt user passwords and check if they have entered the correct password, which is deliberately slow for security purposes. However, discussion with OmniTI has resulted in a recommendation that this is not necessary and a faster hashing algorithm would be more appropriate. This recommendation will be implemented in the next week.

The second bottleneck is caused by deep group memberships, where there are multiple layers of groups that are members of each other, which needs to be traversed when doing permissions checks. Initially, the team stored these in a de-normalized way, so that no traversal would be necessary. This was removed to avoid doing premature optimizations. In discussion with OmniTI, we have decided to start storing these group memberships in an exploded fashion and performance tests on this. If that would still be a problem, we could start storing these memberships in a graph database, but there are currently no indications that this will be necessary.

As the team was starting from scratch, a number of initial problems are fully expected. It's currently very exciting that these problems are known, solutions are being implemented and performance testing will be done to show the difference. Once the existing set of implemented features shows no clear bottleneck, there should be a very strong foundation in place that will allow for building on top of that.

5. Next steps

Next, the team will be focussing on the following tasks:

- Run longer and more meaningful performance tests and publish the numbers
- SSO integration for user authentication
- Optimizations in the area of deep group membership
- More progress on the admin UI
- Increasing the number of performance tests and setting up an automated performance testing environment
- Hooking up the UI to the new REST endpoints
- Search functionality

6. Recommendation

Overall, the team feels more confident than a month ago that the goals that have been set out can be achieved in the time that's left. The team is much further ahead in its progress than it had planned for, and most of the early results are very encouraging. Obviously, there is a lot of work left, but no insurmountable issues have been discovered yet.

Due to this encouraging progress, I strongly recommend the continuation of the current server re-architecture effort, and would propose to set another deadline for a continuation decision at the end of October, where we can check the progress that has been made against the points above.