
# Henge User's Guide

Kenzan

Version 0.9.0, September 2, 2016

# Henge User's Guide


1. Getting Started . . . . . 1
   1.1. Setting Up Henge . . . . . 1
      1.1.1. Dependencies . . . . . 1
      1.1.2. Build and Run Henge . . . . . 1
      1.1.3. Test Henge . . . . . 2
   1.2. REST API Reference . . . . . 3
2. Domain Objects . . . . . 3
   2.1. Property . . . . . 3
   2.2. Scope . . . . . 3
   2.3. PropertyGroup . . . . . 3
   2.4. FileVersion . . . . . 5
   2.5. VersionSet . . . . . 6
   2.6. Mapping . . . . . 7
3. Profiles . . . . . 7
   3.1. Activating Profiles . . . . . 8
      3.1.1. Specify an Application Property . . . . . 8
      3.1.2. Use a Command Line Argument . . . . . 8
   3.2. Henge Profiles . . . . . 9
      3.2.1. Default Profile . . . . . 9
      3.2.2. Dev Profile . . . . . 10
      3.2.3. Eureka Profile . . . . . 10
      3.2.4. Flatfile_local Profile . . . . . 10
      3.2.5. Flatfile_s3 Profile . . . . . 11
      3.2.6. Cassandra Profile . . . . . 11
      3.2.7. Metrics Profile . . . . . 11
   3.3. Profile-Specific Configurations . . . . . 12
      3.3.1. Noteworthy Configurations . . . . . 12
4. Metrics . . . . . 13
   4.1. Required Services . . . . . 13
   4.2. Configuring Docker . . . . . 13
      4.2.1. macOS . . . . . 13
      4.2.2. Linux . . . . . 14
   4.3. Starting InfluxDB and Grafana . . . . . 15
   4.4. Starting Henge With Metrics Enabled . . . . . 16
      4.4.1. Accessing InfluxDB . . . . . 16
      4.4.2. Accessing Grafana . . . . . 16
5. Repositories . . . . . 17
   5.1. S3 Setup . . . . . 17

5.1.1. Create AWS Access Keys . . . . . 17
5.1.2. Create an S3 Bucket . . . . . 18
5.1.3. Set Up AWS Credentials Locally . . . . . 18
5.1.4. Build and Run Henge . . . . . 19
5.2. Cassandra Setup . . . . . 19
5.2.1. Run Cassandra and Load the Schema . . . . . 19
5.2.2. Build and Run Henge . . . . . 20
5.2.3. Stopping Cassandra . . . . . 20
6. Eureka Registry and Discovery Service . . . . . 21
6.1. What is Eureka? . . . . . 21
6.2. Running Henge With Eureka . . . . . 21
6.3. About Our Eureka Implementation . . . . . 22
6.3.1. Eureka Server Implementation . . . . . 22
6.3.2. Eureka Client Implementation . . . . . 24

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. Getting Started

Henge is a flexible key/value store for dynamic configuration properties. The goal of the project is to produce a performant, reliable implementation of everything a production-ready property server application should provide. Because this functionality is composed of a modular architecture, you can customize the setup that works best for your organization.

The idea of creating Henge came from the Netflix Archiaus project, in particular the open issue #132 which calls for creating a standard properties service. As described, the service would allow a PolledConfigurationSource on the client side to get all properties, which could then be changed dynamically on the server.

Henge supports pluggable persistence, life cycle management, validation, and querying capabilities of properties. A set of REST APIs are provided to interface with Henge and its properties.

## 1.1. Setting Up Henge

The following sequence shows how to build and run Henge using a local flatfile repository on both Linux and macOS. For how to run Henge using S3 flatfile or Cassandra repositories, see Repositories.

### 1.1.1. Dependencies

Java 1.8.x

### 1.1.2. 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 the 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.

### 1.1.3. 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

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

## 1.2. 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. 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.

## 2.1. 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.

## 2.2. Scope

A `Scope` is a key-value pair used to specify a situation where a given `Property` assumes a particular value. A `Scope` is useful when a configuration variable has different values depending on the situation. For example, a property may have different values depending on the environment where the application is executed. In this instance you could have a key as **environment** and possible values of **development**, **test**, and **production**. This would define three instances of `Scope` that can be used to set different values for the property.

## 2.3. 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 **development**, **test** and **production** environments where the values for these properties may be different. You would define a `PropertyGroup` with four properties where each of them have three `propertyScopedValues`. 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",
        "propertyScopedValues": [{
            "key": "environment=development", ①
            "value": "127.0.0.1" ②
        }, {
            "key": "environment=test",
            "value": "192.168.0.10"
        }, {
            "key": "environment=production",
            "value": "192.168.0.20"
        }]
    }, {
        "name": "dbport",
        "description": "Port of database",
        "defaultValue": "4321" ③
    }, {
        "name": "dbuser",
        "description": "Username for connecting to database",
        "propertyScopedValues": [{
            "key": "environment=development",
            "value": "devuser"
```

```
        }, {
            "key": "environment=test",
            "value": "testuser"
        }, {
            "key": "environment=production",
            "value": "produser"
        }]
    }, {
        "name": "dbpasswd",
        "description": "Password for connecting to database",
        "propertyScopedValues": [{
            "key": "environment=development",
            "value": "1234"
        }, {
            "key": "environment=test",
            "value": "1234"
        }, {
            "key": "environment=production",
            "value": "b40df5b0d4d173a554b1030c0f453dac"
        }]
    }]
}
```

① Note that the key of the `propertyScopedValue` is a `Scope` whose key is **environment** and value is **development**.

② Also observe that **127.0.0.1** is the value of the **dbhost** property when the `Scope` is **environment=development**.

③ For **dbport**, there are no `propertyScopedValues` and only a `defaultValue` because the **dbport** is the same for all environments.

## 2.4. 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": "TW91bnQgRWxiZXJ0LCBDb2xvcmFkb3wzOS4xMTc4NTEyfC0xMDYuNDQ1MTU5OQpNb3
  VudCBNaXRjaGVsbCwgTm9ydGggQ2Fyb2xpbmF8MzUuNzY0OTYxMnwtODIuMjY1MTEKTW91bnQgUmFp
  bmllciwgV2FzaGluZ3Rvbnw0Ni44NTI5MTI5fC0xMjEuNzYwNDQ0Ng==",
  "filename": "GeoLoc.txt",
  "createdBy": "rdaugherty",
  "createdDate": "2016-08-22T09:44:51.58",
  "modifiedBy": "rdaugherty",
  "modifiedDate": "2016-08-22T09:44:51.58"
}
```

## 2.5. 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`.

## 2.6. 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 maps a set of `Scopes` to a specific version of a `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`. The **application** itself is stored in the `Mapping` as a scope as shown in the following JSON example:

```
{
  "{\"scopeSet\":[{\"key\":\"env\",\"value\":\"development\"},{\"key\":\"stack\",\"val
ue\":\"stack1\"},{\"key\":\"application\",\"value\":\"MasterAppOne\"}]}" : {
    "name" : "VersionSet-1",
    "version" : "1.0.0"
  }
}
```

A search is made 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.

# 3. 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.

# 3.1. 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.

## 3.1.1. 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
```

## 3.1.2. 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
```

## 3.2. Henge Profiles

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

### 3.2.1. 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 |
|---|---|
| spring.application.name | The application name. Eureka uses this as the name of the client service. |
| spring.profiles.active | The list of active profiles. |
| spring.jersey.type | Specifies whether to use Jersey as a Filter or a Servlet. Default value: `filter` |
| multipart.max-file-size | The upper bound on the size of `FileVersion` objects that can be stored in Henge. |
| server.contextPath | The context root of the web application. |
| swagger.api.version | The version of the Swagger API. |
| swagger.schemes | Comma separated list of accepted protocols. Example: `http,https` |
| swagger.base.path | Base path for the Swagger APIs. |
| swagger.resource.package | Package from which Swagger must scan for endpoints. |
| swagger.scan | Specifies whether Swagger should scan for endpoints. Recommended value: `true` |
| 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. |
| cache.expiration.minutes | The time the cache lives after each write to it. |
| text.encoding | The encoding used to convert bytes to text (and vice versa) throughout all repository implementations. |
| scope.precedence.configuration | 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. |
| scope.application.name.key | String that represents the application name in the scope keys. |

| Variable | Definition |
|---|---|
| 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). |
| security.user | User name for authentication when executing REST requests that are not GET. |
| security.password | Password for authentication when executing REST requests that are not GET. |

### 3.2.2. 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). |

### 3.2.3. 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**

### 3.2.4. 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. |

### 3.2.5. 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. |

### 3.2.6. 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. |

### 3.2.7. 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. |

| Variable | Definition |
| --- | --- |
| metrics.influx.password | Password for connecting to the InfluxDB database. |
| metrics.influx.database | Name of the InfluxDB database. |
| metrics.influx.periodInSeconds | The period for publishing metrics. For example, a value of **5** means that metrics are sent to InfluxDB every 5 seconds. |

## 3.3. 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.

### 3.3.1. 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.

# 4. 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.

## 4.1. 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 indvidually 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.

## 4.2. Configuring Docker

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

### 4.2.1. 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.

### 4.2.2. 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.

## 4.3. 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.

## 4.4. 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.

### 4.4.1. 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`

### 4.4.2. 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**.

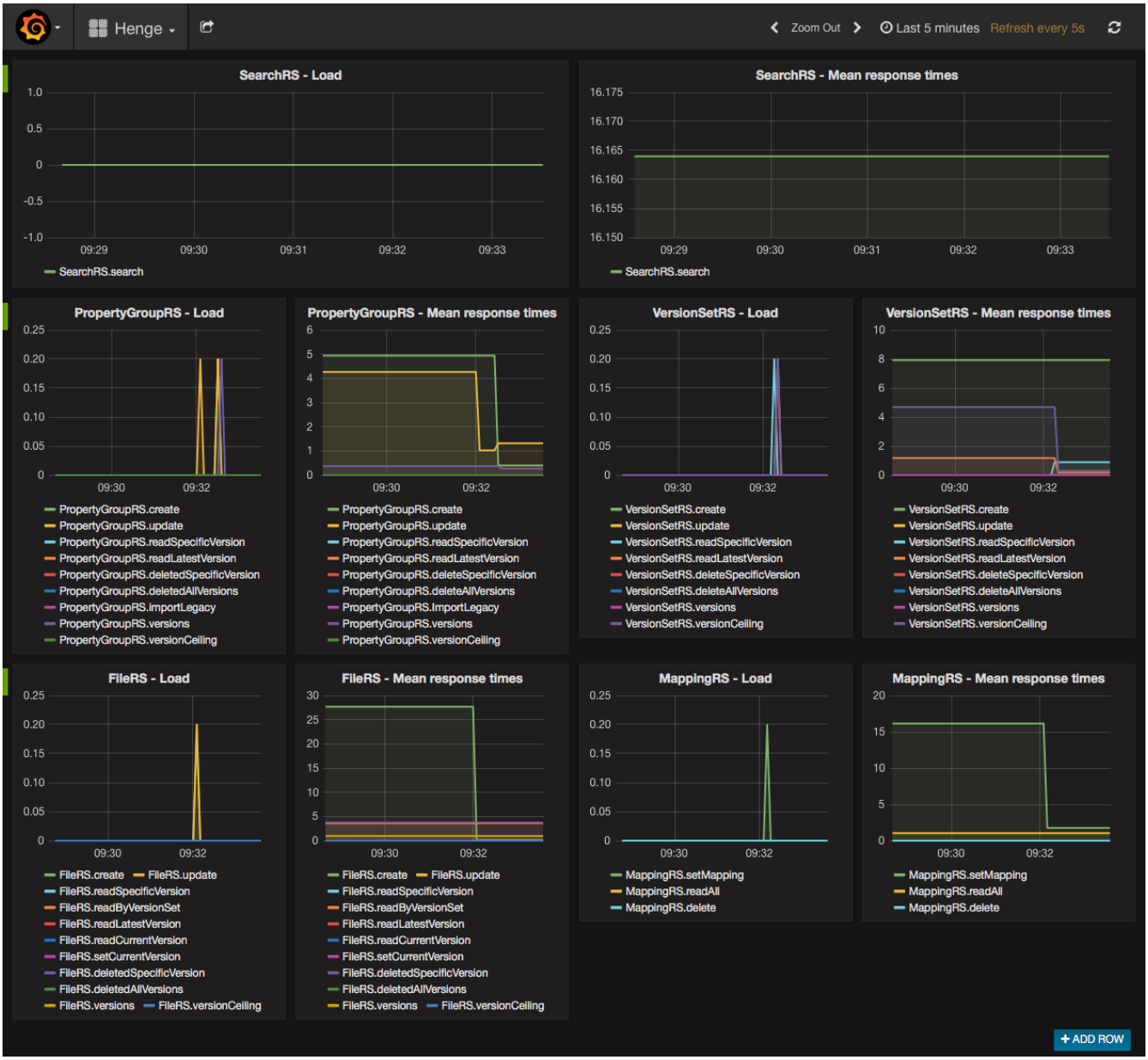


*Figure 1. Henge Dashboard*

# 5. 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.

## 5.1. S3 Setup

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

### 5.1.1. 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.

### 5.1.2. 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.

### 5.1.3. 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}
```

### 5.1.4. 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.

## 5.2. Cassandra Setup

### 5.2.1. 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.

### 5.2.2. 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.

### 5.2.3. 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 auwx | 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
```

# 6. 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.

## 6.1. 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.

## 6.2. 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.

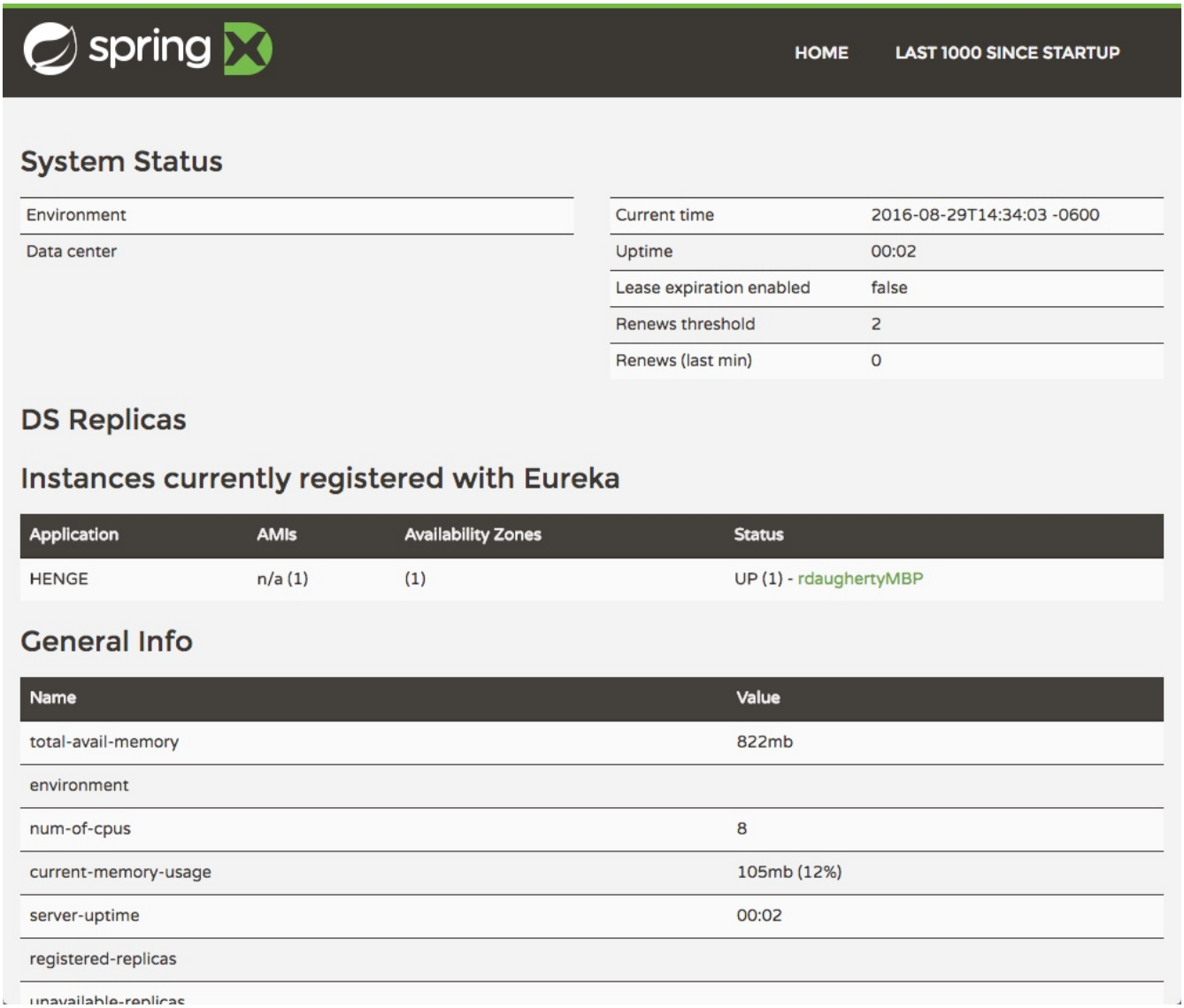


*Figure 2. Eureka Server Web Page*

# 6.3. 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.

## 6.3.1. 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.

### 6.3.2. 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 ①
@Profile("eureka")
public class EurekaClientConfig {

}
```

① 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.