



Henge User's Guide

Kenzan

Version 0.9.0, October 7, 2016

Henge User's Guide

1. Henge Overview	1
1.1. Enter Henge	1
1.2. The Henge Properties Model	2
1.3. Versions and Mappings	3
1.4. Henge in Architecture and App Lifecycle	4
1.4.1. Storing, Running and Getting	4
1.4.2. A Typical Lifecycle (Issues Included)	5
1.5. Back to the Neolithic	6
1.5.1. Feature Summary	7
1.5.2. Henge Architecture	7
2. Getting Started	9
2.1. Setting Up Henge	9
2.1.1. Dependencies	9
2.1.2. Build and Run Henge	9
2.1.3. Test Henge	10
2.2. REST API Reference	10
2.3. Use Cases and the Hello Properties App	10
3. Alternate Methods of Deploying Henge	11
3.1. Deploying the JAR on Tomcat	11
3.2. Deploying the Henge WAR on Tomcat	11
4. Hello Properties Application	12
4.1. Loading Properties	12
4.2. Building and Running Hello Properties	13
4.3. Changing to the Dev Environment	13
4.4. Versioning Properties	14
5. Use Case: Simple Application	15
5.1. Overview	15
5.2. Creating Properties	16
5.2.1. Create a PropertyGroup	16
5.2.2. Create a VersionSet	18
5.2.3. Create a Mapping	19
5.3. Retrieving Properties	19
5.3.1. Java Properties API	19
5.3.2. Commons Configuration	20
5.3.3. Archaius	21
5.4. Updating Properties	22
5.5. Try it Out	23
6. Use Case: Complex Application	24

6.1. Overview	24
6.2. Creating Properties	25
6.2.1. Create a PropertyGroup	26
6.2.2. Create a VersionSet	27
6.2.3. Create a Mapping	28
6.3. Retrieving Properties	29
6.3.1. Java Properties API	29
6.3.2. Commons Configuration	29
6.3.3. Archaius	30
6.4. Updating Properties.....	31
6.5. Try it Out	32
7. Domain Objects	34
7.1. Property	34
7.2. Scope	34
7.3. PropertyGroup	34
7.3.1. Scope Sets and Precedence Hierarchy.....	36
7.4. FileVersion.....	38
7.5. VersionSet	39
7.6. Mapping	40
8. Profiles	41
8.1. Activating Profiles	42
8.1.1. Specify an Application Property.....	42
8.1.2. Use a Command Line Argument.....	42
8.2. Henge Profiles.....	42
8.2.1. Default Profile	42
8.2.2. Dev Profile	44
8.2.3. Eureka Profile	44
8.2.4. Flatfile_local Profile.....	44
8.2.5. Flatfile_s3 Profile	45
8.2.6. Cassandra Profile	45
8.2.7. Metrics Profile.....	45
8.3. Profile-Specific Configurations	46
8.3.1. Noteworthy Configurations	46
9. Metrics	47
9.1. Required Services.....	47
9.2. Configuring Docker	47
9.2.1. macOS	47
9.2.2. Linux	48
9.3. Starting InfluxDB and Grafana.....	49
9.4. Starting Henge With Metrics Enabled.....	50
9.4.1. Accessing InfluxDB	50

9.4.2. Accessing Grafana	50
10. Repositories.....	51
10.1. S3 Setup	51
10.1.1. Create AWS Access Keys	51
10.1.2. Create an S3 Bucket	52
10.1.3. Set Up AWS Credentials Locally	52
10.1.4. Build and Run Henge.....	53
10.2. Cassandra Setup	53
10.2.1. Run Cassandra and Load the Schema	53
10.2.2. Build and Run Henge.....	54
10.2.3. Stopping Cassandra	54
11. Eureka Registry and Discovery Service	55
11.1. What is Eureka?	55
11.2. Running Henge With Eureka	55
11.3. About Our Eureka Implementation.....	56
11.3.1. Eureka Server Implementation	56
11.3.2. Eureka Client Implementation	58

A henge is a Neolithic structure consisting of an earthen bank and ditch encircling a flat area. Like the famous example of Stonehenge, henges might contain a ritual structure such as a circle of timbers or standing stones, and they could be accessed by walking along a processional avenue.

Henges were important places where people from the surrounding area gathered to share ideas and beliefs. Similarly, the Henge project provides an avenue to a central place where applications can go at runtime to receive values for properties, or to set values which can be used by other applications. In this way, Henge acts as a service for properties, and it provides a mechanism for the dynamic exchange of values—much like henges did long ago.



If you have questions about Henge, or how to integrate a dynamic properties service into your application architecture, feel free to drop us a line at support_henge@kenzan.com.

1. Henge Overview

From the dawn of time, developers have been storing configurations for their application in Java properties files. Whenever they pushed their application from one environment to the next (during deployment), they would request that the properties files be edited to reflect the unique values needed for each environment. After a while, even if the software could be produced as an artifact, the configs were hopelessly jumbled across the enterprise. Any exercise in resolving config values within the environment or across environments would lead to premature gray hair.

Although many enterprises are now moving to key-value stores to retrieve properties and alleviate the file chaos, most of these key-value stores lack a complete hierarchical properties model that supports the application lifecycle.

1.1. Enter Henge

At Kenzan, we grew interested in this apparent property service gap through our own experience in developing a properties store using Netflix Archaius. In particular, we noted Archaius' [open issue #132](#), which calls for creating a central properties service that separates the persistence of properties from Archaius. The question then became, how could we build something that fills this gap, is feature rich and marries well with modular application design?

Enter Henge, a REST-based property server that aims to bring order to configuration properties in a number of ways:

- Make configs first class, versioned and immutable artifacts, just like software.
- Group all possible environment values for a property in one place, preventing confusion and dismay.

- Define property groups for each of the libraries composing an application, and aggregate them into a single application config.
- Put the configs on the network where they can be updated centrally for all applications.

1.2. The Henge Properties Model

How does Henge accomplish all this? Its primary value lies in how it models properties using JSON objects. The two most basic building blocks are Property Groups and Scopes.

Property Groups contain a set of properties related to your application. Property Groups can be of type APP, representing the application itself, or of type LIB, representing a dependent library with its own set of properties. Each Property Group is immutable and has a version.

Scopes define cases where a property will have different values depending on the environment it is running in. Scopes, along with the different property values, can be defined within each Property in the Property Group.

As an example, you might have a Property Group for a SendEmail library that supports your application. It has a **supportEmail** property, which has different values for dev, staging and prod environments. In Henge, you would define it as:

Property Group: SendEmail (type=LIB, v1.0)

- **Property:** supportEmail
 - defaultValue = test@awesome.com
 - Scopes
 - env=dev
 - value = testSupport@awesome.com
 - env=stage
 - value = stageSupport@awesome.com
 - env=prod
 - value = techsupport@awesome.com
- **Property:** anotherProperty...

For supportEmail there are three scopes — **env=dev**, **env=stage**, **env=prod** — each of which have different values. The scope key is env, defining a new type of scoping. During runtime, a search is performed on Henge using one of these environment scopes to determine what property value it should retrieve for the application in that environment. Note that there is also a **defaultValue**: if none of the three environments are referenced when searching, it will use the default value.

For more complex environments, you can also define sets of scopes. Some examples of scope sets that define an environment might include:

- **env=staging**

- **env**=dev, **region**=us-west-2
- **env**=prod, **region**=us-west-2, **stack**=coolwebsiteA
- **env**=qa, **region**=us-west-2, **stack**=financePages, **hostname**=localhost

Within each property you can specify one or more scope sets and give a property value for each scope set. If a property has multiple different values based on a complex environment, a hierarchy is used to determine precedence for the various values a property could take on. For more information, see [Domain Objects](#).

1.3. Versions and Mappings

There are two other important pieces to the model that bring it all together. Once you’ve set up Property Groups or File Versions for an application, you group them in a Version Set representing the application as a whole, and create a Mapping used to search for a complete set of properties.

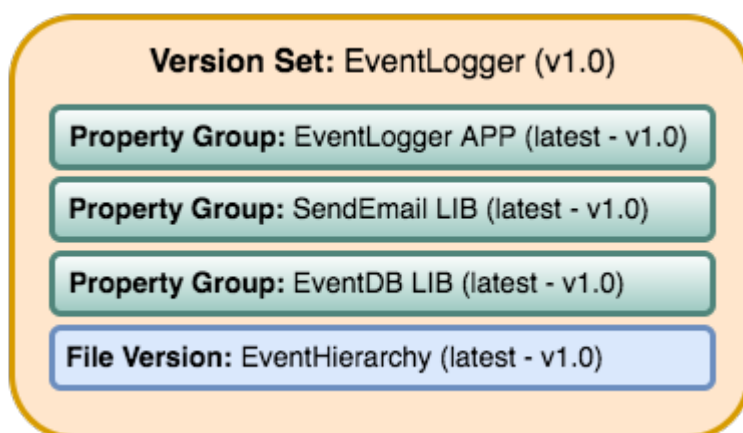
A **Version Set** references multiple Property Groups of type APP or LIB, and defines a complete set of properties for an application. A Version Set is immutable, and has its own version separate from the various Property Groups. Note that it can reference a Property Group with “latest” as the symbolic version, to always include the most recent version of the Property Group.

Mappings link a specific version of a Version Set to an application running in a specific environment (defined by the Scopes). When a search request is sent to Henge, it uses this Mapping to get all properties for the application on its environment. The Mapping itself is mutable, and can also reference the “latest” symbolic version of a Version Set.

Expanding on our previous example, let’s assume that we are building a brand new EventLogger application that logs events to a logging database. Some things to know about our application:

- The EventLogger application itself has its own set of properties.
- EventLogger uses two other libraries that have their own properties: an EventDB library for storing events, and the previously mentioned SendEmail library for sending emails.
- EventLogger relies on searching through a very long EventHierarchy for categorizing Events into logging actions. This is contained in a JSON text file.

Based on this we create Property Groups, a File Version and a Version Set for the application like so:



Within the Version Set we are referencing the “latest” versions of the Property Groups and File

Version, in this case 1.0.

EventLogger runs on dev, stage and prod environments. Many properties in the three Property Groups can have different values in these environments. As we did in the previous SendEmail example, we use dev, stage and prod scopes to assign different property values for those properties that differ in the environments.

We also create three Mappings that link the application and its environments to the “latest” EventLogger Version Set (in this case 1.0):

- EventLogger app running on env=dev → EventLogger Version Set (latest)
- EventLogger app running on env=stage → EventLogger Version Set (latest)
- EventLogger app running on env=prod → EventLogger Version Set (latest)

Each of the above Mappings represent a complete set of properties that we can search for and retrieve from Henge, depending on the environment the application is running in.

1.4. Henge in Architecture and App Lifecycle

Now that we have defined a good properties model for our EventLogger application, how does the application get those properties from Henge in our architecture, and how are properties versioned as an artifact alongside our application through its lifecycle?

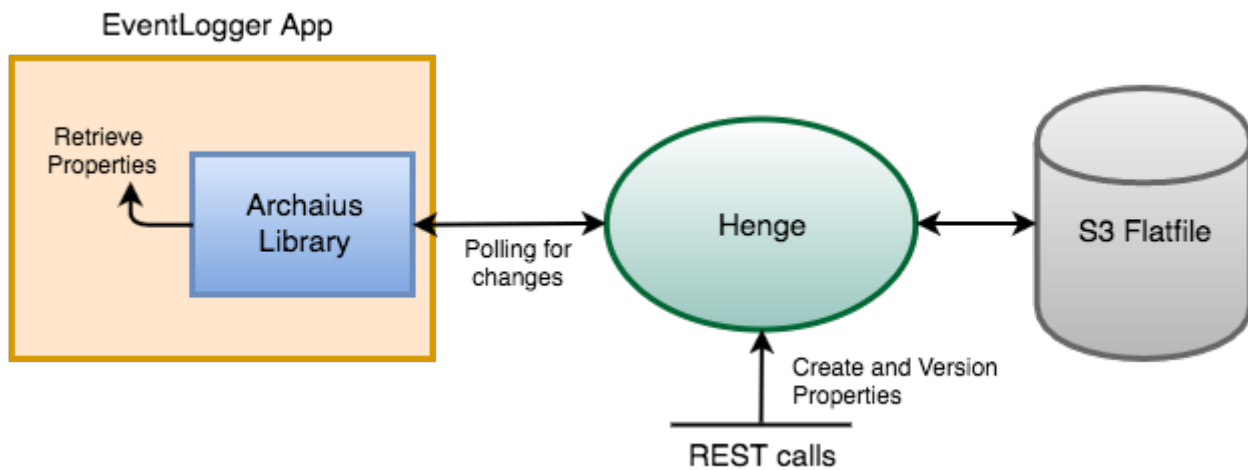
1.4.1. Storing, Running and Getting

Henge currently supports pluggable persistence on three repository types — flatfile local, flatfile on S3, and Cassandra (though Henge could easily be extended to use any data store such as RDBMS, Hibernate, etc.). Because our fictional engineering shop has ample S3 space that is backed up, we opt to run Henge using an S3 flatfile repository.

We intend to have many applications use Henge for accessing properties going forward, not just EventLogger. All of our current applications use Netflix Eureka for discovery and request load balancing, so we’d like to do the same with Henge. We make the obvious decision of running Henge with its built-in Eureka client. This is swiftly accomplished by referencing a Spring Profile on the Henge run command.

Netflix Archaius is also commonly used in many of our applications as a library to retrieve properties. We determine to use a simple integration between Archaius and Henge. On startup, EventLogger loads properties to Archaius. This is accomplished with the Archaius configuration source URL setting, which we set to the Henge search REST endpoint whose function is to retrieve property sets. Archaius polls the Henge search endpoint occasionally for updates. At runtime, EventLogger grabs individual properties as needed using Archaius libraries, and receives dynamic changes to properties as they occur. It is worth noting that any property library could be used to get individual properties, such as Java Properties classes or Apache Commons Configuration.

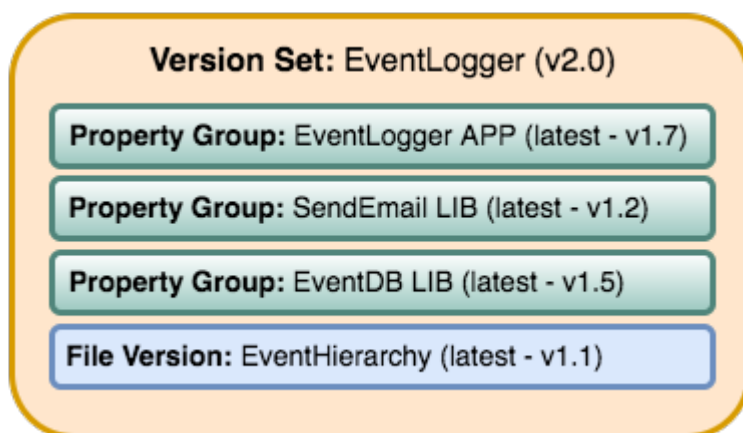
How Henge fits in our architecture now looks as follows:



1.4.2. A Typical Lifecycle (Issues Included)

During development, the EventLogger application goes through several iterative versions where property values change. Property Groups are versioned as needed. Nothing needs to change in the Version Set, as it already references the “latest” version of Property Groups. All the same, we create a new version of the Version Set to parallel the new application version - this will make it easier to perform a rollback should it ever be required (more on this later). The Mappings do not change, as they reference the latest Version Set.

When EventLogger v2.0 is ready for staging, the properties model is:

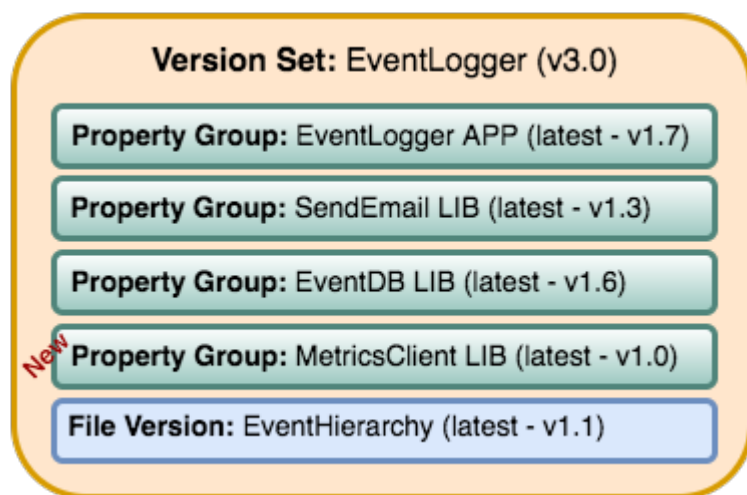


In **staging**, we’d like to test out the performance of how Henge is loading properties via Archaius’ polling, as the properties and file that are loaded are quite large. Using Henge’s built in Codahale metrics, as well as Henge’s pre-built Docker container that runs a metrics environment with InfluxDB and Grafana, Henge is started in staging and run through several test scenarios. With the captured data, it’s determined that the REST call loading properties from Henge is quite fast, but that it is best to tweak the polling in Archaius to be less frequent so that resources are more efficiently used elsewhere.

After EventLogger has gone to **production**, it’s found that the number of thread pools assigned to DB calls is too low and is causing the application to slow down. This thread pool count can be increased in EventDB properties. A single REST call is made to update the EventDB Property Group with the increased thread pool count. Because it is immutable, a new version 1.6 is created, and the Version Set automatically references this latest Property Group. Archaius picks up the change in

Henge via polling, and the application automatically starts using the new thread pool count. Happily, we didn't have to rebuild and redeploy the application to accomplish this feat.

EventLogger later goes through a **release cycle** with several enhancements for version 3.0. During the overhaul, the 3rd party SendEmail library has had a fresh update, and so its property configuration has changed. There is also a new MetricsClient library used by EventLogger to interface with a MetricDB server. The MetricsDB server is a homegrown application that has its own set of properties that will managed by Henge in a separate Version Set, and the MetricsClient will require its own set of properties. For EventLogger, we create a new Property Group for the MetricsClient library, version the SendEmail properties, and version the Version Set to reference to the new MetricsClient properties. Going to production with EventLogger v3.0, our new properties model looks as follows:



Initially everything goes smoothly when moving back into production, but then, disaster strikes. **Something goes horribly amiss** with the new Metrics server, and EventLogger comes to a staggering halt. It's going to take some time to figure out the root cause and fix it, so it's decided the best path is to roll back to EventLogger 2.0 in the meantime. The older version of the application is redeployed, and alongside it, we take care of properties by simply referring back to the older Version Set 2.0. To do so we make a single REST call to update the production Mapping — instead of referencing "latest" for the Version Set, we set it to 2.0. Later when we can find the monkey in our wrench and fix the problem, we can go back into production by re-updating the production Mapping to reference "latest."

You might ask yourself, how would my own engineering shop have handled such a rollback with its current properties architecture?

1.5. Back to the Neolithic

Having seen how Henge allows properties to be centrally versioned across the bumps in an application lifecycle, going back to using .properties files or a simple key-value store seems contrary to the immutable nature of the application itself. Just as Henges were once needed as community gathering places for sharing core ideas carved in stone, your microservice apps need a gathering place for their foundational key values. In this case, going back to the Neolithic Age is not such a bad idea after all.

To quickly download, run and test Henge, see our [Getting Started page on Github](#). Or read on for

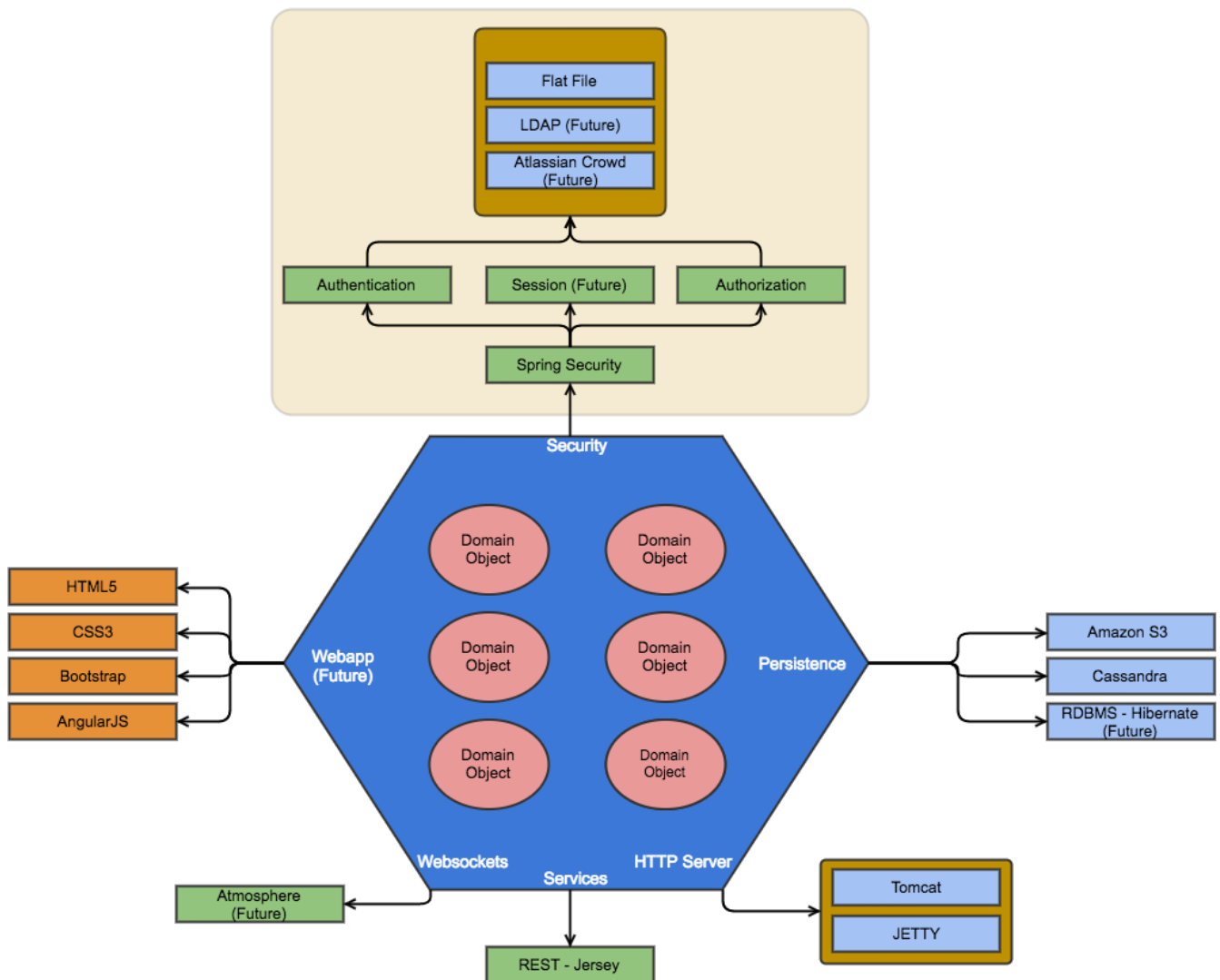
some more specifics.

1.5.1. Feature Summary

- Henge is a central dynamic property store with a REST-based interface for easy integration.
- It offers pluggable persistence — currently flatfile, flatfile on S3, and Cassandra are included as repository options, though Henge could easily be extended to use any data store (RDBMS, Hibernate, etc.).
- Henge has smart modeling of properties that accounts for different property values across deployment environments.
- The property model lets you define property groups for each of the libraries composing an application, and aggregate them into a single application config.
- Everything is immutable and includes a version. The result is a complete view of how properties have changed as your application has grown, with an application config that moves as an artifact alongside your application.
- In addition to properties, Henge stores and retrieves any related files an application might parse.
- Before properties are put into Henge, they are validated with JSR 303 bean validation.
- A Docker container is included for running Henge in a container environment.
- Henge is built with high performance in mind to support properties dynamically changing.
- The REST calls to Henge have Codahale metrics built in. We also include a pre-built Docker metrics environment that uses InfluxDB and Grafana to record and visualize those metrics.
- Henge is built with integration to Eureka, so that it can be deployed as a Eureka client that is discoverable and load balanced.
- It is easily integrated with Archaius for dynamic property retrieval, but property retrieval works with any language or library that can make a REST call to a URL.
- Spring Security is enabled to allow authorization to be set up for the various Henge REST calls. Henge uses Spring Profiles at runtime to select and modify its various features.

1.5.2. Henge Architecture

We built Henge using a suite of modern, flexible open source frameworks and libraries. The architecture below also highlights several future enhancements we are looking at — though by all means fork freely on the [Henge Github page](#).



Henge Component	Technology
Core	Domain Objects (discussed above)
	Spring Profiles
	JSR 303 bean validation
Persistence	Flatfile
	Amazon S3 (flatfile)
	Cassandra
	RDBMS / Hibernate (future)
HTTP Server	Tomcat
	Jetty
Services	REST / Jersey
Security	Spring Security — Authentication, Authorization, and Session (future)
	LDAP / Atlassian Crowd (future)

Henge Component	Technology
Web Application (future)	HTML5 / CSS3
	AngularJS / Bootstrap
WebSockets (future)	Atmosphere

2. Getting Started

Getting Started outlines setting up Henge, the API, and some Henge use cases.

2.1. Setting Up Henge

The following sequence shows how to build, run and test Henge using a local flatfile repository on both Linux and macOS.

2.1.1. Dependencies

- Java 1.8.x
- Maven

2.1.2. Build and Run Henge

1. Clone the Git repository:

```
git clone https://github.com/kenzanlabs/henge.git
```

2. Build the application by running the following in the root project folder:

```
mvn clean install
```



The build process will run several tests in the modules, so it may take some time to complete the operation. To build without the tests, you can add **-DskipTests** to the command.

3. Run the application with a local flatfile repository:

```
mvn -pl henge-service spring-boot:run
```



Though not given as an argument, here Henge is using a default Spring Profile for runtime configuration. For more information on the different profiles available and how to configure them, see [Profiles](#).

For how to run Henge using S3 flatfile or Cassandra repositories, see [Repositories](#). For instructions on how to deploy directly to a Tomcat instance as a JAR or WAR, see [Alternate Methods of Deploying Henge](#).

2.1.3. Test Henge

Adding and searching for Henge properties can be tested by running REST calls in the Postman collection files that are available within the `/documentation/demo` project folder. If you do not have it installed, [Postman can be downloaded here](#). In Postman, click **Import** and simply drag all of the collection files into the Postman Window.

To test a request in Postman, select it in the **Collections** pane, then click **Send**. You can test creating a set of properties in Henge and retrieving them by running the **HengeCollection** REST calls in the following order:

1. PropertyGroup - Create
2. FileVersion - Create
 - For this REST call, in the **Body** tab choose a text file to create the FileVersion with. The text file can be populated or blank.
3. VersionSet - Create with PropertyGroup1
4. Add VerisonSet Mapping with VersionSet1
5. Search by Application with VersionSet1

If you are using local flatfiles as a repository, you can see the files created with the above REST calls under **henge/repository**.

For more information on what the different **PropertyGroup**, **FileVersion**, **VersionSet**, and **Mapping** domain objects do, see [Domain Objects](#).

2.2. REST API Reference

To reference the complete REST API, visit the Swagger documentation at:

<http://localhost:8080/henge/swagger/index.html>

To run REST commands within the Swagger UI, a username and password is required. The default is **user/user**. These credentials can be modified in `/henge/henge-domain/src/main/resources/application.yml`.

2.3. Use Cases and the Hello Properties App

How does one use practically implement Henge in an application, and use it throughout its lifecycle? We're glad you asked. The following links give example use cases of how Henge might be utilized in an application.

- [Henge Overview](#)

- [Simple Application Use Case](#)
- [Complex Application Use Case](#)

We've additionally built [Hello Properties](#), a simple application that demonstrates using Henge properties dynamically via Archaius.

3. Alternate Methods of Deploying Henge

Henge can alternately be deployed directly onto Apache Tomcat using a JAR or WAR file. Before following the instructions, make sure you have [Henge cloned and built](#).

3.1. Deploying the JAR on Tomcat

The Henge service application uses Spring boot when it builds, and a "fat" jar is generated that contains dependencies inside the archive, including the Tomcat web server. A Tomcat instance will startup with Henge just by running the following in the root project folder:

```
java -jar henge-service/target/henge-service.jar
```

The application will now be running on <http://localhost:8080/henge> with the default Spring profile, which uses a local flatfile repository. You can change the Tomcat port by adding the option **-Dserver.port=8081** to the command line. For more information regarding configuration see [Deploying Spring Boot Applications](#).

You can also specify the various [Spring Profiles](#) for running Henge with other options. For instance, to run with flatfiles on S3 as a repository, you would use the command:

```
java -jar -Dspring.profiles.active=dev,flatfile_s3 henge-service/target/henge-service.jar
```

Note that to run flatfiles on S3, you would need to make sure your configuration is setup and Henge is built as indicated in the [Repositories](#) section.

3.2. Deploying the Henge WAR on Tomcat

1. Download Tomcat at <http://tomcat.apache.org>. Extract the archive file into a directory.
2. In a terminal, change your directory to the Tomcat directory you extracted to. Start Tomcat with:

```
bin/startup.sh
```

3. Copy the **henge/henge-war/target/henge.war** to the **<tomcat-dir>/webapps** directory.
4. Within the Tomcat root directory, restart Tomcat by executing:

```
bin/shutdown.sh
bin/startup.sh
```

5. The Henge application should now be running at <http://localhost:8080/henge>, where it displays a simple web page. The application is running with the default Spring profile which uses a local flatfile repository.

You can also specify the various [Henge Spring Profiles](#) by setting these up in JAVA_OPTS. For instance, to run with flatfiles on S3 as a repository, before starting up Tomcat you would use the command:

```
export JAVA_OPTS=-Dspring.profiles.active=dev,flatfile_s3
```

Note that to run flatfiles on S3, you would need to make sure your configuration is setup and Henge is built as indicated in the [Repositories](#) section.

4. Hello Properties Application

Hello Properties is a REST-based Java application that displays messages using Henge properties. It dynamically grabs Henge properties via a simple integration with Archaius:

- Properties are stored in the Henge repository.
- Henge properties are loaded to Archaius on start up with the `<archaius.configurationSource.additionalUrls>` URL setting in the .pom file. This setting is used to give Archaius a URL to load properties from; here we specify the `search` endpoint in Henge which finds and retrieves property groups for an application.
- Hello Properties uses Archaius libraries to retrieve individual properties during runtime and display them in two REST-based method calls.
- As properties are versioned within Henge, Archaius dynamically picks up changes by polling the Henge `search` endpoint URL every few seconds.

The demo below walks through how to start up Hello Properties, how different property values are referenced for the dev and prod environments, and how properties can be dynamically versioned.

4.1. Loading Properties

Before starting Hello Properties, you will need to load properties related to the application into Henge. Make sure you have [Henge cloned and built](#) and [Postman installed](#) before walking through the steps.

1. Within the project root directory, start Henge using a local flatfile repository:

```
mvn -pl henge-service spring-boot:run
```


2. Start Postman. Click **Import**. Drag `/henge/hello-properties/Hello%20Properties.postman_collection.json` into the Postman window.
3. Within the Hello Properties collection, click **Henge - Convenience** and click **Send**. This is a convenience method that will load properties for the application using `PropertyGroup`, `VersionSet` and `Mapping` domain objects with a JSON request.



For more information on what the domain objects do, see [Domain Objects](#).

Here we are using Postman to send REST calls, though this could also be done using curl. To view the complete REST API, see the [Henge Swagger API documentation](#).

4.2. Building and Running Hello Properties

1. Open a separate Terminal window. Within the `/henge/hello-properties` folder, build the application:

```
mvn clean install
```

2. Within the same folder, run the application using the Maven Tomcat plugin:

```
mvn tomcat7:run
```

3. You can now test the two REST endpoints of the application with Postman. Within the Hello Properties collection, run the following REST calls:
 - a. **Hello Property Name** - displays a message using the property `helloproperties.name`.
 - b. **Hello Properties Message** - displays a message that uses both `helloproperties.name` and `helloproperties.message` properties.

4.3. Changing to the Dev Environment

The previous REST calls reference the **default** values for the properties as defined in the `PropertyGroup`. Different property values are also defined for both **dev** and **prod** environment scopes in the `PropertyGroup`. The scopes exist to demonstrate running the application in different environments where property values vary.



See the **Henge - Convenience** REST call for details on what scoped values are loaded into the `PropertyGroup`. For more information on scopes, see the [Scope section of Domain Objects](#).

Continuing from the steps above, you can re-run the application with the **dev** environment scoping using a different Maven profile:

1. Stop Hello Properties by pressing **Control-C**.

2. In **/henge/hello-properties** folder, restart Hello Properties using the **dev** Maven profile.

```
mvn tomcat7:run -Denv=dev
```

3. Rerun the following Postman calls in the Hello Properties collection. Note that the calls now display different messages based on the **dev** scoping.

- a. **Hello Property Name**
- b. **Hello Properties Message**



Within the **henge/hello-properties/pom.xml**, there are Maven profiles set up to run the application using **default**, **dev** and **prod** environments. Each profile sets the Archaius system property `<archaius.configurationSource.additionalUrls>` to one of three Henge **search** endpoints with different environment scopes:

```
http://localhost:8080/henge/v1/search/helloproperties  
  
http://localhost:8080/henge/v1/search/helloproperties?scopes=env=dev  
  
http://localhost:8080/henge/v1/search/helloproperties?scopes=env=prod
```

Outside of Maven profiles, `<archaius.configurationSource.additionalUrls>` could alternately be set at the command line with:

```
mvn tomcat7:run  
-Darchaius.configurationSource.additionalUrls=Dhttp://localhost:8080/h  
enge/v1/search/helloproperties?scopes=env=dev
```

For more information on this Archaius system property, see the [Archaius wiki](#).

4.4. Versioning Properties

As mentioned, versioned changes to properties are picked up in the application dynamically via Archaius, so the application does not require a restart. Continuing from the steps above, you can test versioning the application's properties in Henge with the following steps.

1. Within the Postman Hello Properties collection, run the following REST calls that update the **PropertyGroup**, **VersionSet** and **Mapping**.
 - a. **Update Property Group** - creates a new 1.0.1 PropertyGroup with changed property values
 - b. **Update VersionSet** - creates a new 1.0.1 VersionSet that references the new PropertyGroup
 - c. **Update Mapping** - creates a new Mapping that assigns the **default** scope to the new VersionSet
 - d. **Update Mapping Dev Scope** - creates a new Mapping that assigns the **dev** scope to the new



See the `/henge/repository` folder to view the new flatfile objects that are created.

2. Wait 10 seconds, as this is the polling interval configured for Archaius. Rerun the following Postman calls in the Hello Properties collection. Note that the calls now display different messages based on the versioned property values.

- a. **Hello Property Name**

- b. **Hello Properties Message**



Within the VersionSet, you can also have it reference the PropertyGroup as "latest" instead of a hard-coded version such as "1.0.1". If the VersionSet had been set up as such, step 1 would only require versioning the `PropertyGroup`, as the `VersionSet` would then pick up the latest version of the `PropertyGroup`. The `Mapping` can also use "latest" to reference the latest `VersionSet`.

5. Use Case: Simple Application

As a developer building cloud-native applications, you need a service to host your application configurations so that you can deploy config changes on the fly without needing to rebuild your apps. Files would be nice, too! Your environment isn't too complex, just a single application consisting of about a dozen microservices with separate environments for Dev, QA, and Prod. Good news—Henge can definitely help you out.

5.1. Overview

In this use case, we have a simple application that is made up of several microservices, each of which can include shared libraries as dependencies. The microservices can be deployed in multiple environments, such as development and production. Microservices require different property values depending on the environment they are deployed in.

1. Properties are loaded into the repository using Henge:
 - Properties for each microservice are stored in a PropertyGroup of type `APP` (one for each microservice).
 - Properties for each library are stored in a PropertyGroup of type `LIB` (one for each library).
 - The PropertyGroups are added to a VersionSet (one for each microservice). The VersionSet references all properties used by the microservice as well as any included libraries. (The VersionSet can also reference FileVersions—files with parsable data.)
 - The VersionSet is mapped to application (microservice) and scope (environment). For each application, a mapping is created for each relevant environment scope (such as `dev` and `prod`).
2. At runtime, each microservice requests properties from the Henge service using a URL. The URL

includes the application name and (optionally) the environment as query parameters.

3. Henge retrieves the appropriate properties for the microservice and environment from the repository, including properties for any shared libraries or referenced files, and returns them to the application.
4. Properties can be updated with new values in Henge at any time by making a REST call. The application will receive the new properties the next time it polls Henge.

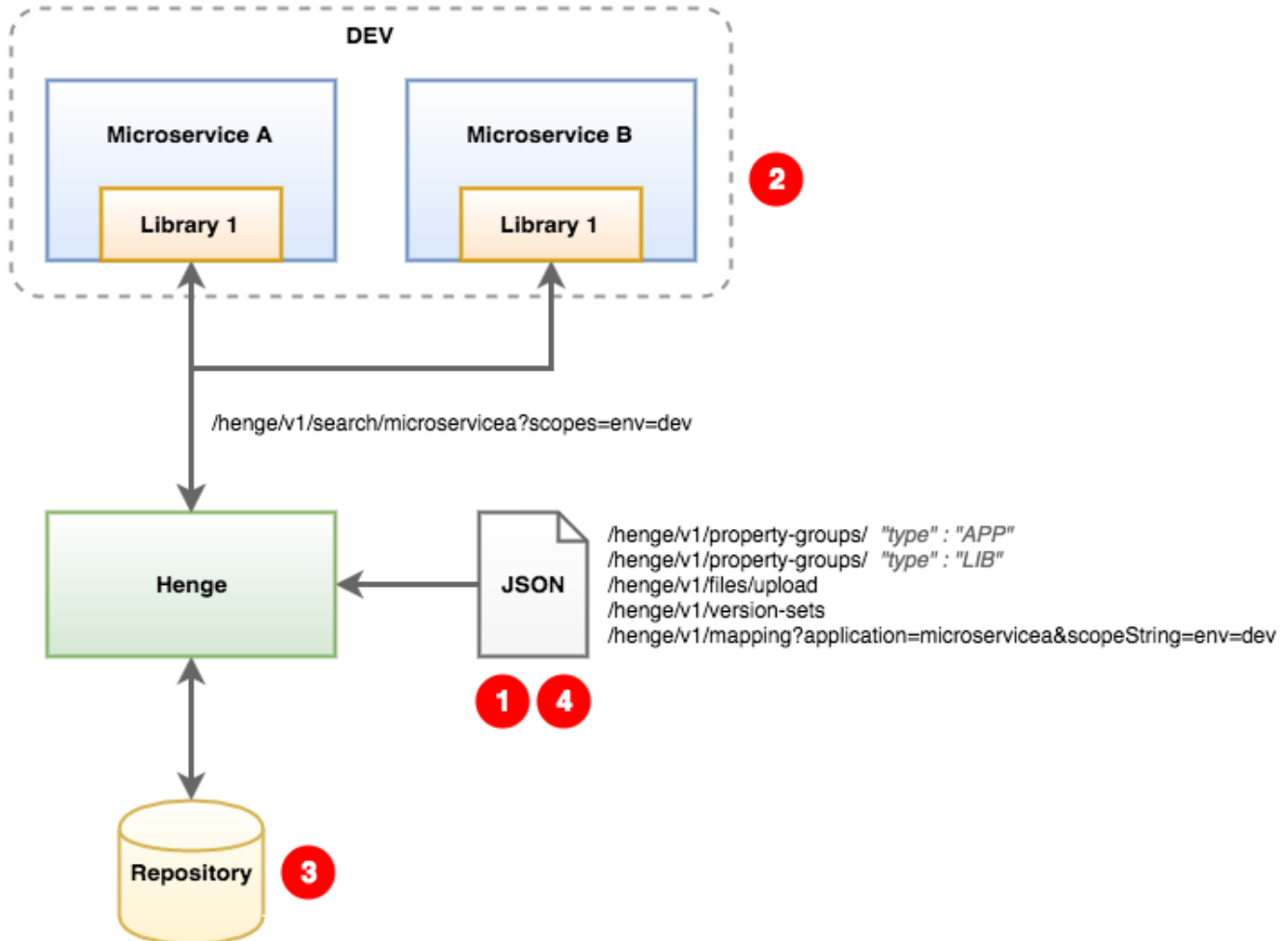


Figure 1. Use Case Overview: Simple Application

5.2. Creating Properties

Use the following requests to load application properties in the Henge repository.



Alternately, you can use the <http://localhost:8080/henge/v1/convenience/batch> endpoint to load a PropertyGroup, VersionSet, and Mapping in a single step.

5.2.1. Create a PropertyGroup

A PropertyGroup contains properties that belong to the same context, such as a single application. To create a PropertyGroup, use the <http://localhost:8080/henge/v1/property-groups/> endpoint.

The request body must be in JSON format and should contain the PropertyGroup name and version, as well as key value pairs for each property and environment (see [Domain Objects](#)). Define one

PropertyGroup of type **APP** for each microservice. In addition, define one PropertyGroup of type **LIB** for each shared library. All microservices that include the shared library will use the same library properties.



To use a file (such as a list of IP addresses) as part of the configuration, create a FileVersion using the <http://localhost:8080/henge/v1/files/upload/> endpoint.

```
{
  "name" : "SimpleAppServiceA",
  "version" : "1.0.0",
  "description" : "Simple App Microservice A",
  "type" : "APP",
  "active" : true,
  "properties" : [ {
    "name" : "simpleapp.appmessage",
    "description" : "Message property",
    "defaultValue" : "Some message",
    "propertyScopedValues": [
      {
        "value": "Hello in Development Environment (APP)",
        "scopeSet": [
          {
            "key": "env",
            "value": "dev"
          }
        ]
      }
    ],
    "value": "Hello in Production Environment (APP)",
    "scopeSet": [
      {
        "key": "env",
        "value": "prod"
      }
    ]
  }
], {
  "name" : "simpleapp.appname",
  "description" : "An APP Property",
  "defaultValue" : "Some value",
  "propertyScopedValues" : [
    {
      "value": "APP Property for Microservice A (Development Mode)",
      "scopeSet": [
        {
          "key": "env",
          "value": "dev"
        }
      ]
    }
  ]
}
```

```

    },
    {
      "value": "APP Property for Microservice A (Production Mode)",
      "scopeSet": [
        {
          "key": "env",
          "value": "prod"
        }
      ]
    }
  ]
}

```

5.2.2. Create a VersionSet

A VersionSet groups together specific versions of PropertyGroups. (A VersionSet can include FileVersions as well.) To create a VersionSet, use the <http://localhost:8080/henge/v1/version-sets> endpoint.

The request body must be in JSON format and should contain the VersionSet name and version, as well as the PropertyGroup name and version (see [Domain Objects](#)). Define one VersionSet for each microservice. The VersionSet should include the **APP** PropertyGroup for the microservice, as well as the **LIB** PropertyGroup for each shared library included in the microservice.



Instead of specifying a specific version of a PropertyGroup, you can instead specify **"version": "latest"** for version. In this case, the VersionSet will always include the most recent version of the PropertyGroup.

```

{
  "name": "SimpleAppVersionSetServiceA",
  "version": "1.0.0",
  "description": null,
  "propertyGroupReferences": [
    {
      "name": "SimpleAppServiceA",
      "version": "1.0.0"
    },
    {
      "name": "SimpleAppLibrary1",
      "version": "1.0.0"
    }
  ],
  "createdDate": null,
  "scopedPropertyValueKeys": null,
  "typeHierarchyEnabled": true
}

```

5.2.3. Create a Mapping

A Mapping entry associates a set of Scopes with a specific version of a VersionSet. To create a Mapping, use the <http://localhost:8080/henge/v1/mapping> endpoint.

Create a mapping for each microservice. You must include the **application** parameter with the URL to associate the VersionSet with the correct microservice. Optionally, you can include the **scopeString** parameter to associate the VersionSet with a particular scope (environment):

```
http://localhost:8080/henge/v1/mapping?application=microservicea&scopeString=env=dev
```

The request body must be in JSON format and should contain the VersionSet name and version (see [Domain Objects](#)).

```
{
  "name": "SimpleAppVersionSetServiceA",
  "version": "1.0.0"
}
```

5.3. Retrieving Properties

Properties can be retrieved at microservice runtime using a URL to access the [Henge search API](#). The URL must include the application (microservice) as a query parameter:

```
http://localhost:8080/henge/v1/search/microservicea
```

In the above example, the default property values are retrieved. To retrieve property values for a specific scope, the URL must also include the scope (such as the environment) as a query parameter:

```
http://localhost:8080/henge/v1/search/microservicea?scopes=env=dev
```



To retrieve environment-specific properties, the microservice must be able to send the correct query parameter for the environment that it's running in. This can be handled in several ways, for example, by creating a Maven profile for each environment. For an example of profiles in action, see [the Hello Properties demo application](#).

5.3.1. Java Properties API

To retrieve properties using Java:

```

URL url = new URL("http://localhost:8080/henge/v1/search/microservicea");
InputStream in = url.openStream();
Reader reader = new InputStreamReader(in, "UTF-8"); // for example

Properties prop = new Properties();
try {
    prop.load(reader);
} finally {
    reader.close();
}

```

5.3.2. Commons Configuration

To retrieve properties using [Apache Commons Configuration](#):

```

Parameters params = new Parameters();
// Read data from this URL
URL propertiesURL = new URL("http://localhost:8080/henge/v1/search/microservicea");

FileBasedConfigurationBuilder<FileBasedConfiguration> builder =
    new
FileBasedConfigurationBuilder<FileBasedConfiguration>(PropertiesConfiguration.class)
    .configure(params.fileBased()
        .setURL(propertiesURL));
try
{
    Configuration config = builder.getConfiguration();
    // config contains all properties read from the URL
}
catch(ConfigurationException cex)
{
    // loading of the configuration file failed
}

```

Make sure to add the following dependency to the Maven **pom.xml** file for your project:

```

<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-configuration2</artifactId>
    <version>2.1</version>
</dependency>
<dependency>
    <groupId>commons-beanutils</groupId>
    <artifactId>commons-beanutils</artifactId>
    <version>1.9.2</version>
</dependency>

```


5.3.3. Archaius

To retrieve properties using [Netflix Archaius](#):

1. Add a dependency for Archaius to the Maven **pom.xml** file for your project:

```
<dependency>
  <groupId>com.netflix.archaius</groupId>
  <artifactId>archaius-core</artifactId>
  <version>0.7.4</version>
</dependency>
```

2. Also in the **pom.xml** file, add a dependency for the Tomcat Maven plug-in and specify the following Archaius properties:

Property	Value
archaius.configurationSource.defaultFileName	Default configuration file name (use config.properties)
archaius.configurationSource.additionalUrls	Henge search URL with query parameters
archaius.fixedDelayPollingScheduler.initialDelayMills	Initial delay (in milliseconds) before reading from Henge
archaius.fixedDelayPollingScheduler.delayMilliseconds	Delay (in milliseconds) between reads from Henge

```

<plugin>
  <groupId>org.apache.tomcat.maven</groupId>
  <artifactId>tomcat7-maven-plugin</artifactId>
  <version>2.2</version>
  <configuration>
    <path>/microservicea</path>
    <port>${tomcat.port}</port>
    <useTestClasspath>>false</useTestClasspath>
    <systemProperties>

<application.home>${basedir}${file.separator}src${file.separator}test${file.separator}resources${file.separator}</application.home>
    <http.port>${tomcat.port}</http.port>

<archaius.configurationSource.defaultFileName>config.properties</archaius.configurationSource.defaultFileName>

<archaius.configurationSource.additionalUrls>http://localhost:8080/henge/v1/search/microservicea</archaius.configurationSource.additionalUrls>

<archaius.fixedDelayPollingScheduler.initialDelayMills>1000</archaius.fixedDelayPollingScheduler.initialDelayMills>

<archaius.fixedDelayPollingScheduler.delayMills>10000</archaius.fixedDelayPollingScheduler.delayMills>
    </systemProperties>
  </configuration>
</plugin>

```



For an example of using Archaius, see the **pom.xml** file for the **hello-properties** application, located in the **/hello-properties** directory.

5.4. Updating Properties

Property values can be changed at any time, even while the application is running, by updating a `PropertyGroup` as well as the relevant `VersionSets` and `Mappings`. You don't have to rebuild or redeploy the application.

To update a `PropertyGroup`, use the `/v1/property-groups/{propertyName}` endpoint. The request body is the same as when creating a `PropertyGroup`, but make sure to increment the version number in addition to updating property values.



To update a `FileVersion`, use the `http://localhost:8080/henge/v1/files/update/` endpoint.

After updating a `PropertyGroup`, you need to update the `VersionSet` and `Mapping` for any applications that should use the updated properties:

- Update the VersionSets to refer to the new PropertyGroup version using the `/v1/version-sets/{versionSetName}` endpoint.
- Update the Mappings to refer to the new VersionSet version using the `http://localhost:8080/henge/v1/mapping/` endpoint.

5.5. Try it Out

Use the following [Postman](#) requests to load and retrieve properties using a URL.



Before trying the examples below, make sure the Henge server is up and running (see [Getting Started](#)).

First, import the Henge Use Case Simple App Postman collection:

1. In Postman, click **Import**, and then click **Choose Files**.
2. Select the **HengeUseCaseSimpleApp.postman_collection.json** file (located in the `/documentation/demo/` directory), and then click **Open**.

Next, send the following requests (located under the **HengeUseCaseSimpleApp** collection in Postman):

Request	Description
1. Create APP PropertyGroup - Microservice A	Creates an APP PropertyGroup named SimpleAppServiceA .
2. Create APP PropertyGroup - Microservice B	Creates an APP PropertyGroup named SimpleAppServiceB .
3. Create LIB PropertyGroup - Library 1	Creates a LIB PropertyGroup named SimpleAppLibrary1 .
4. Create VersionSet - Microservice A	Creates a VersionSet named SimpleAppVersionSetServiceA that contains the PropertyGroups SimpleAppServiceA and SimpleAppLibrary1 .
5. Create VersionSet - Microservice B	Creates a VersionSet named SimpleAppVersionSetServiceB that contains the PropertyGroups SimpleAppServiceB and SimpleAppLibrary1 .
6. Create Mapping for Dev - Microservice A	Associates the VersionSet SimpleAppVersionSetServiceA with the application named microservicea and the dev environment.
7. Create Mapping for Dev - Microservice B	Associates the VersionSet SimpleAppVersionSetServiceB with the application named microserviceb and the dev environment.
8. Retrieve Properties for Dev - Microservice A	Returns the properties associated with the application microservicea and the dev environment. The properties returned include APP properties for Microservice A as well as LIB properties for Library 1.

Request	Description
9. Retrieve Properties for Dev - Microservice B	Returns the properties associated with the application microserviceb and the dev environment. The properties returned include APP properties for Microservice B as well as LIB properties for Library 1.
10. Update APP PropertyGroup - Microservice A	Updates the APP PropertyGroup named SimpleAppServiceA with new property values and increments to v1.0.1.
11. Update VersionSet - Microservice A	Updates the VersionSet named SimpleAppVersionSetServiceA to reference v1.0.1 of the PropertyGroup SimpleAppServiceA and increments to v1.0.1.
12. Update Mapping for Dev - Microservice A	Associates the VersionSet SimpleAppVersionSetServiceA v1.0.1 with the application named microservicea and the dev environment.
13. Retrieve Properties for Dev - Microservice A	Returns the properties associated with the application microservicea and the dev environment. Note that the v1.0.1 properties are returned.



There are also Postman requests for mapping and retrieving properties for the **prod** environment. Send these requests if you'd like to test getting properties for another environment.

6. Use Case: Complex Application

As an architect designing cloud application platforms, you need a service to that helps you manage configurations as versioned artifacts the same as software artifacts. Your environment is complex, with multiple application stacks needing different versions of configs in different environments and sub-environments. Don't worry—Henge has you covered, and can handle complex deployments like yours.

6.1. Overview

In this use case, we have a complex application that is made up of numerous microservices, each of which can include shared libraries as dependencies. The microservices can be deployed in multiple environments, such as development and production, and in different AWS regions. In addition, microservices can be arranged into different product stacks within an environment or region. Microservices may require different property values depending on the environment or stack they are deployed in.

1. Properties are loaded into the repository using Henge:

- Properties for each microservice are stored in a PropertyGroup of type **APP** (one for each microservice).
- Properties for each library are stored in a PropertyGroup of type **LIB** (one for each library).
- The PropertyGroups are added to a VersionSet (one for each microservice). The VersionSet references all properties used by the microservice as well as any included libraries. (The

VersionSet can also reference FileVersions—files with parsable data.)

- The VersionSet is mapped to application (microservice) and scope (stack, region, and environment). For each application, a mapping is created for each relevant environment (such as **dev** and **prod**), region (such as **us-west-2**), and stack (such as **supersite** and **powerpage**).
2. At runtime, each microservice requests properties from the Henge service using a URL. The URL includes the application name and (optionally) the environment, region, and stack as query parameters.
 3. Henge retrieves the appropriate properties for the microservice, environment, region, and stack from the repository, including properties for any shared libraries or referenced files, and returns them to the application.
 4. Properties can be updated with new values in Henge at any time by making a REST call. The application will receive the new properties the next time it polls Henge. VersionSets can be maintained as versioned artifacts by creating a new VersionSet version for each new microservice version. This makes it easy to support different versions of the same microservice running in different stacks.

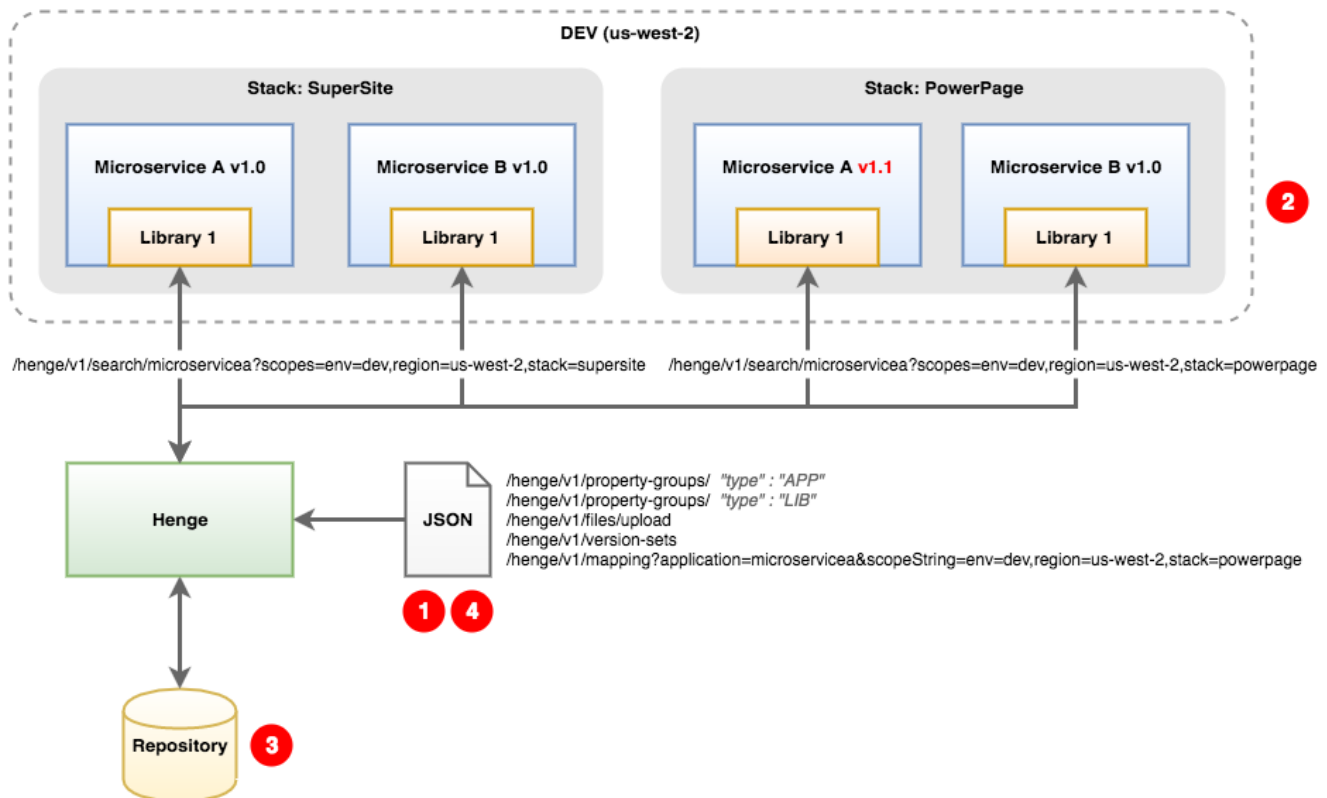


Figure 2. Use Case Overview: Complex Application

6.2. Creating Properties

Use the following requests to load application properties in the Henge repository.



Alternately, you can use the <http://localhost:8080/henge/v1/convenience/batch> endpoint to load a PropertyGroup, VersionSet, and Mapping in a single step.

6.2.1. Create a PropertyGroup

A PropertyGroup contains properties that belong to the same context, such as a single application. To create a PropertyGroup, use the <http://localhost:8080/henge/v1/property-groups/> endpoint.

The request body must be in JSON format and should contain the PropertyGroup name and version, as well as key value pairs for each property, environment, region, and stack (see [Domain Objects](#)). Define one PropertyGroup of type **APP** for each microservice. In addition, define one PropertyGroup of type **LIB** for each shared library. All microservices that include the shared library will use the same library properties.



To use a file (such as a list of IP addresses) as part of the configuration, create a FileVersion using the <http://localhost:8080/henge/v1/files/upload/> endpoint.

```
{
  "name" : "ComplexAppServiceA",
  "version" : "1.0.0",
  "description" : "Complex App Microservice A",
  "type" : "APP",
  "active": true,
  "properties" : [ {
    "name" : "complexapp.appmessage",
    "description" : "Message property",
    "defaultValue" : "Some message",
    "propertyScopedValues": [
      {
        "value": "This is Microservice A in SuperSite (Development Mode)",
        "scopeSet": [
          {
            "key": "env",
            "value": "dev"
          },
          {
            "key": "region",
            "value": "us-west-2"
          },
          {
            "key": "stack",
            "value": "supersite"
          }
        ]
      },
      {
        "value": "This is Microservice A in SuperSite (Production Mode)",
        "scopeSet": [
          {
            "key": "env",
            "value": "prod"
          }
        ]
      }
    ]
  }
]
```

```

        "key": "region",
        "value": "us-west-2"
    },
    {
        "key": "stack",
        "value": "supersite"
    }
]
},
{
    "value": "This is Microservice A in PowerPage (Development Mode)",
    "scopeSet": [
        {
            "key": "env",
            "value": "dev"
        },
        {
            "key": "region",
            "value": "us-west-2"
        },
        {
            "key": "stack",
            "value": "powerpage"
        }
    ]
},
{
    "value": "This is Microservice A in PowerPage (Production Mode)",
    "scopeSet": [
        {
            "key": "env",
            "value": "prod"
        },
        {
            "key": "region",
            "value": "us-west-2"
        },
        {
            "key": "stack",
            "value": "powerpage"
        }
    ]
}
]
}
]
}

```

6.2.2. Create a VersionSet

A VersionSet groups together specific versions of PropertyGroups. (A VersionSet can include

FileVersions as well.) To create a VersionSet, use the <http://localhost:8080/henge/v1/version-sets> endpoint.

The request body must be in JSON format and should contain the VersionSet name and version, as well as the PropertyGroup name and version (see [Domain Objects](#)). Define one VersionSet for each microservice. The VersionSet should include the **APP** PropertyGroup for the microservice, as well as the **LIB** PropertyGroup for each shared library included in the microservice.



Instead of specifying a specific version of a PropertyGroup, you can instead specify `"version": "latest"` for version. In this case, the VersionSet will always include the most recent version of the PropertyGroup.

```
{
  "name": "ComplexAppVersionSetServiceA",
  "version": "1.0.0",
  "description": null,
  "propertyGroupReferences": [
    {
      "name": "ComplexAppServiceA",
      "version": "1.0.0"
    },
    {
      "name": "ComplexAppLibrary1",
      "version": "1.0.0"
    }
  ],
  "createdDate": null,
  "scopedPropertyValueKeys": null,
  "typeHierarchyEnabled": true
}
```

6.2.3. Create a Mapping

A Mapping entry associates a set of Scopes with a specific version of a VersionSet. To create a Mapping, use the <http://localhost:8080/henge/v1/mapping> endpoint.

Create a mapping for each microservice. You must include the **application** parameter with the URL to associate the VersionSet with the correct microservice. Optionally, you can include the **scopeString** parameter to associate the VersionSet with a particular set of scopes (environment, region, and stack):

```
http://localhost:8080/henge/v1/mapping?application=microservicea&scopeString=env=dev,r
egion=us-west-2,stack=powerpage
```

The request body must be in JSON format and should contain the VersionSet name and version (see [Domain Objects](#)).


```
{
  "name": "ComplexAppVersionSetServiceA",
  "version": "1.0.0"
}
```

6.3. Retrieving Properties

Properties can be retrieved at microservice runtime using a URL to access the [Henge search API](#). The URL must include the application (microservice) as a query parameter:

```
http://localhost:8080/henge/v1/search/microservicea
```

In the above example, the default property values are retrieved. To retrieve property values for a specific scope, the URL must also include the scope set (such as the environment, region, and stack) as a query parameter:

```
http://localhost:8080/henge/v1/search/microservicea?scopes=env=dev,region=us-west-2,stack=powerpage
```



To retrieve scope-specific properties, the microservice must be able to send the correct query parameter for the environment, region, and stack that it's running in. This can be handled in several ways, for example, by creating a Maven profile for each environment. For an example of profiles in action, see [the Hello Properties demo application](#).

6.3.1. Java Properties API

To retrieve properties using Java:

```
URL url = new URL("http://localhost:8080/henge/v1/search/microservicea");
InputStream in = url.openStream();
Reader reader = new InputStreamReader(in, "UTF-8"); // for example

Properties prop = new Properties();
try {
    prop.load(reader);
} finally {
    reader.close();
}
```

6.3.2. Commons Configuration

To retrieve properties using [Apache Commons Configuration](#):

```

Parameters params = new Parameters();
// Read data from this URL
URL propertiesURL = new URL("http://localhost:8080/henge/v1/search/microservicea");

FileBasedConfigurationBuilder<FileBasedConfiguration> builder =
    new
FileBasedConfigurationBuilder<FileBasedConfiguration>(PropertiesConfiguration.class)
    .configure(params.fileBased()
        .setURL(propertiesURL));
try
{
    Configuration config = builder.getConfiguration();
    // config contains all properties read from the URL
}
catch(ConfigurationException cex)
{
    // loading of the configuration file failed
}

```

Make sure to add the following dependency to the Maven **pom.xml** file for your project:

```

<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-configuration2</artifactId>
  <version>2.1</version>
</dependency>
<dependency>
  <groupId>commons-beanutils</groupId>
  <artifactId>commons-beanutils</artifactId>
  <version>1.9.2</version>
</dependency>

```

6.3.3. Archaius

To retrieve properties using [Netflix Archaius](#):

1. Add a dependency for Archaius to the Maven **pom.xml** file for your project:

```

<dependency>
  <groupId>com.netflix.archaius</groupId>
  <artifactId>archaius-core</artifactId>
  <version>0.7.4</version>
</dependency>

```

2. Also in the **pom.xml** file, add a dependency for the Tomcat Maven plug-in and specify the following Archaius properties:

Property	Value
archaius.configurationSource.defaultFileName	Default configuration file name (use config.properties)
archaius.configurationSource.additionalUrls	Henge search URL with query parameters
archaius.fixedDelayPollingScheduler.initialDelayMills	Initial delay (in milliseconds) before reading from Henge
archaius.fixedDelayPollingScheduler.delayMills	Delay (in milliseconds) between reads from Henge

```

<plugin>
  <groupId>org.apache.tomcat.maven</groupId>
  <artifactId>tomcat7-maven-plugin</artifactId>
  <version>2.2</version>
  <configuration>
    <path>/microservicea</path>
    <port>${tomcat.port}</port>
    <useTestClasspath>>false</useTestClasspath>
    <systemProperties>

    <application.home>${basedir}${file.separator}src${file.separator}test${file.separator}resources${file.separator}</application.home>
      <http.port>${tomcat.port}</http.port>

    <archaius.configurationSource.defaultFileName>config.properties</archaius.configurationSource.defaultFileName>

    <archaius.configurationSource.additionalUrls>http://localhost:8080/henge/v1/search/microservicea</archaius.configurationSource.additionalUrls>

    <archaius.fixedDelayPollingScheduler.initialDelayMills>1000</archaius.fixedDelayPollingScheduler.initialDelayMills>

    <archaius.fixedDelayPollingScheduler.delayMills>10000</archaius.fixedDelayPollingScheduler.delayMills>
      </systemProperties>
    </configuration>
  </plugin>

```



For an example of using Archaius, see the **pom.xml** file for the **hello-properties** application, located in the **/hello-properties** directory.

6.4. Updating Properties

Property values can be changed at any time, even while the application is running, by updating a `PropertyGroup` as well as the relevant `VersionSets` and `Mappings`. You don't have to rebuild or

redeploy the application until it's time to deploy a new version of the application.

To update a PropertyGroup, use the `/v1/property-groups/{propertyName}` endpoint. The request body is the same as when creating a PropertyGroup, but make sure to increment the version number in addition to updating property values.



To update a FileVersion, use the `http://localhost:8080/henge/v1/files/update/` endpoint.

After updating a PropertyGroup, you need to update the VersionSet and Mapping for any applications that should use the updated properties:

- Update the VersionSets to refer to the new PropertyGroup version using the `/v1/version-sets/{versionSetName}` endpoint.
- Update the Mappings to refer to the new VersionSet version using the `http://localhost:8080/henge/v1/mapping/` endpoint.

For example, say you have two different stacks (Web sites) called SuperSite and PowerPage. You deploy a new version of a microservice (v1.1) into your PowerPage stack in the dev environment in the AWS us-west-2 region. First you update the microservice's APP PropertyGroup with new property values and increment it to v1.1. Next you update the microservice's VersionSet to v1.1 and reference the v1.1 PropertyGroup. Finally, you update the Mapping to associate the v1.1 VersionSet with the microservice and the scope set of dev (environment), us-west-2 (region), and PowerPage (stack).

Now, the microservice v1.1 running in the PowerPage stack uses the updated v1.1 properties. But the older microservice v1.0 running in the SuperSite stack still uses the v1.0 properties. That's the benefit of versioned configurations!

6.5. Try it Out

Use the following [Postman](#) requests to load and retrieve properties using a URL.



Before trying the examples below, make sure the Henge server is up and running (see [Getting Started](#)).

First, import the Henge Use Case Simple App Postman collection:

1. In Postman, click **Import**, and then click **Choose Files**.
2. Select the **HengeUseCaseComplexApp.postman_collection.json** file (located in the `/documentation/demo/` directory), and then click **Open**.

Next, send the following requests (located under the **HengeUseCaseComplexApp** collection in Postman):

Request	Description
1. Create APP PropertyGroup - Microservice A	Creates an APP PropertyGroup named ComplexAppServiceA .
2. Create LIB PropertyGroup - Library 1	Creates a LIB PropertyGroup named ComplexAppLibrary1 .
3. Create a VersionSet - Microservice A	Creates a VersionSet named ComplexAppVersionSetServiceA that contains the PropertyGroups ComplexAppServiceA and ComplexAppLibrary1 .
4. Create Mapping for Microservice A - SuperSite Dev	Associates the VersionSet ComplexAppVersionSetServiceA with the application named microservicea and the dev environment, us-west-2 region, and supersite stack.
5. Create Mapping for Microservice A - PowerPage Dev	Associates the VersionSet ComplexAppVersionSetServiceA with the application named microservicea and the dev environment, us-west-2 region, and powerpage stack.
6. Retrieve Properties for Microservice A - SuperSite Dev	Returns the properties associated with the application microservicea and the dev environment, us-west-2 region, and supersite stack. The properties returned include APP properties for Microservice A as well as LIB properties for Library 1.
7. Retrieve Properties for Microservice A - PowerPage Dev	Returns the properties associated with the application microservicea and the dev environment, us-west-2 region, and powerpage stack. The properties returned include APP properties for Microservice A as well as LIB properties for Library 1.
8. Update APP PropertyGroup - Microservice A	Updates the APP PropertyGroup named ComplexAppServiceA with new property values and increments to v1.1.
9. Update VersionSet - Microservice A	Updates the VersionSet named ComplexAppVersionSetServiceA to reference v1.1 of PropertyGroup ComplexAppServiceA and increments to v1.1.
10. Update Mapping for Microservice A - PowerPage Dev	Associates the VersionSet ComplexAppVersionSetServiceA v1.1 with the application named microservicea and the dev environment, us-west-2 region, and powerpage stack.
11. Retrieve Properties for Microservice A - PowerPage Dev	Returns the properties associated with the application microservicea and the dev environment, us-west-2 region, and powerpage stack. Note that the v1.1 properties are returned.
12. Retrieve Properties for Microservice A - SuperSite Dev	Returns the properties associated with the application microservicea and the dev environment, us-west-2 region, and supersite stack. Note that the v1.0 properties are returned.



There are also Postman requests for mapping and retrieving properties for Microservice B. Send these requests if you'd like to test getting properties for another application.

7. Domain Objects

Henge relies on a number of domain objects to encapsulate the data needed to provide its property configuration service. Each domain object has a Java class inside the application code, and a JSON representation for REST calls. Most of the domain objects have a name and version attribute associated with them, and they are also immutable. All update methods applied to the objects will create a new entry in the repository.

7.1. Property

A **Property** is a configuration variable. It has the following attributes:

- **name**: the name of the property.
- **description**: the description of the property.
- **defaultValue**: the default value of the property when no **Scope** is defined.
- **propertyScopedValues**: a set of alternative values for the property, one for each **Scope**.

Properties are not referenced directly in JSON format. Instead, several properties are defined inside a **PropertyGroup**. See **PropertyGroup** below for a JSON example.

7.2. Scope

A **Scope** is a key-value pair used to specify different values for properties in a given scenario. A **Scope** is useful when a configuration variable should have different values depending on the environment the application is run in. For example, you could have a scope key as **env** and possible values of **dev**, **test**, and **prod** which are particular environments the application runs in. This would define three instances of **Scope** that can be used to set different values for the properties in those environments.

See the **Property Group** section for examples of using one or multiple scopes, as well as how to set up a precedence hierarchy for scopes.

Scopes are used when setting up the **Mapping** domain object and when retrieving a set of properties using the **search** endpoint. For more information, see the **Mapping** section.

7.3. PropertyGroup

A **PropertyGroup** groups properties that belong to the same context and always go together. It has the following attributes:

- **name**
- **version**
- **createdBy**
- **createdDate**
- **description**

- **type**: the type of the **PropertyGroup**. It can be one of two values: **APP** or **LIB**. **APP** should be used for your application, and **LIB** for properties that belong to some library or dependency of your project. This distinction is used when evaluating property values. If the same **Property** name exists in both an **APP** and **LIB PropertyGroup** (there is a name conflict), **APP** type properties override **LIB** properties.
- **isActive**: true indicates the **PropertyGroup** is active.
- **properties**: the set of **Properties** the **PropertyGroup** contains.

An example of a **PropertyGroup** might be a configuration for a database connection. For this **PropertyGroup** you need **Property** values for **host**, **port**, **username** and **password**. Let's assume you have **dev**, **test** and **prod** environments where the values for these properties may be different. You would define a **PropertyGroup** with four properties. Each property would have three **propertyScopedValues** and a **defaultValue**. The following is a JSON representation of this example:

```
{
  "name": "dbconfig",
  "version": "1.0.0",
  "description": "Example database configuration",
  "type": "APP",
  "active": true,
  "properties": [{
    "name": "dbhost",
    "description": "Ip of database host",
    "defaultValue": "localhost", # ③
    "propertyScopedValues": [{
      "key": "env=dev", # ①
      "value": "127.0.0.1" # ②
    }, {
      "key": "env=test",
      "value": "192.168.0.10"
    }, {
      "key": "env=production",
      "value": "192.168.0.20"
    }
  ]
}, {
  "name": "dbport",
  "description": "Port of database",
  "defaultValue": "4321" # ④
}, {
  "name": "dbuser",
  "description": "Username for connecting to database",
  "defaultValue": "user",
  "propertyScopedValues": [{
    "key": "env=dev",
    "value": "devuser"
  }, {
    "key": "env=test",
    "value": "testuser"
  }, {
```

```

        "key": "env=prod",
        "value": "produser"
    }]
}, {
    "name": "dbpasswd",
    "description": "Password for connecting to database",
    "defaultValue": "1234",
    "propertyScopedValues": [{
        "key": "env=dev",
        "value": "1234"
    }, {
        "key": "env=test",
        "value": "1234"
    }, {
        "key": "env=prod",
        "value": "b40df5b0d4d173a554b1030c0f453dac"
    }]
}]
}

```

- ① The key of the **propertyScopedValue** is a **Scope** whose key is **env** and value is **dev**.
- ② **127.0.0.1** is the value of the **dbhost** property when the **Scope** is **env=dev**.
- ③ The **dbhost** property uses the **defaultValue** of **localhost** if no **Scope** is specified when the search is performed on Henge.
- ④ For **dbport**, there are no **propertyScopedValues** and only a **defaultValue** because the **dbport** is the same for all environments.

7.3.1. Scope Sets and Precedence Hierarchy

The previous example shows a single scoping of different values based on dev, test and prod environments. But how are more complex environments set up?

Scopes can be defined in **scope sets** that represent a specific environment the application runs in that has multiple scopes. Some examples of scope sets include:

- **env=staging**
- **env=dev, region=us-west-2,**
- **env=prod, region=us-west-2, stack=coolwebsiteA**
- **env=qa, region=us-west-2, stack=financePages, hostname=localhost**

Each **Property** can define one or more scope sets, where a specific property value is designated for each scope set.

As an example, let's say you have an application that has a URL that can be different based on the environment, the region, and a hostname. Within the **PropertyGroup** you would define various **scopeSets** under **propertyScopedValues** as shown in the following JSON example.


```

{
  "active": true,
  "name": "EventLoggerAPP",
  "properties": [
    {
      "defaultValue": "www.website.com",
      "name": "eventlogger.url",
      "propertyScopedValues": [
        {
          "scopeSet": [ ❶
            {
              "key": "env",
              "value": "dev"
            }
          ],
          "value": "www.dev-website.com" ❷
        },
        {
          "scopeSet": [
            {
              "key": "env",
              "value": "dev"
            },
            {
              "key": "region",
              "value": "us-west-2"
            }
          ],
          "value": "us-west-2.dev-website.com" ❸
        },
        {
          "scopeSet": [
            {
              "key": "hostname",
              "value": "localhost"
            }
          ],
          "value": "localhost:8080" ❹
        }
      ]
    }
  ],
  "type": "APP",
  "version": "1.0.3"
}

```

- ❶ Instead of defining different key/values as in the previous example, a **scopeSet** is used to define a set of scopes that have a specific value.
- ❷ If deployed in the **dev** environment, the URL is **www.dev-website.com**.

- ③ If deployed in the **dev** environment and in the **us-west-2** region, the URL is **us-west-2.dev-website.com**.
- ④ If deployed in any environment or region and **hostname=localhost**, the URL is **localhost:8080**. This acts as an override for testing.

Note that in #4, the stipulation is *any* environment or region. For example, let's say the application is deployed in:

- env=dev
- region=us-west-2
- hostname=localhost

When searching for this property using this same set of scopings, the return value would always be **localhost:8080**. Yet how does Henge know to use precedence for the URL value **localhost:8080**, instead of returning **www.dev-website.com** or **us-west-2.dev-website.com**?

Henge uses a hierarchy of scopings for resolving which scoped value takes precedence over another. The resolution order is defined in the Spring profile **henge/henge-domain/src/main/resources/application.yml** by the **scope.precedence.configuration** property. Its default hierarchy is:

env;env+region;env+region+stack;hostname;application

Sets of scopes are defined using the plus + symbol, and are placed in precedence order. Precedence weight is greater as you move from left to right, so **hostname** has a greater priority than **env** or **env+region**.

In our example, the higher precedence of the **hostname=localhost** scope is the reason why the URL resolves to **localhost:8080** in any environment or region. To see a working Postman collection with the above example, see

henge/documentation/demo/HengeScopeHierarchy.postman_collection.json.

The **scope.precedence.configuration** property can be configured in any way depending on the scopes you've defined and your environment needs.



If you are using a new scope set in a **PropertyGroup**, make sure that the scope set is represented in **scope.precedence.configuration**, as the hierarchy also generally defines which scope sets can be used when searching. Note that the **application** scope is always added on the end as the last precedence.

7.4. FileVersion

Henge can also be used to store configuration files that cannot be translated to a **.properties** file. For example, let's assume that some part of your application needs to store a long list of geolocations that are fixed, but could change between instances of the system. Being a long list, you would probably not want to store it as property values. It would be more appropriate to store the geolocations in a text file. Henge stores files like these in an entity called **FileVersion**, which has the following attributes:

- **name**
- **version**
- **description**
- **content**: a byte array containing the contents of the file.
- **fileName**: the source file name.
- **createdBy**
- **createdDate**
- **modifiedBy**
- **modifiedDate**

Here is an example of **FileVersion** in JSON format:

```
{
  "name": "GeoLoc",
  "version": "1.0.0",
  "description": "List of GeoLocations of Mountains",
  "content": "TW91bnQgRWxiZXJ0LCBDb2xvcmFkb3wzOS4xMTc4NTEyfC0xMDYuNDQ1MTU5OQpNb3VudCBNaXRjaGVsbCwgTm9ydGggQ2Fyb2xpbmF8MzUuNzY0OTYxMnwtODIuMjY1MTEKTW91bnQgUmFpbmllciwgV2FzaGluZ3Rvbnw0Ni44NTI5MTI5fC0xMjEuNzYwNDQ0Ng==",
  "filename": "GeoLoc.txt",
  "createdBy": "rdaugherty",
  "createdDate": "2016-08-22T09:44:51.58",
  "modifiedBy": "rdaugherty",
  "modifiedDate": "2016-08-22T09:44:51.58"
}
```

7.5. VersionSet

A **VersionSet** groups specific versions of **PropertyGroups** and **FileVersions**, wrapping up all the properties needed for a given application. The **VersionSet** itself has a version number associated with it. The reasoning behind this is that applications using Henge are versioned and the corresponding configuration must be able to keep up with the application's evolution, having different versions that can coexist and attend to multiple releases of the application it serves.

When a **VersionSet** is returned by a query to Henge, it is processed and all the properties contained in its **PropertyGroups** are evaluated considering the **Scopes** given in the query. A **.properties** file is then generated and sent back to the client.

VersionSets have the following attributes:

- **name**
- **version**
- **createdBy**
- **createdDate**

- **description**
- **propertyGroupReferences**: a set of references to **PropertyGroups**. A reference contains only **name** and **version**, which are sufficient to identify a **PropertyGroup**.
- **fileVersionReference**: a set of references to **FileVersions** (similar to above).

Here is an example of a **VersionSet** in JSON format:

```
{
  "name": "ExampleVersionSet",
  "version": "1.0.0",
  "description": "Example of a VersionSet",
  "fileVersionReferences": [{
    "name": "configfile",
    "version": "1.0.0"
  }],
  "propertyGroupReferences": [{
    "name": "dbconfig",
    "version": "1.0.0"
  }, {
    "name": "some-other-property-group",
    "version": "latest" # ①
  }]
}
```

① A **VersionSet** can point to a symbolic version (latest), in which case it will always point to the highest version number for that **PropertyGroup**.

7.6. Mapping

After creating **PropertyGroups** and **VersionSets**, the configuration properties defined in them are not yet available to clients. They only become live after creating a **Mapping** entry, which binds an application and a set of **Scopes** to a specific **VersionSet**.

A **Mapping** entry is created with REST parameters that include an **application** (required), a **scopeString** that defines the set of scopes (optional), and a **body** that indicates the name and version of the **VersionSet**. So for instance, you could create a **Mapping** with this URL request indicating the application and scopes:

```
http://localhost:8080/henge/v1/mapping?application=loginApp&scopeString=env=prod,stack=financeWebStack
```

And this body indicating the **VersionSet**:

```
{
  "name": "VersionSet-1",
  "version": "1.0.0" ❶
}
```

❶ The version could also be specified as "latest", in which case it will always point to the highest version number for that **VersionSet**.

The **Mapping** is stored as a JSON object where the **application** itself is a scope, as shown:

```
{
  "{ \"scopeSet\": [{ \"key\": \"env\", \"value\": \"prod\" }, { \"key\": \"stack\", \"value\": \"financeWebStack\" }, { \"key\": \"application\", \"value\": \"loginApp\" } ] }" : {
    "name" : "VersionSet-1",
    "version" : "1.0.0"
  }
}
```

Using the **search** endpoint, a complete set of properties is retrieved by providing an **application** (required) and a set of scopes (optional). The **Mapping** is looked up to provide the specific **VersionSet**, which is then converted to a **.properties** file. For instance, if the above **Mapping** was in place, you could then retrieve a complete set of properties using the following **search** call:

```
http://localhost:8080/henge/v1/search/loginApp?scopes=env=prod,stack=financeWebStack
```

8. Profiles

Sometimes an application needs to behave differently depending on its context. For example, the application might need to use a different IP address or port number when running in a development environment compared to a production environment.

Spring Profiles offer a mechanism for switching configurations or altering how functionality is implemented at runtime. With profiles, you can indicate that parts of your application are available only in a particular context. When a profile is active, the parts of the application that are associated with the profile are available, while parts associated with non-active profiles are not available.

To associate classes with a particular profile, use the **@Profile** annotation. You can use **@Profile** with any class that uses the **@Component** or **@Configuration** annotation. In the following example, the **ProductionConfiguration** class is associated with the **dev** profile:

```
@Configuration
@Profile("dev")
public class ProductionConfiguration {

    // ...

}
```

By assigning different implementations of a common interface to different profiles, it's possible to choose which implementation you want to use at runtime. For example, Henge supports storing properties in a local flat file, an S3 flat file, or a Cassandra database. With profiles, you can choose which storage option to use based on the environment the application is running in.



To learn more about Spring Profiles, see the [Spring Boot documentation](#).

8.1. Activating Profiles

Configurations or implementations associated with a profile are only available when the profile is active. There are two ways to activate profiles: specify an application property or use a command line argument.

8.1.1. Specify an Application Property

You can activate profiles using the `spring.profiles.active` property. Specify the active profiles with this property in the `application.yml` file. For example:

```
spring.profiles.active=dev,cassandra
```

8.1.2. Use a Command Line Argument

You can activate profiles when starting the application using a command line argument. Use the switch `--spring.profiles.active=profile` or the switch `-Dspring.profiles.active=profile`. This will override the value of the `spring.profiles.active` variable. For example:

```
mvn -pl property-service spring-boot:run -Dspring.profiles.active=dev,flatfile_local
```

8.2. Henge Profiles

Henge uses a number of pre-defined profiles. See the sections below for information about each profile and its configuration variables.

8.2.1. Default Profile

The **default** profile is different from other profiles in that you don't have to activate it. Instead, it

represents the default configuration contained in the **application.yml** file. Use the **default** profile to set values for running the application in a production environment on AWS.

Table 1. Default Profile Variables: `/henge-domain/src/main/resources/application.yml`

Variable	Definition
<code>spring.application.name</code>	The application name. Eureka uses this as the name of the client service.
<code>spring.profiles.active</code>	The list of active profiles.
<code>spring.jersey.type</code>	Specifies whether to use Jersey as a Filter or a Servlet. Default value: filter
<code>multipart.max-file-size</code>	The upper bound on the size of FileVersion objects that can be stored in Henge.
<code>server.contextPath</code>	The context root of the web application.
<code>swagger.api.version</code>	The version of the Swagger API.
<code>swagger.schemes</code>	Comma separated list of accepted protocols. Example: http,https
<code>swagger.base.path</code>	Base path for the Swagger APIs.
<code>swagger.resource.package</code>	Package from which Swagger must scan for endpoints.
<code>swagger.scan</code>	Specifies whether Swagger should scan for endpoints. Recommended value: true
<code>swagger.domain</code>	The domain where the Swagger UI resides in the production environment.
<code>swagger.port</code>	The port where the Swagger UI resides in the production environment.
<code>cache.expiration.minutes</code>	The time the cache lives after each write to it.
<code>text.encoding</code>	The encoding used to convert bytes to text (and vice versa) throughout all repository implementations.
<code>scope.precedence.configuration</code>	Defines an order, from most generic to most specific, of scope keys. This changes the way the search behaves when the key given does not have an exact match.
<code>scope.application.name.key</code>	String that represents the application name in the scope keys.
<code>eureka.client.serviceUrl.defaultZone</code>	The URL used by the discovery client (this application) to register on the Eureka server.
<code>cassandra.host</code>	The host for the Cassandra database server (production environment).
<code>cassandra.port</code>	The port for the Cassandra database server (production environment).
<code>security.user</code>	User name for authentication when executing REST requests that are not GET.

Variable	Definition
security.password	Password for authentication when executing REST requests that are not GET.

8.2.2. Dev Profile

Use the **dev** profile to set configuration variables for running Henge in a local environment.

Table 2. Dev Profile Variables: `/henge-domain/src/main/resources/application-dev.yml`

Variable	Definition
swagger.domain	The domain where the Swagger UI resides in the production environment.
swagger.port	The port where the Swagger UI resides in the production environment.
eureka.client.serviceUrl.defaultZone	The URL used by the discovery client (this application) to register on the Eureka server.
cassandra.host	The host for the Cassandra database server (production environment).
cassandra.port	The port for the Cassandra database server (production environment).

8.2.3. Eureka Profile

Use the **eureka** profile to enable Eureka as the discovery service. Enabling Eureka provides support for running clustered instances of Henge. See [Eureka Registry and Discovery Service](#) for more information.



The **eureka** profile does not use any configuration variables. It is used in the **EurekaClientConfig** class located in: `/henge-service/src/main/java/com/kenzan/henge/config/EurekaClientConfig.java`

8.2.4. Flatfile_local Profile

Use the **flatfile_local** profile to enable local storage of the flatfile implementation of the repositories.

Table 3. Flatfile_local Profile Variables: `/henge-repository/src/main/resources/application-flatfile_local.yml`

Variable	Definition
repository.location	The folder, relative to the user home, where the application data is stored.
versionset.mapping.file.name	The name of the file where the mapping from Scope objects to VersionSet objects is stored.

8.2.5. Flatfile_s3 Profile

Use the **flatfile_s3** profile to enable Amazon S3 storage of the flatfile implementation of the repositories.

Table 4. Flatfile_s3 Profile Variables: /henge-repository/src/main/resources/application-flatfile_s3.yml

Variable	Definition
repository.bucket.name	The name of the Amazon S3 bucket where the application data is stored.
amazon.profile.name	The name of the Amazon AWS profile, inside the credentials file, associated with Henge.
versionset.mapping.file.name	The name of the file where the mapping from Scope objects to VersionSet objects will be stored.

8.2.6. Cassandra Profile

Use the **cassandra** profile to enable Cassandra database implementation of the repositories.

Table 5. Cassandra Profile Variables: /henge-repository/src/main/resources/application-cassandra.yml

Variable	Definition
cassandra.keyspace	The name for the Cassandra keyspace. The name is defined here and is used by all environments.

8.2.7. Metrics Profile

Use the **metrics** profile to enable the publishing of Henge metrics. When the **metrics** profile is active, Henge publishes metrics data to the InfluxDB database. You can then display the metrics on a Grafana dashboard, with real-time charts that update every five seconds by default. The charts include information about load and latency, as well as the application endpoints. See [Metrics](#) for more information.

Table 6. Metrics Profile Variables: /henge-service/src/main/resources/application-metrics.yml

Variable	Definition
metrics.influx.host	IP address of the InfluxDB database where metrics are stored.
metrics.influx.port	Port number of the InfluxDB database.
metrics.influx.user	User name for connecting to the InfluxDB database.
metrics.influx.password	Password for connecting to the InfluxDB database.
metrics.influx.database	Name of the InfluxDB database.
metrics.influx.periodIn Seconds	The period for publishing metrics. For example, a value of 5 means that metrics are sent to InfluxDB every 5 seconds.

8.3. Profile-Specific Configurations

For each profile, there are specific configuration variables you can set, as described in the section above. Edit the values for these variables in the **src/main/resources/application-{profile}.yml** configuration file in each module.

Most (but not all) modules in the project include a configuration file. We attempted to place each configuration file where it made the most sense. For example, the **application-flatfile_local.yml** configuration file is located in the **henge-repository** module.



See the tables in the section above for the location of each profile-specific configuration file.

8.3.1. Noteworthy Configurations

Below are some configuration variables worthy of special attention.

Flatfile_local Profile

repository.location

The folder where the application data is stored.



The **repository.location** folder is relative to the user home folder.



The Maven build process automatically creates the folder defined for **repository.location**.

Flatfile_s3 Profile

amazon.profile.name

The name of the Amazon AWS profile associated with Henge. The default value is **henge**. The specified profile must be present inside your **~/.aws/credentials** file. For example:

```
[henge]
aws_access_key_id={key}
aws_secret_access_key={secret_key}
```



Make sure the credentials given have read and write access to the S3 bucket where the data is stored.

repository.bucket.name

The name of the Amazon S3 bucket where the application data is stored.

9. Metrics

Think of metrics as a way to add instrumentation to your application. Just as instruments like gauges and tachometers give you useful information while driving your car, metrics let you see relevant usage and performance data for your application while it's running.

Henge offers metrics for each service endpoint that allow you to monitor their execution. We rely on [Dropwizard Metrics](#) and its integration with [Spring Actuator](#) to provide metrics values. These values are stored in an [Influx DB](#) database, and they are made available for visualization on a [Grafana](#) dashboard.

9.1. Required Services

To run Henge and publish metrics, the InfluxDB and Grafana servers must be running. In addition, Henge needs to be configured to publish metrics data to the InfluxDB server.

To simplify this process, and to provide an environment for the purposes of load test analysis, we created Docker images. With Docker, you can quickly start up the metrics environment without having to individually install and configure the required services.



The purpose of the Docker images is to provide a metrics environment for temporary use, such as test analysis. This is why the metrics services are configured to run on the same host as Henge.

If you need to run the metrics environment for longer periods, it's best to configure the metrics environment on another host. In this case, you need to change the class **MetricsConfig.java** (in the **henge-service** module) to point to the new InfluxDB host.

9.2. Configuring Docker

You must install and configure Docker to run the Docker containers for metrics. Follow the steps below for your operating system.

9.2.1. macOS

For systems running macOS, install and configure the Docker Toolbox. This runs Docker in a virtual machine that you can access using the Docker Quickstart Terminal.

1. Open the file **/henge-service/src/main/resources/application-metrics.yml** in a text editor (like **TextEdit**) and change the value for **metrics.influxdb.host** to **192.168.99.100**:

```
# Metrics - InfluxDB Configuration
metrics.influx:
  host: 192.168.99.100 ①
  port: 8086
  user: admin
  password: admin
  database: henge
  periodInSeconds: 5
```

① Change this value to **192.168.99.100**, and then save and close the file.

2. Download the [Docker Toolbox package for Mac](#).
3. Double-click the package to run the installer, and then follow the [installation instructions](#).
4. When you reach the **Quick Start** step, select the **Docker Quickstart Terminal** option. This will launch the Docker terminal.



In the future, if you need to launch the Docker terminal, open the **Docker** folder in your **Applications** folder, and then double-click **Docker Quickstart Terminal**.

5. Configure required environment variables. To do this, enter the following commands in the Docker terminal (press **<Enter>** after each command):

```
export DOCKER_CERT_PATH=~/.docker/machine/certs
export DOCKER_HOST=tcp://192.168.99.100:2376
export DOCKER_TLS_VERIFY=1
```

Docker is now ready to go. Keep the terminal window open, and continue with [Starting InfluxDB and Grafana](#) below.

9.2.2. Linux

For systems running Linux, install and configure the Docker Engine. This runs Docker natively on your system (rather than inside a virtual machine, like on macOS).

1. Open the file **/henge-service/src/main/resources/application-metrics.yml** in a text editor and make sure the value for **metrics.influxdb.host** is **127.0.0.1**:

```
# Metrics - InfluxDB Configuration
metrics.influx:
  host: 127.0.0.1 ①
  port: 8086
  user: admin
  password: admin
  database: henge
  periodInSeconds: 5
```

① If necessary, change this value to **127.0.0.1**, and then save and close the file.

2. Install Docker by following the [installation instructions](#) for your Linux distribution.
3. Configure the Docker daemon to enable communication with the Maven plug-in. To do this, open the file **/etc/default/docker** in a text editor and add the following line:

```
DOCKER_OPTS="-H tcp://127.0.0.1:4041 -H unix:///var/run/docker.sock" ①
```

① The IP address must be the same as for **localhost**, but the port can be modified as needed. Using the default port of **2376** caused issues, so here we are using port **4041** instead.

4. Restart Docker using the following command:

```
sudo restart docker
```

5. Set the **DOCKER_HOST** environment variable using the following command:

```
export DOCKER_HOST=tcp://127.0.0.1:4041 ①
```

① Use the same port as in Step 3 above.

Docker is now ready to go. Keep the terminal window open, and continue with [Starting InfluxDB and Grafana](#) below.

9.3. Starting InfluxDB and Grafana

To start the metrics environment, change to the root directory of the project, and then use the following command:

```
mvn -pl henge-docker -P metrics
```



When running this command, make sure you are in the same terminal window you used to export the environment variables.

9.4. Starting Henge With Metrics Enabled

To start Henge and enable publishing of metrics to InfluxDB, change to the root directory of the project, and then use the following command:

```
mvn -pl henge-service spring-boot:run
-Dspring.profiles.active=dev,flatfile_local,metrics
```



When running this command, make sure you are in the same terminal window you used to export the environment variables.



The `-Dspring.profiles.active` switch is used to specify the Spring profiles to activate when running Henge. You can specify different profiles as needed. However, to publish metrics values, the `metrics` profile must be active. See [Profiles](#) for more information.

9.4.1. Accessing InfluxDB

Metrics are published to InfluxDB each time an endpoint is used. To access the InfluxDB web interface, use the appropriate URL for your operating system:

macOS

<http://192.168.99.100:8083>

Linux

<http://127.0.0.1:8083>

9.4.2. Accessing Grafana

The Grafana dashboard loads metrics values from InfluxDB and makes them available for visualization. To access the Grafana dashboard, use the appropriate URL for your operating system:

macOS

<http://192.168.99.100:3000>

Linux

<http://127.0.0.1:3000>

You can view the dashboard by clicking **Home** in the top banner, and then clicking **Henge**.



Figure 3. Henge Dashboard

10. Repositories

In addition to running Henge using local flatfiles, it can be configured to store properties in an AWS S3 repository or Cassandra database. The following outlines how to build and run Henge using either repository. The instructions apply to both Linux and macOS.

10.1. S3 Setup

The following AWS setup is required to build and run Henge using an S3 repository.

10.1.1. Create AWS Access Keys

If you haven't already done so, generate AWS root access keys to allow Henge to authenticate with AWS.

1. In your EC2 Management Console, click your user name, and then click **Security Credentials**.

2. Expand **Access Keys**, then click **Create New Access Key**.
3. Click **Download Key File** to download the access key file.



The access key file is a CSV file that contains values for two keys: **AWSAccessKeyId** and **AWSSecretKey**. You will need these two key values in a later step.

For more information, see the [AWS documentation on Security Credentials](#).

10.1.2. Create an S3 Bucket

1. Within the S3 Dashboard, click **Create Bucket** to create a new S3 Bucket as a repository for Henge flat files.
2. Give the bucket a unique name, such as **henge-repository-[OrganizationName]**, then click **Create**.

For more information, see the [AWS documentation on creating an S3 Bucket](#).

10.1.3. Set Up AWS Credentials Locally

You'll want to add the AWS access keys you previously downloaded to the **credentials** file in the `~/.aws` directory. This will allow Henge access to the S3 bucket.

1. Install the AWS CLI. See the [documentation on installing the AWS CLI](#).
2. The access keys should be added to a **henge** profile within the **credentials** file. Using the AWS CLI, the keys can be added to the file with the following command:

```
aws configure --profile henge
```

At the prompts, enter the following:

```
AWS Access Key ID [None]: {Your Access Key Here}
AWS Secret Access Key [None]: {Your Secret Access Key Here}
Default region name [None]: {Appropriate region here}
Default output format [None]: <press ENTER>
```

3. You should now have an `~/.aws/credentials` file that looks similar to the following:

```
[henge]
aws_access_key_id = {Your Access Key Here}
aws_secret_access_key = {Your Secret Access Key Here}
```


10.1.4. Build and Run Henge

1. Clone the Git repository:

```
git clone https://github.com/kenzanlabs/henge.git
```

2. Open up `/henge/henge-repository/src/main/resources/application-flatfile_s3.yml` in a text editor. Change the property `repository.bucket.name` to the S3 bucket you created, similar to:

```
repository.bucket.name: henge-repository-[OrganizationName]
```

3. Build the application by running the following in the root project folder:

```
mvn clean install -P S3-tests
```



With the **-P S3-Tests** option, the build process will run all tests including S3 tests on the modules, so it may take some time to complete the operation. To build without any tests, you can use the **-DskipTests** option instead.

4. Run Henge with S3:

```
mvn -pl henge-service spring-boot:run -Dspring.profiles.active=dev,flatfile_s3
```

As shown in the run command, Henge uses Spring Profiles for runtime configuration. For more information on the different profiles available and how to configure them, see [Profiles](#).

10.2. Cassandra Setup

10.2.1. Run Cassandra and Load the Schema

1. Download Cassandra from <http://cassandra.apache.org/download/>.
2. Install Cassandra by extracting it to a folder of your choice.



Cassandra's **cqlsh** needs to have Python 2 installed and accessible in your system's path. It is **only compatible with version 2 of Python** and will not work with the version 3. Python can be downloaded from <https://www.python.org/downloads/>.

3. Start Cassandra by running:

```
{cassandra_install_folder}/bin/cassandra
```

4. You will need to use **cqlsh** to run the Henge schema creation script located in **henge-repository/src/cassandra/cql/load.cql**. Do the following:
 - a. Change your directory to the Henge project root folder.
 - b. Run **cqlsh** to execute schema creation:

```
{cassandra_install_folder}/bin/cqlsh -f henge-repository/src/cassandra/cql/load.cql
```

More information on **cqlsh** is available at the [datastax cqlsh reference](#).

10.2.2. Build and Run Henge

1. Clone the Git repository:

```
git clone https://github.com/kenzanlabs/henge.git
```

2. Build the application by running the following in the root project folder:

```
mvn clean install
```



The build process will run several tests in the modules, so it may take some time to complete the operation. To build without the tests, you can add **-DskipTests** to the command.

3. Start Henge using the Cassandra repository:

```
mvn -pl henge-service spring-boot:run -Dspring.profiles.active=dev,cassandra
```



Henge defaults to using port 9042 to send Cassandra data (the default port setup in Cassandra). If Cassandra is not a new install, make sure that within the **conf/cassandra.yaml** file the **native_transport_port** is set to 9042, and **start_native_transport** is set to true.

10.2.3. Stopping Cassandra

When you are through testing, you can stop Henge by pressing **Control+C**.

To stop Cassandra, do the following:

1. Enter the following command:

```
ps auxx | grep cassandra
```

Look at the output from the command and note the first 3–5 digit number that appears in the output. This is the process ID for Cassandra.

2. Enter the following command where pid is the process ID you found (you'll be prompted for your administrator password):

```
sudo kill pid
```

11. Eureka Registry and Discovery Service

Henge includes the built-in capability to run Eureka server and register itself as a Eureka client via Spring Boot. This provides an easy way to implement load balancing and integrate with other services that use Eureka. This section outlines what Eureka does, how to run Henge with Eureka, and the steps we took to implement Henge with Eureka.

11.1. What is Eureka?

Eureka is an AWS discovery service that allows middle-tier services to register and connect with one another. Its primary purpose is load balancing by ensuring service requests are always routed to an available instance. Eureka includes a server and a client. The server acts as a REST-based registry for clients that runs in a Java servlet container. The Eureka client is a Java library for interfacing with the Eureka server. When a client launches, it sends instance metadata to the Eureka service and notifies the server when it's ready to receive traffic. The client has a built-in round-robin load balancer and periodically sends information to the Eureka server indicating which instances are still functioning.

More information on Eureka is available at the [AWS Eureka wiki](#).

11.2. Running Henge With Eureka

1. To start the Eureka server using Spring Boot, run the following in the Henge root project folder:

```
mvn -pl eureka-server spring-boot:run
```

This command will start the Eureka server already configured. You don't need to download Eureka separately.

2. In order for Henge to register itself as a client to the Eureka server, run the project using the **eureka** profile:

```
mvn -pl henge-service spring-boot:run
-Dspring.profiles.active=dev,flatfile_local,eureka
```

3. By accessing the Spring Boot Eureka server page at <http://localhost:8761/>, you can see information about the client instances currently registered to the Eureka server.

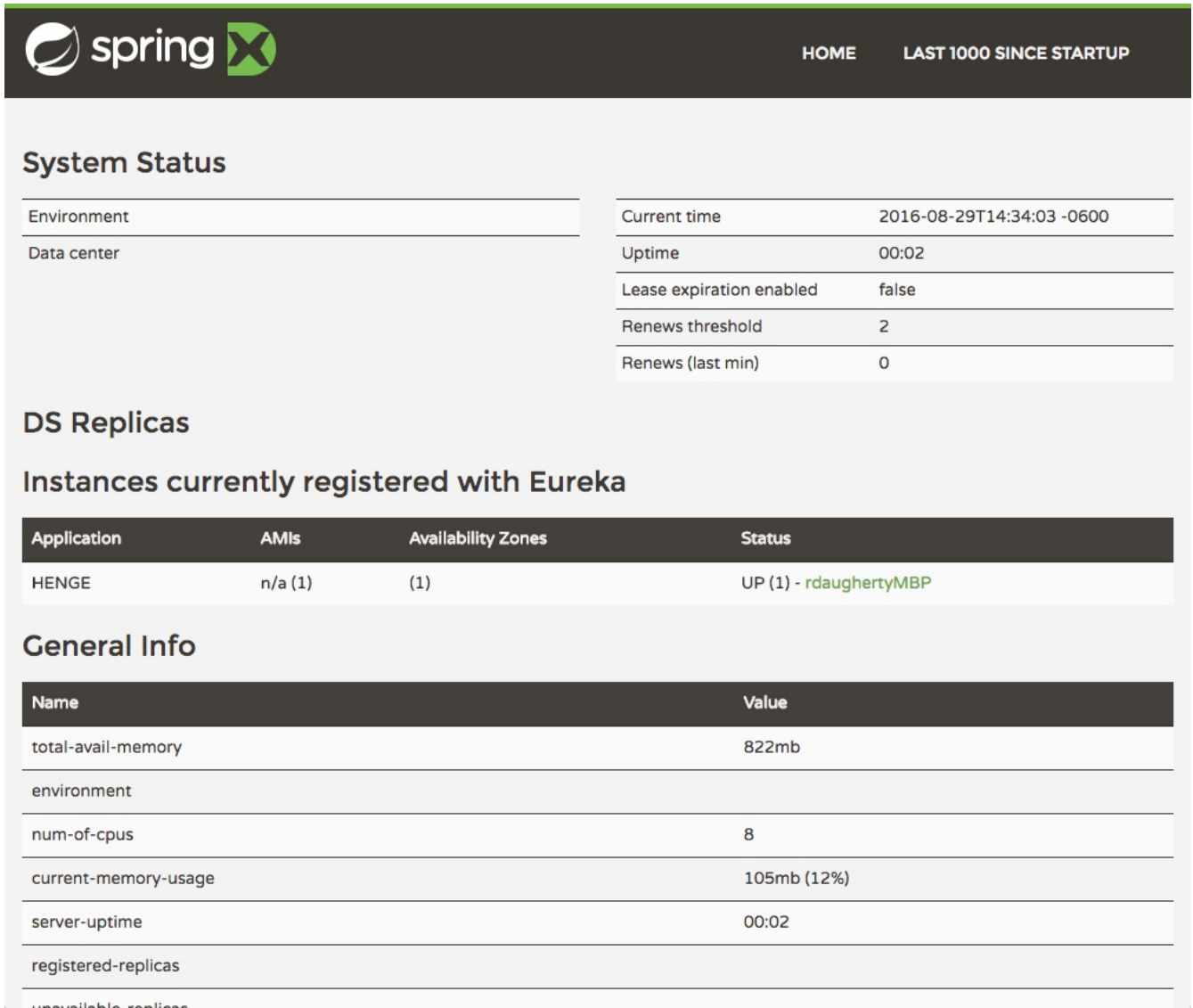
The screenshot shows the Spring Boot Eureka Server Web Page. At the top, there's a dark header with the Spring logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. Below the header, the 'System Status' section contains two tables. The left table lists 'Environment' and 'Data center'. The right table shows 'Current time' (2016-08-29T14:34:03 -0600), 'Uptime' (00:02), 'Lease expiration enabled' (false), 'Renews threshold' (2), and 'Renews (last min)' (0). The 'DS Replicas' section follows. Below it, the 'Instances currently registered with Eureka' section features a table with columns: 'Application', 'AMIs', 'Availability Zones', and 'Status'. One instance, 'HENGE', is listed with 'n/a (1)' AMIs, '(1)' Availability Zones, and a status of 'UP (1) - rdaughertyMBP'. The 'General Info' section at the bottom contains a table with columns 'Name' and 'Value', listing metrics like 'total-avail-memory' (822mb), 'environment', 'num-of-cpus' (8), 'current-memory-usage' (105mb (12%)), 'server-uptime' (00:02), 'registered-replicas', and 'unavailable-replicas'.

Figure 4. Eureka Server Web Page

11.3. About Our Eureka Implementation

This section explains how we implemented the Eureka server and client using Spring Boot. The steps we took are here for the sake of clarity; there is no need to recreate any of them.

11.3.1. Eureka Server Implementation

In order to implement a Eureka Server on Spring Boot, we took the following steps.

1. Add dependencies:

```
org.springframework.boot:spring-cloud-starter-eureka-server
```

2. Create a Spring Boot **Application** class that acts as the Eureka server implementation:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication ①
@EnableEurekaServer ②
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

① Sets this project as a Spring Boot **Application**.

② Loads the Eureka server.

3. Configure the Eureka server in the **application.yml** file:

```
server:
  port: 8761

eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false ①
    fetchRegistry: false ①
    serviceUrl:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ ②
```

① Simply tells this instance to not register itself with Eureka

② Property to set the URL and port of Eureka Server

4. Create the **bootstrap.yml** file:

```
spring:
  application:
    name: henge
```

Spring Cloud uses the information in **bootstrap.yml** at service startup to discover the Eureka service registry and register the service and its **spring.application.name**, **host**, **port**, etc.

11.3.2. Eureka Client Implementation

The following steps were taken to implement Henge as a Eureka client via Spring Boot.

1. Create the Eureka Client configuration class:

```
@Configuration
@EnableEurekaClient ❶
@Profile("eureka")
public class EurekaClientConfig {

}
```

❶ Enable this project to connect to a Eureka server.

2. Configure the Eureka client in **application.yml**.

```
eureka.client.serviceUrl.defaultZone: http://localhost:8761/eureka/
```

This property is required in **application.yml** (or **application.properties**) to set the Eureka Server URL. It is used by the client, in this case Henge, to register itself with Eureka.