Richard Good

IS Applications Division  - UNIVERSITY OF EDINBURGH

API004

Enterprise Data Services Summary Report

1 Introduction 4

2 Executive Summary 5

2.1 Microservices deployed to Dev 5

2.2 Microservice standards 6

2.3 Planned microservice deployments 7

2.4 Unplanned microservice developments 7

2.5 Future developments 8

3 Microservices created 9

3.1 Student Microservice 9

3.2 Central Authorisation Microservice 9

3.3 Student Finance items Microservice 10

3.4 IS Alerts Microservice 11

3.5 Library Microservice 12

3.6 Event booking Microservice 12

4 Security and Authorisation 13

4.1 Summary 14

5 API Documentation 14

5.1 Summary 15

6 Automation 15

6.1 Summary 15

7 Audit 15

7.1 Summary 15

8 Version control 15

8.1 Summary 16

9 Monitoring 16

9.1 Summary 17

10 Scale and Fault tolerance 17

10.1 Rationalisation of pre-production infrastructure 18

10.2 Circuit Breakers 18

10.3 Grow and shrink demand using service discovery 18

10.4 Summary 19

11 Service culture 19

12 References 21

# Introduction

This document is a summary report of the API004 Enterprise Data Services innovation fund project. To quote the brief of the project:

Currently, there is no easy or standard way to access our data. When staff and students look to produce innovative new projects, they often end up having to use CSV files, or static data, which is very limiting.  Innovation is stifled by these limitations.

We need to:

* Foster a better culture of creative uses of central data
* Make sure data is accessed in a secure, controlled and auditable manner
* Introduce consistent data usage across applications at the University
* Reduce misinterpretation of corporate data used in applications across the University

From the current strategy of harmonising the University portal and web site, to access to open data and the ability for schools and departments (including those within IS) to rapidly and consistently develop applications, it is clear we need an implemented strategy for consistently delivering data in a robust, secure manner.

This project will implement a “mid-tier” secured set of REST-ful web services delivering data in JSON format.

This project looked to deliver a set of web services which would both help set the design pattern and software standards for deploying web services, and also build candidates for actual production-ready code.

The project took an initial design decision to adopt the [microservice](https://en.wikipedia.org/wiki/Microservices) design approach, in our case in creating small focused services on a particular business area or entity. This document therefore makes heavy references to microservices from here on in.

The next section contains an executive summary of the key recommendations and deployment plans. Subsequent sections provide details on specific aspects of microservice deployment.

# Executive Summary

## Microservices deployed to Dev

The following microservices were created as part of this project:

Figure 1. Microservices deployed as part of this project

Each of the microservices built by this project have the following features:

Figure 2. Features of each microservice deployed

**NOTE:** The Notification Backbone microservice pre-exists this project, and has been taken as far as Test by [the WEB007 project](https://www.projects.ed.ac.uk/project/web007).

## Microservice standards

The following recommended standards for microservices have or will be submitted to [the IS Applications Operational Guidance Group](https://www.wiki.ed.ac.uk/display/insite/About+the+Operational+Guidance+Group) for ratification:

* **All microservices are HTTPS only**
* **All microservices use OAuth2 for authorisation**
* **All web clients using microservices authenticate via EASE**
* **All OAuth clients are set up with basic access authorisation, e.g. read, write on service x**
* **All Client (server) credentials are encrypted in the OAuth2 database using** [**bcrypt**](https://en.wikipedia.org/wiki/Bcrypt)
* **Fine grained access decisions are made in individual services using client and/or user**
* **All client access (user or machine) is logged in OAuth server logs**
* **Each microservice will be deployed in a separate Tomcat container**
* **Each microservice will be individually health-checked**
* **Microservices will be versioned using Semantic Versioning**
* **URLs will be provided for each service which allow stable and latest versions to be requested**
* **All microservices will be documented using Swagger**
* **All microservices will have automated software deployment via Bamboo**
* **All microservices will have comprehensive test coverage**
  + **ideally also covering typical business UAT test cases**

## Planned microservice deployments

The following microservices are being taken through to live by other projects:

* Notification microservice (WEB007 to Test, WEB010 to Live)
  + These projects will also deliver the OAuth2 server
* Library microservice (LMP006)

## Unplanned microservice developments

The following microservices have no confirmed plans to put them into production:

* Central Authorisation microservice
  + If live could provide client side user lookup
* Event Booking microservice
  + If live could allow sites to embed events using client side scripting
  + Could allow MyEd event booking channel to be replaced with client side implementation
* IS Alerts microservice
  + If live could allow sites and MyEd channels to embed service specific alerts (removing need for manual channel editing when service issues occur)
* Student microservice
  + If live could allow applications and schools to access definitive data
  + And allow MyEd student channels to be replaced with client-side scripts (reducing load on MyEd)
* Student Finance Item microservice
  + If live could allow MyEd Student Finance channel to be replaced with client-side scripts (reducing load on MyEd)

## Future developments

The following are future developments which are recommended, but have no confirmed plans:

Figure 3. Recommended future enhancements for Microservices

# Microservices created

The following sections summarise the microservices which have been created during this project, and future plans (if any) for them.

## Student Microservice

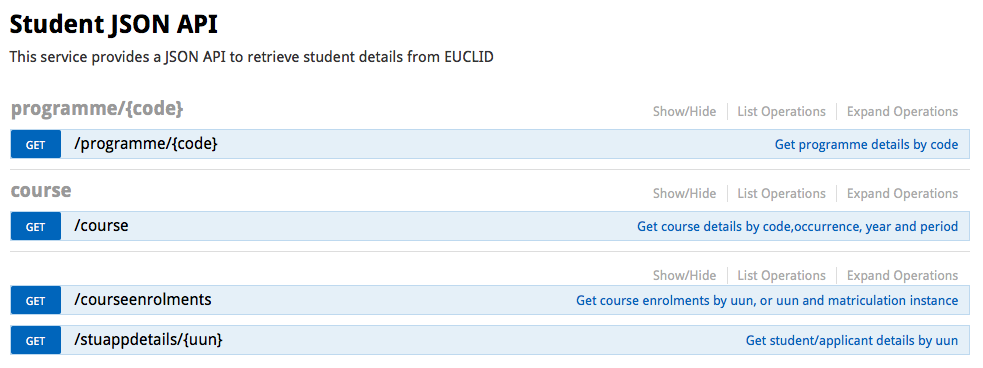


Figure 4. Student Swagger API screenshot

This microservice queries the EUGEX database and returns the following:

* Programme by code
* Course by details
* Course enrolments by UUN and optional matriculation instance
* Student/Application details by UUN

There are currently no plans to take this into production. Access to an API containing information on students is consistently something which schools ask for.

## Central Authorisation Microservice

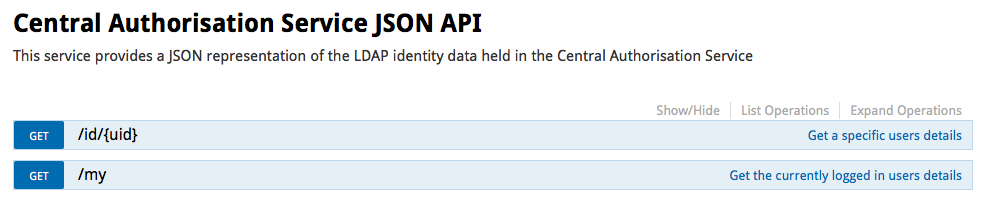


Figure 5. Central Authorisation Swagger API screenshot

This simple microservice sits over the Central Auth OpenLDAP directory, and allows:

* The lookup of a user by UUN
* The retrieval of the currently logged in user, which is intended for web client access

There are currently no plans to take this forward to production. Given this web service allows a web client to look up the currently logged in user, it is an invaluable one for allowing applications to look up the current user without requiring back-end integration.

## Student Finance items Microservice

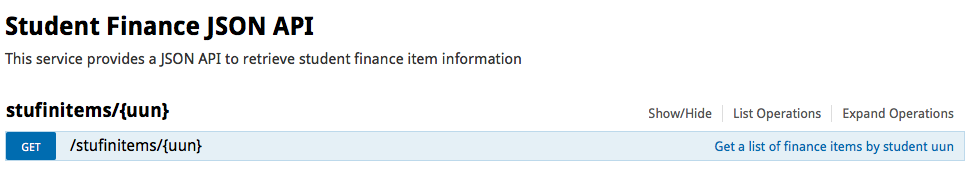


Figure 6. Student Finance Swagger API screenshot

This simple microservice makes use of the generic student finance item database view built for the Cold Fusion MyEd Student Finance channel, and displays:

* A list of student finance items given a UUN

There are currently no firm plans to take this into production, however the advantage of doing so would be to allow the Student Finance channel to be migrated from WebProxy Cold Fusion to being delivered in client-side HTML/JavaScript.

## IS Alerts Microservice

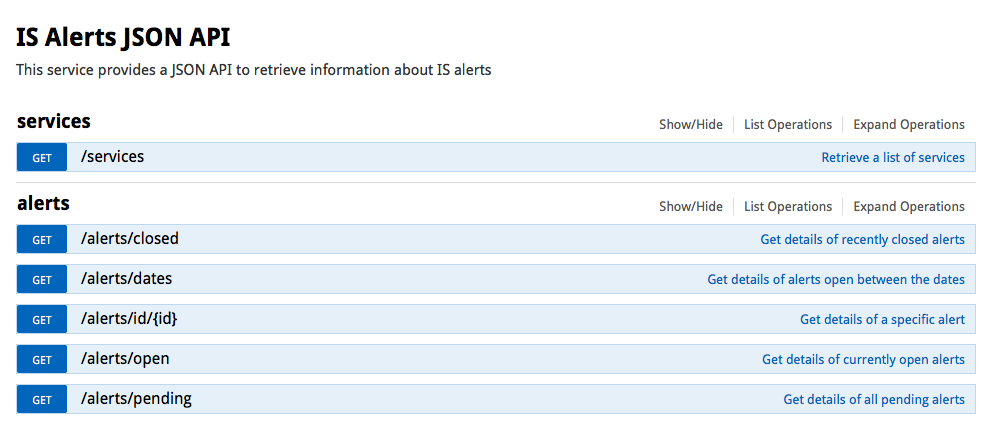


Figure 7. IS Alerts Swagger API screenshot

This microservice allows the lookup of IS Alert information with the following features:

* An alert by id
* A list of alerts by date range
* A list of alerts by service
* A list of recently closed alerts
* A list of pending alerts
* A list of open alerts

There are currently no firm plans to take this service into production, however the advantage of doing so would be to allow the current service alert display to be delivered client-side rather than using Proxy. An additional advantage would be to allow services to query their own alerts and display outage notifications within their own system, and/or MyEd portlets to display dynamic notifications in portlets when alerts exist for the service to which they pertain.

Ideally this service should move to an entirely cloud hosted one, so that University outages do not affect the ability to create and retrieve alerts.

## Library Microservice

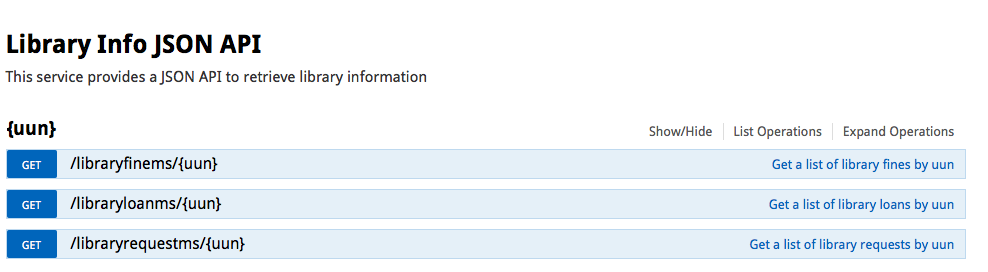


Figure 8. Library info Swagger API screenshot

The Library microservice reflects the features of the MyEd portlet, in that it provides a lookup on the following information:

* Items on loan given a UUN
* Requested items given a UUN
* Fines/fees given a UUN

These features are essentially ported from the live MyEd channel, and as such are considered production ready.

The LMP006 project is going to deliver the ability to create an account in the library system, it will take the currently developed features and the new account creation feature and deliver the microservice to live. This will then allow the MyEd channel to be switched to use the microservice, and the complexity of the MyEd Library Account portlet dramatically reduced.

## Event booking Microservice

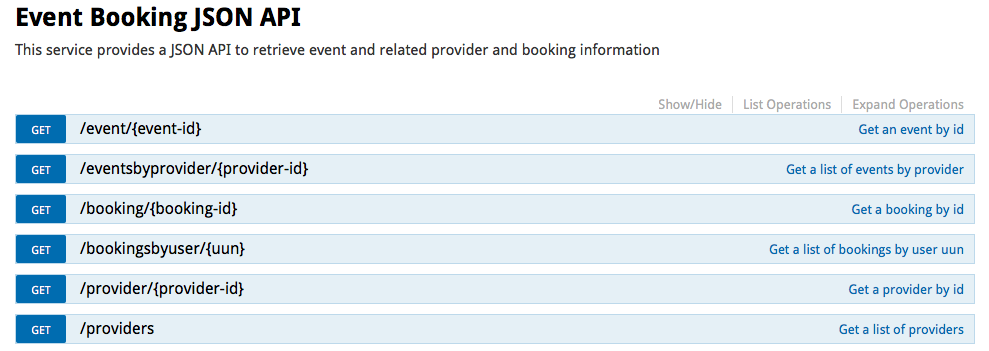


Figure 9. Event Booking Swagger API screenshot

The Event Booking Microservice provides standard features for querying the event booking system:

* Get event by ID
* Get list of events by provider ID
* Get booking by ID
* Get list of bookings by UUN
* Get a provider by ID
* Get a list of providers

There are currently no firm plans to take this into production. Having this service available would allow sites to display the variety of events, bookings and providers as per above. It would also allow the MyEd event booking channel to be replaced with client side implementation.

# Security and Authorisation

Key to deployment of web services is security. Each web service must be appropriately secured such that data is transferred securely, and requests to access data are appropriately authorized and authenticated.

We have employed OAuth2, in conjunction with EASE for web client authentication to protect web services. We have deployed an OAuth2 server as part of the Notification Backbone project WEB007, and used the Edinburgh GEL to improve the user facing authorisation page. OAuth2 provides high level access controls to individual services (for example read access on Central Authorisation service for a client). Both users and machines authorisation are applied via clients.

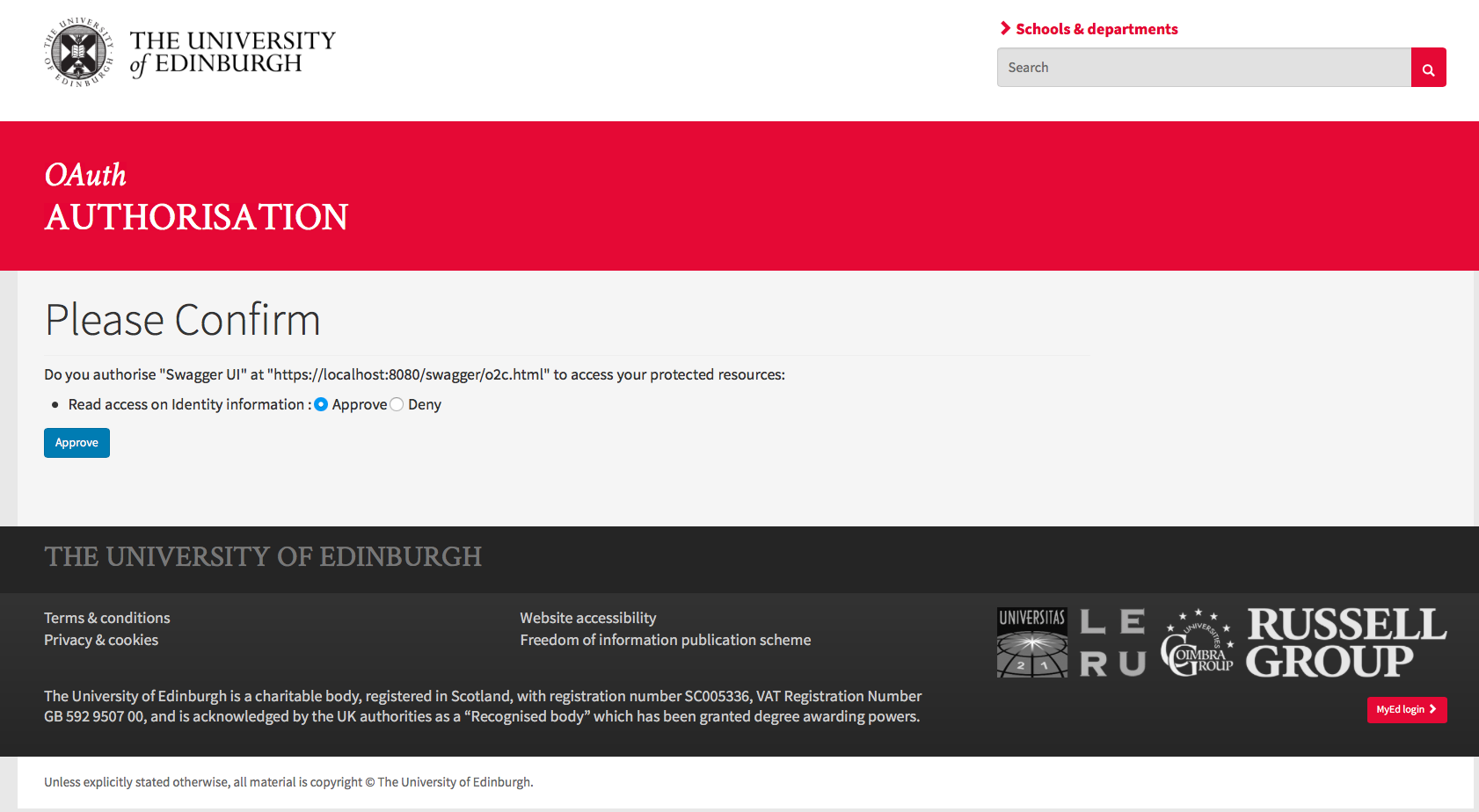


Figure 10. Example OAuth authorisation screenshot

## Summary

* **All services are HTTPS only**
* **All services use OAuth2 for authorisation**
* **All web clients authenticate via EASE**
* **Clients are set up with high level access authorisation**
* **Client (server) credentials are encrypted in the OAuth2 database using** [**bcrypt**](https://en.wikipedia.org/wiki/Bcrypt)
* **Fine grained access decisions are available in individual services using client and/or user**
* **Future developments would include** 
  + **Administrative UI for clients**
  + **Support features for viewing/revoking tokens from users and clients**
  + **More client level access information provided in the authorisation screen shown to users**

# API Documentation

Documentation for each API will mainly be covered using [Swagger](http://swagger.io/). Swagger is a leading web service documentation tool which allows the code to be the producer of documentation, and also allow developers to try out APIs without necessarily writing any code. As such it covers both the documentation and developer interaction use cases.

## Summary

* **All microservices will be documented using Swagger**

# Automation

Services have to be able to rapidly scale to meet demand, and also allow rapid deployment of changes as necessary. For these reasons it is recommended that full automation of both the infrastructure and software is employed. As Puppet rollout is still ongoing, the project made use of software automated build, testing and deployment using Bamboo.

Testing of APIs for our non-technical business partners can be difficult, as typically they are used to testing user facing applications. Ideally, typical UAT plans should be converted into automated tests such that changes to software are automatically verified against test conditions agreed with business partners. As these microservice APIs will likely be used in high volumes, testing coverage must be high and comprehensive.

## Summary

* **All microservices will have automated software deployment via Bamboo**
* **All microservices will have comprehensive test coverage**
  + **ideally also covering typical business UAT test cases**
* **Future developments would include**
  + **Automating infrastructure so that new API versions are automatically mapped in**
  + **Automating new microservices infrastructure build**

# Audit

Audit of access is important in the event of any queries regarding user level access for read or write activities. The OAuth2 server logs any access (whether successful or not) in server logs with timestamp. In addition, the OAuth2 database shows which users have access tokens.

## Summary

* **All client access is logged in OAuth server logs**
* **Created access tokens can be viewed in the OAuth database**
* **Future developments could include**
  + **Exposing access logs via a user interface**
  + **Exposing view of which user has access tokens via user interface**

# Version control

Being able to version individually deployed services is key to managing change. Any breaking changes must be managed in order to give dependent applications time to change to support the new version of the web service.

In order to support this, we will adopt [Semantic Versioning](http://semver.org/), and a stable/latest major version URL model. For example if the service is on 1.2.0, and introduces a breaking change of 2.0.1, then two URLs will be available to allow clients to select the stable version, or go with the later breaking version

* https://<service\_url>/stable
* https://<service\_url>/latest

For each service, clients can check which version they are accessing by accessing /info:

* https://<service\_url>/stable/info

Any minor or patch versions will not result in changed URLs, and changes should strive to be made in a backwards compatible way where possible.

Regarding support of the previous version:

* We will commit to supporting current version plus previous version. The definition of “support” in the context of previous version is that defects will be fixed and service availability will be maintained but the software will not be enhanced in any way.

## Summary

* **Microservices will be versioned using Semantic Versioning**
* **URLs will be provided for each service which allow stable and latest versions to be requested**

# Monitoring

As these services are likely to be used by a large number of applications and services in potentially high volumes, monitoring their health and status is key. The Spring Boot services have actuator enabled, which [provides various endpoints useful to monitoring services](http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-endpoints). In the last few years, they have also integrated much of the Netflix open source java code in as features.

Arguably the most useful is the [Hystrix](https://github.com/Netflix/Hystrix/wiki/Dashboard) stream, which allows real time monitoring of activity and success/failure rates. Hystrix Dashboard and [Turbine](https://github.com/Netflix/Turbine/wiki) allows the aggregation of multiple Hystrix streams into a single dashboard, as per the screenshot below.

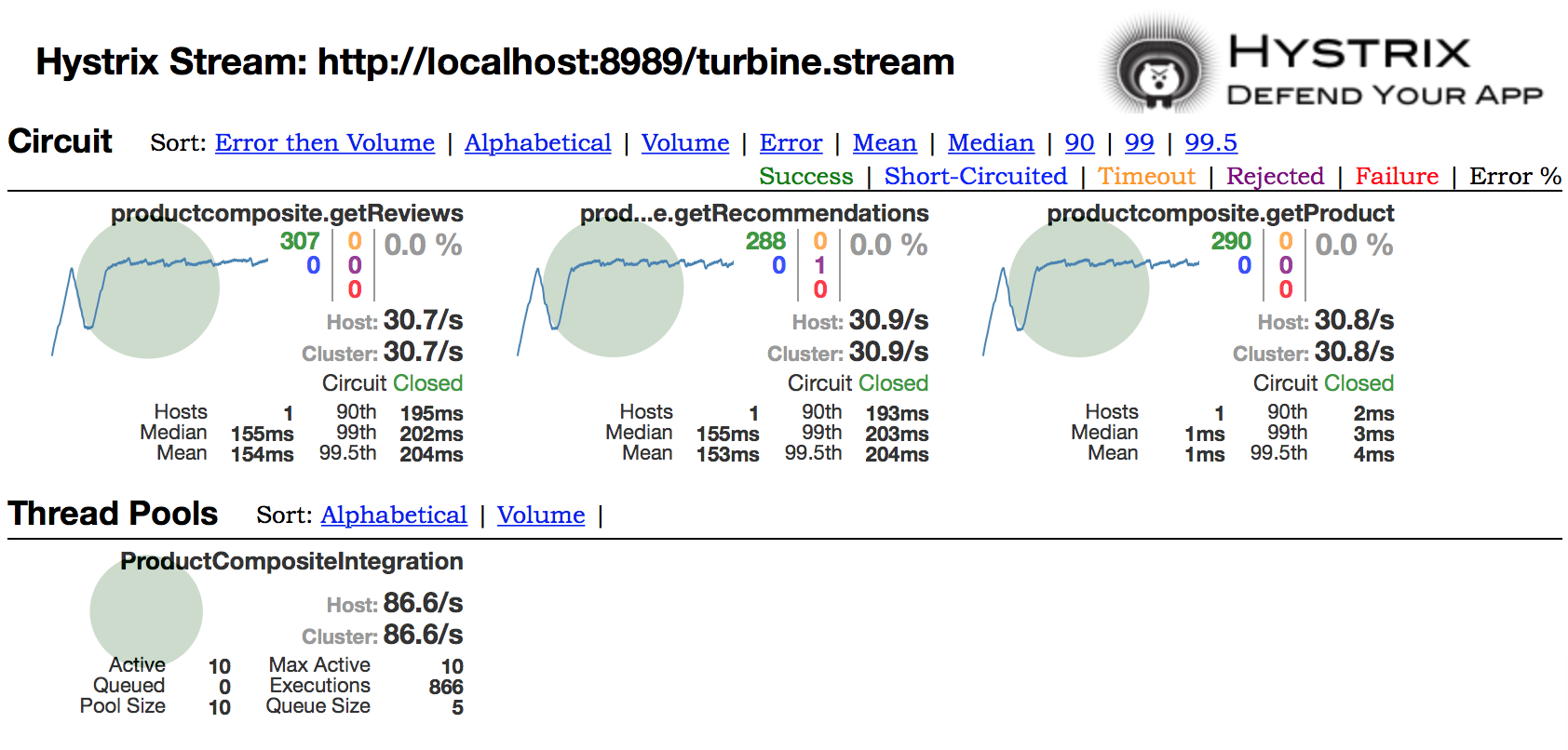


Figure 11. Example Hystrix monitoring screenshot

## Summary

* Actuator must be enabled on all Java microservices
* Future developments would include
  + Using Hystrix Dashboard in conjunction with Turbine aggregation for real-time monitoring

# Scale and Fault tolerance

Scale and fault tolerance are key to being able to run enterprise web services. Scale and fault tolerance can be managed through the use of our current load balancing technology, in that each microservice is individually health-checked and failure of the hardware load balancer to reach the healthcheck will take that server out of service.

Each of the microservices is a java container running in Tomcat, with Apache connected using mod\_jk. In order to minimise interdependencies each microservice will run in its own Tomcat.

The infrastructure as deployed would support basic resilience in our typical multi-sited setup (one server at Appleton Tower and one at Kings Buildings). Failure handling is controlled by load-balancer healthcheck on each server as per our standard software setups. Each microservice must be set up with an individual healthcheck in order to avoid failure of one service causing failing of all services.

Intelligent use of caching is also key, whether that be using Cache Headers, ETag, Caching on the server or a mixture of the three. The general rule of thumb though is to maximize throughput of requests whilst delivering the right level of real-time changes.

## Rationalisation of pre-production infrastructure

In order to streamline the development and maintenance of the microservices, it is recommended that we move from the model of beta/dev/test/live and instead adopt a pre-production/live infrastructure model. This reduces the number of environments required, and allows services to use stable/latest production API versions rather than the current situation of pre-production software integrating with other pre-production software. The pre-production microservice would allow us to run load tests and test various scenarios such as disaster recovery, failover testing and major software release testing.

Ideally no service would use pre-production microservices.

## Circuit Breakers

Hystrix also has an elegant implementation of [the Circuit Breaker design pattern](https://en.wikipedia.org/wiki/Circuit_breaker_design_pattern), allowing error states to be handled via a re-routing process with less impact, for example to queue up write requests until the database becomes available again, without the calling application waiting on the standard timeout.

## Grow and shrink demand using service discovery

Another Netflix open source component, [Eureka](https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance), provides cloud ready auto service discovery and routing. Essentially built for Amazon Web Services, it allows rapidly deployable services which register and are proxied via a central discovery service (Eureka). Eureka itself can be multi-sited, and each service which registers centrally also stores a local copy of the registry for increased resilience. Eureka itself provides a dashboard page which shows status information on which services are registered and which replication sites are up.

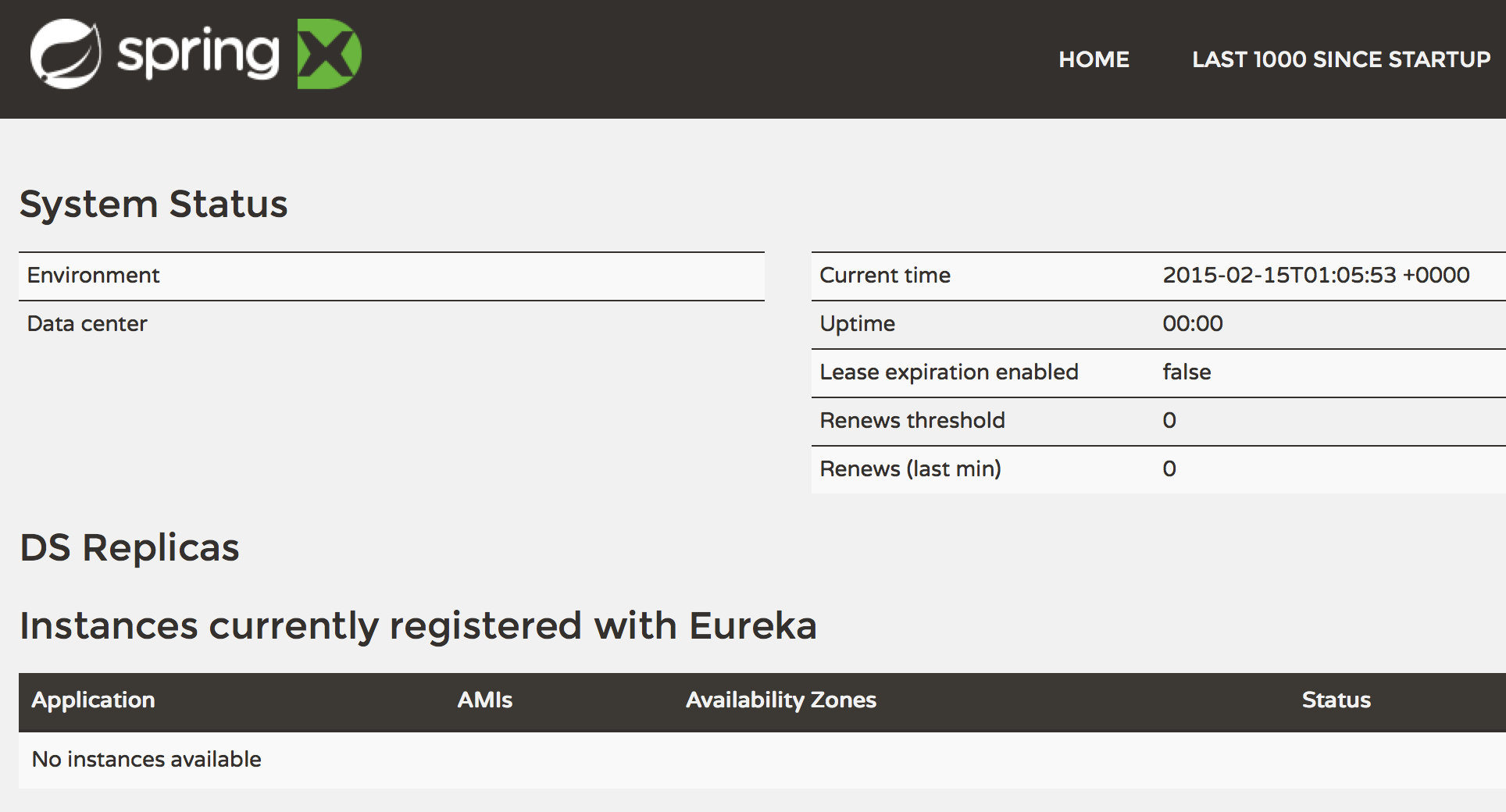


Figure 12. Eureka dashboard example

This project has done some initial proof of concept work on how Eureka could be deployed on our infrastructure to speed up both deployment of new microservices and also allow scale and shrink on demand for meeting key peak periods in the various business systems at the University.

## Summary

* **Each microservice will be deployed in a separate Tomcat container**
* **Each microservice will be individually health-checked**
* **Future developments will be**
  + **Rationalise pre-production environments**
  + **Implementing Circuit Breaker on failure to better control error states and prevent cascading failures**
  + **Exploring using a service discovery tool like Eureka to rapidly deploy and grow services**

# Service culture

If these services are to be deployed and be used by applications as the central authorised way of interacting with central corporate systems, then it is crucial to draw together all of the recommended approaches and be able to rapidly respond to failures or changes in demand.

Figure 13. Lifecycle operations

Other organisations approach this by using dedicated teams to cover the full lifecycle of the services, and often employ modern techniques such as [DevOps](https://en.wikipedia.org/wiki/DevOps) to allow the team to have the empowerment and capability to deploy changes through to production.

This approach would necessitate a dedicated team responsible for the successful running of these central services, who are empowered to react to events and spikes in demand with the necessary changes to maximize uptime and throughput.

# References

* <https://en.wikipedia.org/wiki/Bcrypt> - bcrypt encryption
* <https://en.wikipedia.org/wiki/Circuit_breaker_design_pattern> - The Circuit Breaker design pattern
* <https://en.wikipedia.org/wiki/DevOps> - Dev Ops
* <https://www.projects.ed.ac.uk/project/lmp006> - Feeding Visitor Data into Alma project
* <https://www.wiki.ed.ac.uk/display/insite/About+the+Operational+Guidance+Group> - IS Applications Operational Guidance Group
* <https://github.com/Netflix/Hystrix/wiki/Dashboard> - Netflix Hystrix
* <https://github.com/Netflix/Turbine/wiki> - Netflix Turbine
* <https://www.projects.ed.ac.uk/project/web007> - Notification Backbone project
* <https://www.projects.ed.ac.uk/project/web010> - Notifications Service project
* <http://semver.org> - Semantic Versioning
* <http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-endpoints> - Spring Boot actuator endpoints
* <http://swagger.io> - Swagger