

# **Prophasis - An IT Infrastructure Monitoring Solution**

*Cameron Gray*



Fourth Year Project Report

School of Informatics

University of Edinburgh

2016



## **Abstract**

Prophasis is an IT infrastructure monitoring system that is designed to suit small to medium size businesses where a system needs to be intuitive to manage. Management of the entire system can therefore be handled from a single, responsive web interface. It is also suitable as a one-stop tool with support for both time series monitoring in addition to real time alerting. Traditionally two different tools would be needed to gain this level of monitoring.



# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Background . . . . .	7
1.2	Prior Work . . . . .	8
1.2.1	Nagios . . . . .	8
1.2.2	Icinga 2 . . . . .	9
1.2.3	Munin . . . . .	10
1.3	Improvements . . . . .	10
1.3.1	Configuration Management . . . . .	11
1.3.2	Time Series Monitoring & Real Time Alerting . . . . .	11
1.3.3	Expandability . . . . .	11
<b>2</b>	<b>Design</b>	<b>13</b>
2.1	Technology Choice . . . . .	13
2.2	System Structure . . . . .	14
2.2.1	Agent . . . . .	14
2.2.2	Core . . . . .	15
2.2.3	Web Interface . . . . .	19
2.3	Monitoring Methodology . . . . .	19
2.3.1	Host Management . . . . .	19
2.3.2	Plugins . . . . .	20
2.3.3	Checks . . . . .	20
2.3.4	Schedules . . . . .	20
2.3.5	Services . . . . .	21
2.3.6	Alerts . . . . .	21
2.3.7	Classification . . . . .	22
2.4	User Interface . . . . .	22

<b>3</b>	<b>Implementation</b>	<b>23</b>
3.1	Technologies . . . . .	23
3.1.1	Flask Web Framework . . . . .	23
3.1.2	Tornado HTTP Server . . . . .	23
3.1.3	SQLAlchemy ORM . . . . .	24
3.1.4	Lupa . . . . .	24
3.1.5	PostgreSQL Database . . . . .	24
3.1.6	Appdirs . . . . .	24
3.1.7	AdminLTE & Bootstrap . . . . .	25
3.1.8	Handlebars.js . . . . .	25
3.2	Plugin Interface . . . . .	25
3.3	Database . . . . .	25
3.4	Common Module . . . . .	28
3.5	Core . . . . .	28
3.5.1	Dispatcher . . . . .	28
3.5.2	Classification . . . . .	29
3.6	Agent . . . . .	30
3.7	Web Interface . . . . .	31
3.7.1	Configuration . . . . .	31
3.7.2	Reporting . . . . .	36
<b>4</b>	<b>Testing</b>	<b>39</b>
4.1	Unit Testing . . . . .	39
4.2	Mock Agent . . . . .	40
<b>5</b>	<b>Evaluation</b>	<b>43</b>
<b>6</b>	<b>Conclusion</b>	<b>45</b>

# Chapter 1

## Introduction

### 1.1 Background

In recent years, almost all businesses have been expanding their IT infrastructure to handle the modern demand for IT systems. As these systems grow and become increasingly important for business operation it is crucial that they are sufficiently monitored to prevent faults and periods of downtime going unnoticed. There is already a large market of tools for monitoring IT systems however they are designed for use on massive scale networks managed by teams of specialised systems administrators. They are therefore complicated to set up and manage and multiple tools are often required to gain a suitable level of monitoring.

For example, tools generally either fall into the category of real time alerting (i.e. telling someone when something breaks) and time series monitoring (i.e. capturing data about the performance of systems and presenting graphs and statistics based on it), there is a large gap in the market for tools that provide both of these in one package. This reduces the time required to manage the system as it eliminates the need to set up and configure two completely separate tools.

These tools are also generally managed and configured through various configuration files split across different machines on the network. This means that in order to efficiently use these tools a configuration management system such as Puppet must be used. In a small business with limited IT resources, a completely self contained system is often preferable.

## 1.2 Prior Work

This section will review current IT infrastructure monitoring systems and evaluate them on several points as follows:

- Support for timeseries monitoring and real time alerting
- How they can be configured to monitor custom metrics
- How are alert thresholds defined
- How configuration and custom code is delivered to nodes (if required)
- How the user configures the system
- How dependencies are handled

### 1.2.1 Nagios

**Timeseries monitoring and real time alerting** Nagios is primarily focused at real time alerting and therefore has very little in the way of timeseries monitoring. Additional plugins are available which can be used to graph metrics over time but these cannot be used to make decisions on the status of a given system or service. All that is supported in terms of alerting on historical data is to refrain from alerting until a given condition has been observed in the previous  $n$  checks, there is no support for alerting based on trends in historical data. Supports basic display of changes in state of hosts/services over time but not individual metrics.

**Support for custom metrics** Nagios has support for custom metrics through the NRPE (Nagios Remote Plugin Executor) plugin. These plugins can be any sort of executable which prints out a message to represent the data read as well as a specific exit code which defines the status, for example "OK", "Critical" .etc

**Alert threshold definition** Thresholds for NRPE agents must be set on the remote server itself. These thresholds are passed into the remote plugin as an argument when it is executed and are used internally by the script to output the appropriate alert level.

**Code/Config delivery to nodes** Nagios does not have any in built functionality to distribute configuration files or plugin code to remote nodes. In order to automate this, additional software such as Puppet would be required.



**How the user configures the system** Configuration for Nagios is primarily managed through text files stored on disk. Third party configuration tools are available to allow the system to be configured through a web interface. Configuration lives on both the Nagios server as well as on the machines being monitored.

**How dependencies are handled** Rigid tree - No way to define that a service/host is dependent on a given host OR another host being available. This reduces its usefulness in modern networks where redundancy and failover is commonplace. These are defined in config files that live on the Nagios server.

### 1.2.2 Icinga 2

**Timeseries monitoring and real time alerting** Like Nagios, Icinga's primary focus is around real time alerting however it has now introduced support for graphing performance metrics. No support for alerting based on trends in historical data beyond Nagios's idea of a state change changing to HARD once it has been observed  $n$  times.

**Support for Custom Metrics** Custom metrics can be written as scripts in any language as long as they echo a message to describe the status of what they are checking and use a certain exit code to define the status e.g. "OK", "CRITICAL", "WARNING".etc.

**Alert threshold definition** Thresholds are passed into the check commands as command line arguments. Therefore they are defined on the machines being monitored individually.

**Code/Config delivery to nodes** Icinga 2 does not have any built-in mechanism to distribute code and config files to remote hosts. In order for this to be achieved, additional software such as Puppet would be required.

**How the user configures the system** Configuration is managed through configuration files stored on disk. Configuration files exist both on the Icinga server as well as the nodes being monitored.

**How dependencies are handled** Same as Nagios with a rigid tree. No way to define a host/service as being dependent on one of several different machines as is common in modern environments with redundancy/failover systems.

### 1.2.3 Munin

**Timeseries monitoring and real time alerting** Munin is targeted primarily as a timeseries monitoring tool. It therefore has good functionality for graphing data over time. However, it does not have much power in the way of alerting other than some very basic functionality where plugins must manually send out emails/syslog alerts. This is not really sufficient for any sort of production use and it is instead recommended to use a tool such as Nagios and push the data from Munin into it.

**Support for custom metrics** Munin has a simple interface for custom plugins where a plugin is a simple script that prints out the name and value of the data being collected. This is then served by the Munin node which the Munin server contacts over the network to fetch values from the nodes.

**Alert threshold definition** Nothing built in, plugins however can create thresholds internally and use them for alerting (as detailed above) although this isn't really the intended use of Munin.

**Code/Config delivery to nodes** Nothing built in, code and config must be distributed manually or automatically through the use of additional software such as Puppet.

**How the user configures the system** Configuration is handled through text files stored on disk. These are stored on both the Munin server as well as machines being monitored.

**How dependencies are handled** No dependency functionality.

## 1.3 Improvements

Prophesis is designed for use in a small to medium business with limited IT resources. They may have a small IT team with limited resources or may not even have

a dedicated IT team at all, instead relying on one or two employees in other roles who manage the business's IT systems on the side of their regular jobs. Therefore the system needs to be quick to deploy and manage with a shallow learning curve. In order to use the system efficiently there should be no requirement for additional tooling to be deployed across the company.

### **1.3.1 Configuration Management**

It should be possible to manage the configuration of the system from a single location. Prophasis therefore provides a responsive web interface where every aspect of the system's operation can be configured, Prophasis then handles distributing this configuration to all other machines in the system in the background. Custom code for plugins is handled in the same way; it is uploaded to the single management machine and is then automatically distributed to the appropriate remote machines when it is required.

### **1.3.2 Time Series Monitoring & Real Time Alerting**

Prophasis provides both the ability to alert administrators in real time when a fault is discovered with the system alongside functionality to collect performance metrics over time and use this data to generate statistics about how the system has been performing. This time series data can be used to both investigate the cause of a failure in post-mortem investigations in addition to being able to be used to predict future failures by looking at trends in the collected data.

### **1.3.3 Expandability**

It is important that a monitoring tool can be expanded to support the monitoring of custom hardware and software. An example of this would be hardware RAID cards. Getting the drive health from these types of devices can range from probing for SMART data all the way to communicating with the card over a serial port. It is therefore crucial that Prophasis can be easily expanded to support custom functionality such as this. Therefore Prophasis supports a system of custom "plugins" which can be written and uploaded to the monitoring server where they can then be configured to monitor machines. These plugins are designed to be self contained and to follow a well defined and documented structure. This provides scope for a plugin "marketplace" therefore eliminating the need for every user to implement custom monitoring

code for the systems they are using.

# Chapter 2

## Design

### 2.1 Technology Choice

**Python** It was decided to use Python as the language of choice to implement Prophasis. This was chosen for many different reasons: it is available on practically all platforms and is usually bundled with most UNIX-like operating systems. It also abstracts a large amount of low level details away which both saves development time and prevents subtle, hard to find errors such as memory management issues. Python also has a huge library of available plugins along with a powerful and flexible package manager (pip) and package repository (PyPi). This allows external plugins to be used to provide functionality instead of requiring everything to be implemented from scratch. This saves development time and means that specialised, well tested and maintained code can be used to provide some of the functionality. Pulling in packages from PyPi instead of bundling external libraries with Prophasis ensures that any external libraries are kept up to date and prevents any licensing issues that could arise from bundling code from other sources alongside the Prophasis source code.

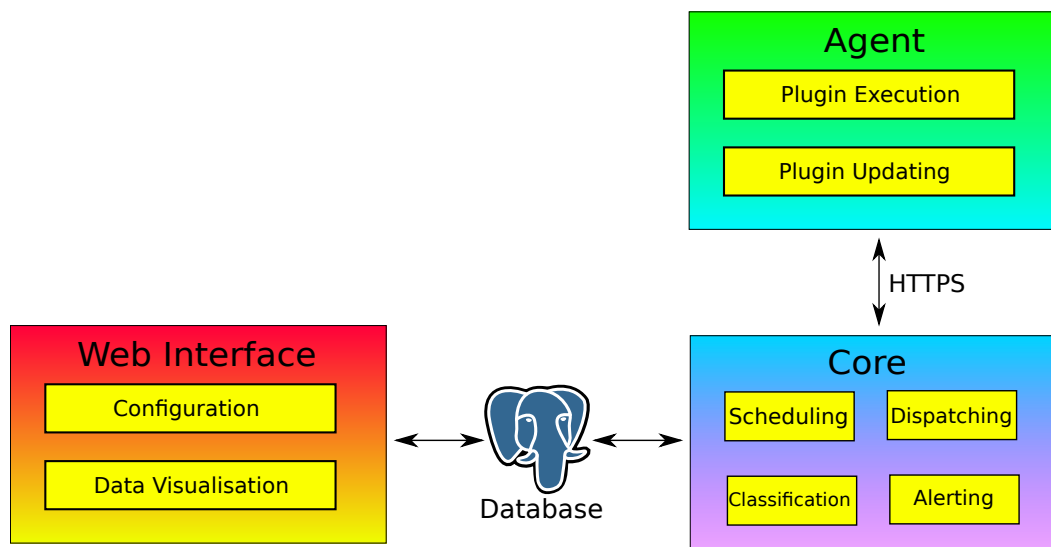
**HTTP for Communication Protocol** HTTP was chosen as a communication protocol for several reasons. It is well supported and understood with a large variety of server and client libraries available. Using a premade library has major benefits for development time, security and support. HTTPS also provides suitable encryption and certificate validation technologies to secure the system. HTTP is also widely permitted through firewalls and can be routed through web/reverse proxies. This means that Prophasis can be operated on most networks without causing problems with firewalling. HTTP also includes methods for transferring simple data as well as large bi-

nary files which is required for sending plugins to remote agents. Originally an attempt was made to use SSH to implement communication between the core and the agent. However, it was found that this introduced a huge amount of additional complexity on both the server and client, especially if the SSH server were to be embedded into the agent to keep it as a self contained application. This additional complexity not only made maintenance and development harder, it also increases the risk of errors being introduced. SSH was also more difficult to route through restricted networks. Another benefit of using HTTP is how many well documented HTTP libraries and servers there are available, this means that alternative agents can be implemented with ease versus SSH which would involve having to handle a custom wire protocol.

## 2.2 System Structure

The system is split up into three separate components; Web, Core and Agent. Web and core both share the same relational database allowing data to be shared between them. Figure 2.1 shows the individual components of the system and how they all interact.

Figure 2.1: Diagram showing the layout of the components of the system



### 2.2.1 Agent

The agent runs on every machine that is being monitored and provides an API for communication with the monitoring server. It listens on port 4048 (by default) and

exposes an HTTPS API. This API includes methods to check the version of a plugin currently installed on the agent, a method to push updated plugins to the agent and another method to execute a plugin and to retrieve the value and message produced by it.

The agent's API is authenticated using a long, pre-shared key of which a salted hash is stored on the agent in a configuration file. Being hashed prevents users who may have access to read the configuration file (possibly through misconfiguration) from getting the key to be able to communicate with the agent.

### 2.2.2 Core

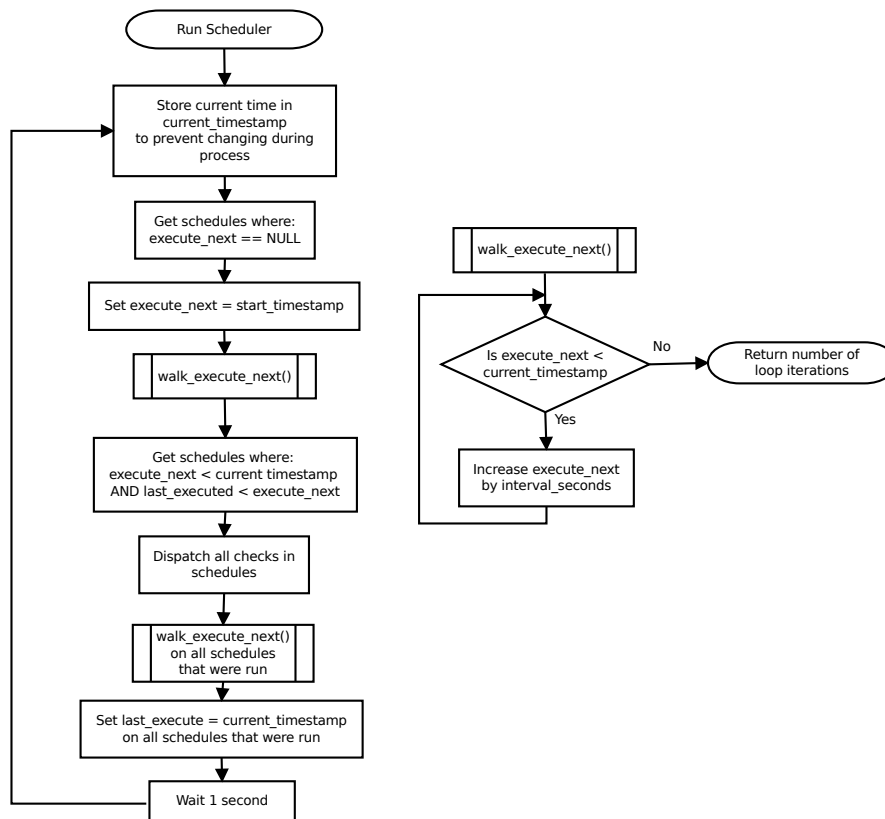
The core runs on the machine that is performing the monitoring, it has several different roles; scheduling, dispatching, classification and alerting.

#### 2.2.2.1 Scheduling

The core is responsible for looking at the schedules stored in the database and executing the appropriate checks on the correct hosts at the correct time. There is a configuration value for "maximum lateness" that defines how late after its defined time slot a check can be executed. The core repeatedly checks the database looking at the intervals for each schedule along with the time at which a given schedule was last executed. If it decides that a schedule is due to be executed it passes this onto the dispatcher.

Figure 2.2 describes how the scheduler operates. Each schedule has a `start_timestamp` which is defined by the user when the schedule is created, an `interval` which is how often the schedule executes and a value for `execute_next` which is the timestamp that the schedule is next to be executed. When the scheduler starts up it first gets all schedules that do not have an `execute_next` value - These are schedules that have never run. It then calls `walk_execute_next` which is a simple algorithm that "fast forwards" the `execute_next` value until it reaches a timestamp that is in the future. It then retrieves any schedules that are due to be executed (`execute_next` is in the past) and executes them, it then calls `walk_execute_next` on each of these to set the `execute_next` value to the time that the schedule should be run again. The algorithm will then wait for 1 second before executing the process again.

Figure 2.2: Flowchart describing the operation of the scheduler



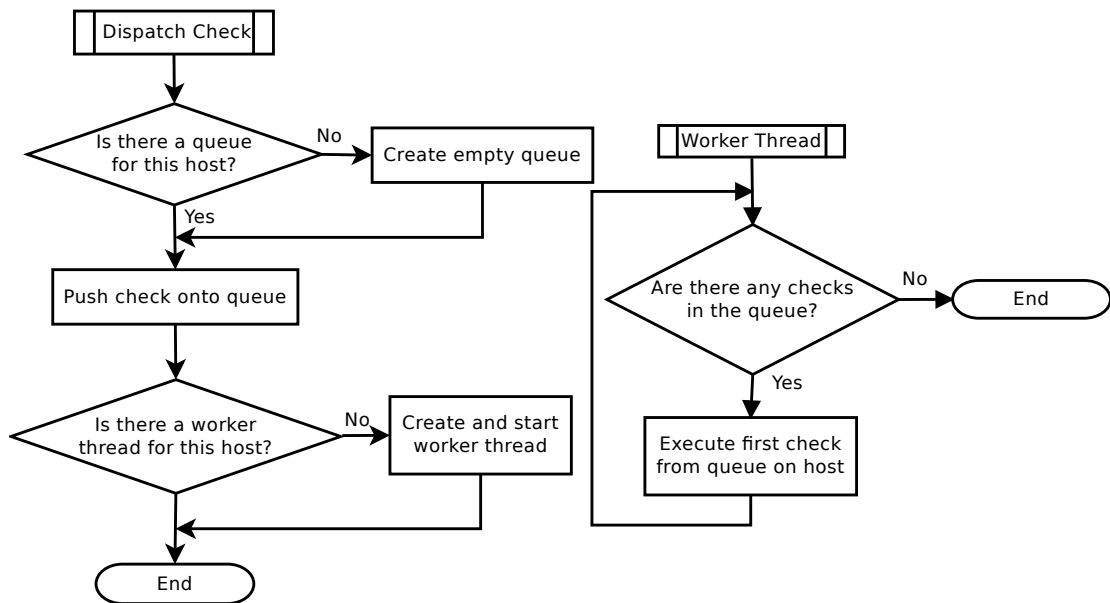
### 2.2.2.2 Dispatching

The dispatcher component of the core is responsible for issuing checks to agents when they are due to be run (as decided by the scheduler). Checks may take some time to execute so executing these all in series would all be impractical. The solution for this was for the dispatcher to spawn a process for each agent that it is currently executing checks for. Each process maintains a queue of plugins that are due to be executed and issues them to the agent in the order that they were dispatched. This way only one plugin can be executing on a given agent at any moment in time. This both prevents agents from becoming overwhelmed and means that plugin developers do not need to be concerned about other plugins interfering with their plugin. Figure 2.3 shows the



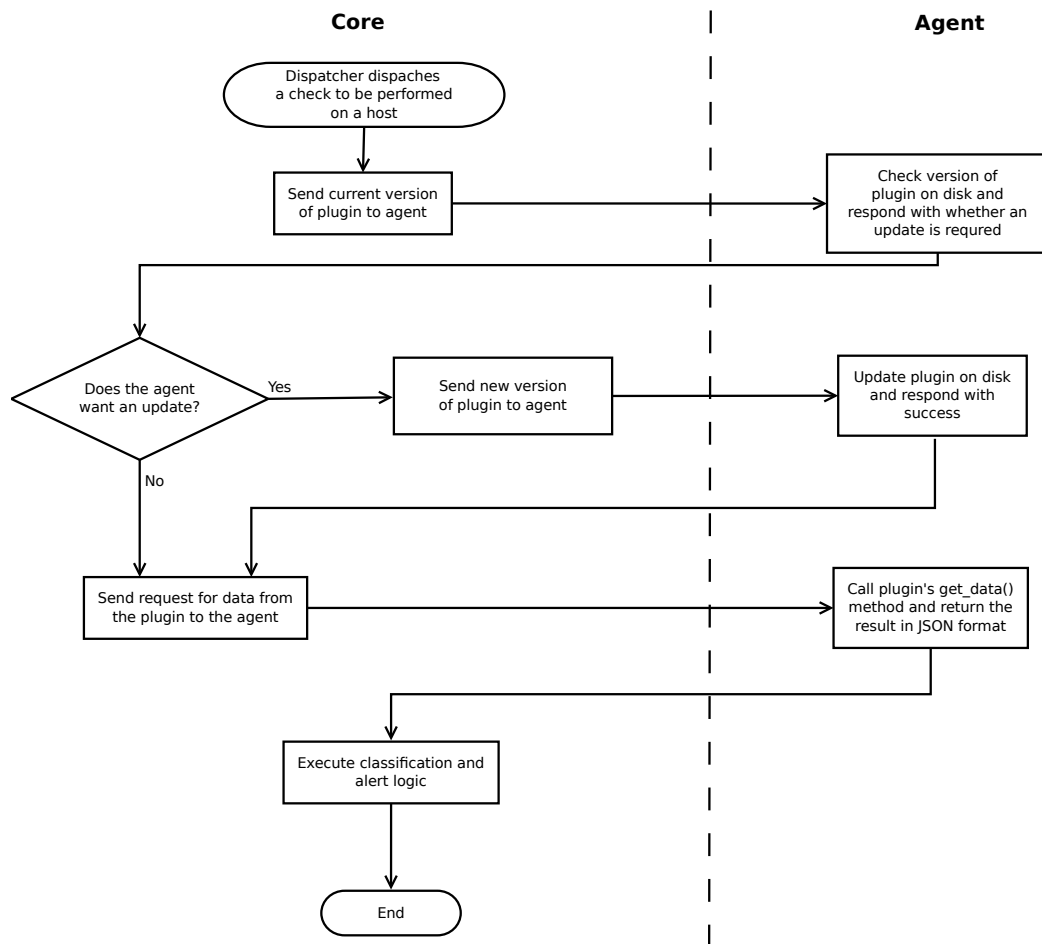
operation of the dispatcher.

Figure 2.3: Flowchart describing the operation of the dispatcher



**Communication Between Core and Agent** When a check is dispatched, the core and agent communicate to first of all establish if the agent already has the correct version of the plugin due to be executed installed. If it does not then an update will be sent to the agent. Once this is done the core will then request the agent to execute the plugin and the data will be returned for classification. Figure 2.4 describes the communication between the core and the agent.

Figure 2.4: Flowchart describing the communication between the core and the agent when a check is dispatched



### 2.2.2.3 Classification

When data is collected from an agent, it needs to be classified as "ok", "major", "minor", "critical" or "unknown". Classification is performed by lua code that is stored alongside each plugin. When the core finishes collecting a from an agent it will retrieve the classification code for the plugin and execute it in a sandboxed lua runtime environment. The result of the classification is then stored in the database. The use of Lua code provides total flexibility, classifications can be as simple as comparing values to a threshold or could go as far as looking at previous historical values and classifying based on a trend in the data. Plugins define how many previous historical values they want to classify on ( $n$ ), the plugin's classifier code is then executed with two Lua tables, one containing the previous  $n$  "values" stored and the other containing the previous  $n$  "messages".

#### **2.2.2.4 Alerting**

Once the core has classified a piece of data, the core now needs to work out whether it needs to send an alert or not. To do this it looks at the previous state for the plugin and host along with the current state. It then looks through the database for alerts that match that state transition and are applicable to the host/plugin combination. If any alerts match it will call the alert module to send the alert out to inform the necessary people about the state change.

### **2.2.3 Web Interface**

Prophasis provides a web interface for both configuring the system and viewing collected data. The interface is designed to be clear and easy to understand for users. It is also built to be fully responsive so will work correctly on mobile device without any loss of functionality. Designing it to be responsive was the aim from the outset as servers often fail when the administrator may not currently be in easy reach of a computer so will need to use a mobile device to investigate the issue.

The web interface will connect to the same database as the core, the web interface will update the configuration data stored in this database and will retrieve the data about hosts.

The web interface provides dashboards for visualisation of data collected from plugins. The collected data can then be displayed using graphs, tables, lists of messages or any other way that suits the type of data collected.

## **2.3 Monitoring Methodology**

This section explains how Prophasis is laid out and introduces terminology used throughout the system.

### **2.3.1 Host Management**

In Prophasis, a "host" refers to a single machine that is being monitored. To aid management and organisation, it is possible to organise hosts into "Host Groups." These can be comprised of hosts or other groups and can be used in place of hosts when

defining services and checks. Hosts can be grouped in various ways such as their role (Webserver, VM Host), hardware configuration (Hardware RAID, Contains GPU), location (Edinburgh Office, Datacenter) or any other way that makes sense given the specific implementation.

### 2.3.2 Plugins

A "plugin" is a package that checks a single attribute of a system. For example a plugin could check the CPU load of a given machine or be used to check the health of all the hard drives in a system. Plugins are implemented as Python modules that implement a specific API, they can return both a "value" (a numerical representation of the data they collected) and/or a "message" (a string representation of the collected data). Plugins are then packaged in a .tar.gz archive along with a manifest JSON file which contains information about the plugin for use when it is installed.

Plugins are automatically distributed from the core to remote machines and when executed by the agent the value and message are returned to the core for classification and storage in the database.

### 2.3.3 Checks

A "check" is a named set of hosts/host groups and a set of plugins to be executed on them. When a check is executed, all of the plugins specified in the check will be executed across all of the hosts specified in the check.

Checks allow logical grouping of various plugins. For example you may have a check for "Webserver Resource Usage" which will execute plugins to check the CPU load and memory across all hosts in the "Webservers" host group.

### 2.3.4 Schedules

A schedule defines when one or more checks are to be executed. Each schedule can contain multiple "intervals" which define when the check is run. An interval has a start date/time and a time delta that defines how regularly it is to be implemented.

### 2.3.5 Services

Services can be used to define a more fine grained representation of how the availability of a host affects the performance of the system overall. A service is defined in terms of "dependencies" and "redundancy groups". Each dependency represents a host that must be operational for the service to work. A redundancy group can contain multiple hosts where at least one must be operational for the service to work.

As an example, you may have a "Website" service that has a dependency on the single router but has a redundancy group containing multiple web servers. Therefore, if the router fails then the website will be marked as failed however if one of the web servers fails but at least one other web server is still operational, the website service will only be marked as degraded.

The use of services provides a clearer view of the impact of a given failure on the functionality of the network. It also allows alerts to be set up so that they are only triggered when the service functionality is severely impacted and prevents alerts from being sent out for host failures that do not have a severe impact.

### 2.3.6 Alerts

In Prophasys, alerts are set up to communicate with a specific person/group of people in a certain situation. Alerts are defined in terms of to/from stage changes where an alert will be sent if the state of a host/service changes from one state to another (e.g. "ok" to "critical"). These can then be restricted to certain hosts, host groups, services, plugins and checks.

The system supports "alert modules" - Python modules that are used to send alerts using various methods. Examples of alert modules could be email, SMS or telephone call.

Multiple alerts can be set up to handle different severities of issues. For example, if a redundant service becomes degraded, a SMS or email message may be sufficient but if a service becomes critical it may be desirable to use a telephone call to gain quicker attention.

### 2.3.7 Classification

A classification defines how serious or important the data from a plugin is, for example, the CPU load being slightly higher than normal would probably be classed as a minor issue whereas a system not responding to checks from Prophasis would be likely to be classed as a critical issue. Table 2.1 lists all the different classifications ordered by severity.

Table 2.1: Different classifications that can be given to data ordered by severity

Severity	Classification	Notes
Most severe	Critical	Only applies to services
⋮	Major	
⋮	Minor	
⋮	Degraded	
⋮	Unknown	
⋮	Ok	
Least severe	No data	Plugin has never been executed on this host

## 2.4 User Interface

The nature of a monitoring system means that it will often need to be used from mobile devices when systems need to be checked on the move or out of hours. Therefore Prophasis's web interface must work correctly on both mobile devices and conventional desktops and laptops. It was therefore decided to build a responsive web interface that will scale automatically based on the screen size of the device without any loss of functionality.

In order to keep Prophasis as intuitive as possible, the user interface should contain clear explanations of different functionality as well as help text and detailed error messages.

# Chapter 3

## Implementation

### 3.1 Technologies

The vast majority of the system is implemented in Python 3. This allows for a large variety of modules to be used during development. Python is also widely available on UNIX systems and is easy to install on machines where it is not included.

The Python "Virtual Environment" (Virtualenv) system is also extremely useful in this system. It allows all dependencies to be kept totally separate from the rest of the system, this is particularly important for the agent as it ensures that the agent cannot interfere with the Python environment of the systems it is running on.

#### 3.1.1 Flask Web Framework

The agent and web interface are both built using the Flask web framework which handles routing URLs to the correct Python functions as well as handling request data and building the correct HTTP responses. The agent purely uses the routing and request/response functionality provided by Flask whereas the web interface also uses Flask's bundled templating engine, Jinja2, to render the HTML pages that are displayed to the user. The web interface also uses the Flask-Login package to provide login and user session management functionality.

#### 3.1.2 Tornado HTTP Server

While Flask does provide a built in webserver, it is only designed for development use. In order to provide a production suitable web server for the agent. Tornado uses

the uWSGI interface provided by Flask. Tornado was chosen as it can easily be integrated directly into the Flask application and therefore does not require any sort of external webserver to be installed/configured on the system.

### 3.1.3 SQLAlchemy ORM

All database functionality in Prophasis is handled through the SQLAlchemy ORM which abstracts the database into a set of Python classes (known as models). This not only reduces development time, it also reduces the likelihood of errors as all of the database queries are generated by the library rather than being handwritten. The SQLAlchemy models file can also be shared between both the web interface and the core preventing duplication of database query logic.

### 3.1.4 Lupa

Lupa provides an interface between the system's Lua runtime and Python. It is used by Prophasis to execute the user provided Lua code that is used to classify the results of checks. A sandboxed Lua environment is configured to prevent this user provided code from performing undesired operations. Sandboxing is covered further in section 3.5.2.1.

### 3.1.5 PostgreSQL Database

Prophasis has been officially developed to support PostgreSQL databases, however through the use of an ORM it is possible to easily move to different database platforms such as MySQL or Oracle. SQLAlchemy transparently handles the difference between different database platforms when generating its queries.

### 3.1.6 Appdirs

Prophasis has been designed to support a variety of different operating systems. Different platforms have different conventions for where an application should store configuration files and other application data. Appdirs is a Python library that works out the correct path for various types of files for the detected platform. This means that files are stored in the correct directory no matter if Prophasis is installed on Linux, BSD, Windows or an other supported platform.



### 3.1.7 AdminLTE & Bootstrap

AdminLTE is an open source theme for building administration dashboards and control panels. It is MIT licensed and is therefore compatible with the MIT licence Prophasis is built under. AdminLTE is built on top of Twitter's Bootstrap framework which is well supported with a large range of plugins available. AdminLTE and Bootstrap have extensive support for responsive user interfaces and therefore very little manual work needs to be done to make the user interface responsive to work well on both desktop and mobile devices.

### 3.1.8 Handlebars.js

Handlebars.js is a very lightweight Javascript templating engine. It is used in the front-end of the web interface for rendering portions of the page that are generated after the page has loaded, for example alerts and items that are dynamically added to the DOM such as dependencies when editing services or intervals when editing schedules. This means that any HTML logic can be totally separated from the Javascript source code. Unfortunately Handlebars templates are slightly different to Jinja2 ones. For simply filling in placeholders they are compatible with one another however, more advanced functionality such as conditionals and loops require slightly different syntax. Therefore care must be taken when rendering the same template with both Jinja2 and handlebars.

## 3.2 Plugin Interface

- Explain structure of a plugin
- UML Diagram?
- Why Lua for classification logic?

## 3.3 Database

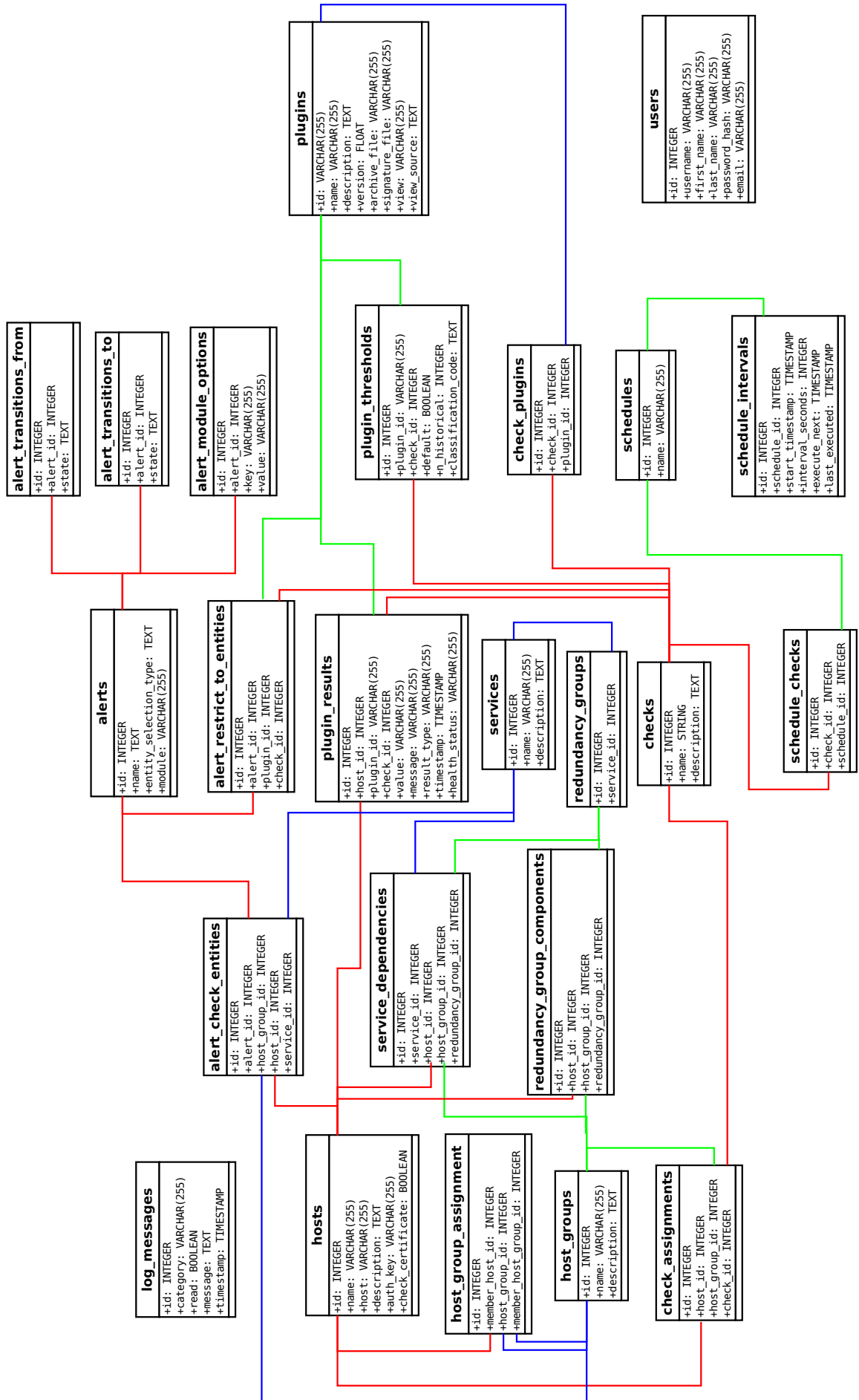
The database is at the very center of Prophasis, it is used to store the configuration of the system along with data collected from agents, it is also the primary means of communication between the core and the web interface. Prophasis was designed to

use the PostgreSQL RDBMS and great care was taken to ensure that the database design was built to be efficient and flexible enough to allow for future expansion.

The database was designed in a traditional manner using tables and relations between them in order to build an efficient and normalised design however it was implemented using the SQLAlchemy ORM which abstracts the database into a series of classes which saves time and reduces errors as queries do not need to be written manually. The manual design was carefully implemented using extensive use of SQLAlchemy's `relationship()` function. This allows relationships in the database to be traversed as easily as properties of an object. For example, if you have an entity `h` of class `Host` and an entity `r` of class `PluginResult` then it is possible to both access a list of all plugin results attributed to the host with `h.check_results` and to access, for example, the name of a host from the plugin result with `r.host.name`. The ORM handles all the logic of building the queries to gather the correct data. The ORM also makes it easy to edit data in the database, it is a simple case of retrieving the object, changing it's attributes and committing the session. It is also transactional which means that changes can be staged up and then either committed or rolled back, this is ideal as it makes it easy to abort from changes to the database in the event of an error. This also provides functionality to make multiple database operations atomic, this is important since there are multiple processes/threads writing to the same database.

Figure 3.1 shows a diagram that represents the structure of the entire database. The colours have no significance and are simply used to illustrate which lines are crossing over and which are not. This diagram was built in a tool called Dia and stored in an XML file which could be kept on version control, this meant that throughout the development process the diagram could be easily maintained to match changes to the database, in fact, the diagram was always updated before the changes were applied to the actual SQLAlchemy models. Having this diagram available was extremely useful when developing parts of the logic that utilise the database as it makes the layout much clearer than simply reading the source code for the models file or by looking at the actual, running database instance.

Figure 3.1: Diagram representing the Prophase Database Schema



## 3.4 Common Module

Even though the core and web interface are completely separate processes, they still share some common logic: namely the database models, error logging and alert module handling. It would be impractical to write this functionality twice, especially the database models as this is a huge piece of work.

The best solution to this problem was to build a "common" python module known as `prophasis_common`. This contains the database models as well as error logging and alert module handling and is required by both the core and web interface. When installing either of these the common module must be installed first. Structuring the system this way prevents duplication of logic and ensures that there is no risk of there being any differences between the code in the core and the agent.

## 3.5 Core

### 3.5.1 Dispatcher

The dispatcher is the system which sends off requests to the agent and waits for the responses back which are then classified and then stored in the database. Due to the time it may take to execute some checks it would be impractical to execute them in series, therefore the dispatcher is multi-threaded.

The Prophasis dispatcher uses Python's built in "multiprocessing" library. This provides various methods to manage processes as well as thread-safe data structures for communicating between them. The dispatcher uses `multithreading.Process` to spawn worker processes and uses `multiprocessing.Queue` to define thread safe queues for passing data into these worker processes.

**Fork vs Spawn** The multiprocessing library supports three different ways to start a process. By default on UNIX systems it uses `os.fork()` to fork the existing process. This is unsuitable in this situation as the process creates a copy of the already open database connection - this causes conflicts when the workers try to write to the database. The solution to this is to tell the multiprocessing library to use the "spawn" context (the default on Windows) which will create a fresh Python interpreter process, this prevents open handles from being carried over into the child process. Using

spawn is comparatively slower than forking the process but since we are only creating schedules are called (which is on a real world timescale), this difference will not be noticeable.

### 3.5.2 Classification

When data is collected from the agent it needs to be "classified" to determine whether it is "ok", "major", "critical".etc. Classification is done by Lua code that is bundled with the plugin and can also be modified by the user through the web interface. The Lua classification code is stored in the database for ease of modification. The "lupa" Python library is used to integrate the Lua runtimes into the Python code.

#### 3.5.2.1 Sandboxing

Classification code is executed directly on the machine under the same user as the core. Since this classification code can be changed through the Prophasis web interface it is critical that this code cannot perform malicious operations on the system. In order to resolve this, a Lua sandbox is created. This is done by creating a Lua table with only specific, trusted functions such as `math` and `ipairs` added to it. Lua's `setfenv` (set function environment) function is then called to ensure that all user provided code is executed inside this sandbox and can therefore not access more risky operating system functions such as file handling.

#### 3.5.2.2 Functions

In order to make developing classification code easier, several predefined functions are provided to handle common operations such as `arrayMax(array)` which will return the maximum value in an array and `arrayContains(array, value)` which returns a boolean defining whether the given value is in the array or not. These functions are stored in a separate Lua file and are included before the user provided classification code before it is executed.

#### 3.5.2.3 Handling Errors

When dealing with user defined code, there is always the potential for errors to occur when the classification code is executed. In this situation the system will automatically fall back and classify the result as "unknown". The error will also be logged within

Prophesis and can be easily viewed by the user in the web interface's "System Logs" area.

- Scheduling
- Multi-threaded dispatcher

## 3.6 Agent

The agent is implemented using the Flask web framework to expose a HTTPS API that the core communicates with. Requests are sent to the agent using regular HTTP GET and POST requests with information passed using URL parameters or HTTP form data respectively. Responses are formatted as JSON. The Tornado HTTP server is used to handle incoming HTTP connections and communicates with Flask through using uWSGI.

**Authentication** In order to prevent unauthorised actions being performed on the agent, the core must authenticate with every request. The agent stores a hash of a long authentication key in its configuration file. This key is generated on agent installation and is different for every agent. This ensures that if the key for one agent is obtained, an attacker cannot access every other agent on the network. Storing a hash means that if someone was able to read the agent configuration file they cannot obtain the authentication key which could have allowed them to execute code as the agent's user which may have higher privileges than their user. HTTP's basic access authentication is used which allows a username and password to be easily sent along with an HTTP request, in this situation the username is "core" and the password is the authentication key. A Python decorator (`@requires_auth`) is applied to functions that require authentication which will verify the authentication token and only allow the function to execute if the token is correct, otherwise it will respond with HTTP error 401 (Unauthorised). Using a decorator keeps authentication logic out of the body of the function which ensures that the user is authenticated before the function is even entered. This prevents errors such as the authentication check being moved after some restricted functionality or accidentally being removed during changes to the function body.

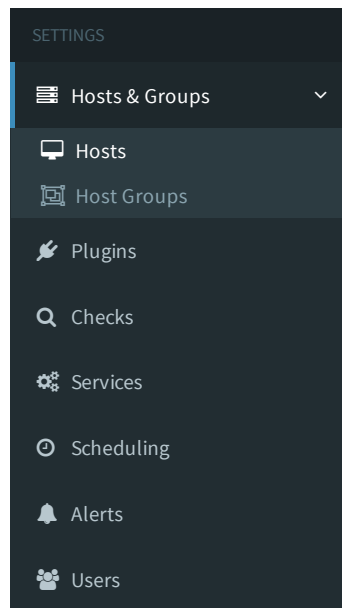
## 3.7 Web Interface

The web interface is implemented using the Flask web framework. The frontend is built using AdminLTE which in turn is based on Twitter's Bootstrap framework. The web interface is built to be fully responsive so that it performs equally as well on both desktop and mobile devices. The Jinja2 templating engine is used to render the pages from HTML templates. This allows certain pieces of template logic to be reused preventing code duplication and enforces a clear separation of logic between the processing and templating logic.

### 3.7.1 Configuration

All of the different configuration options are accessible through a clearly organised menu displayed on every page. This allows the user to access all of the different configuration options from a single location. This navigation bar can be seen in Figure 3.2.

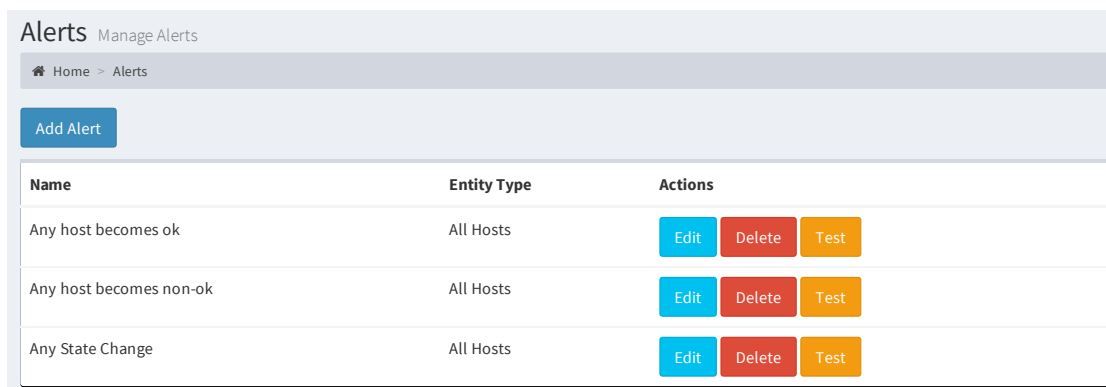
Figure 3.2: The "settings" navigation bar



Clicking each of these links will take the user to the management area for that section allowing them to configure that part of the system. The first page of any of these sections is an index page that lists all objects (e.g. plugins or alerts) in the section with options to manage each of them as well as providing a link to add new objects. An example of the index page for alerts can be seen in Figure 3.3. This page lists all alerts

in the system and provides buttons to edit, delete and test them. There is also a button allowing the user to add new alerts to the system.

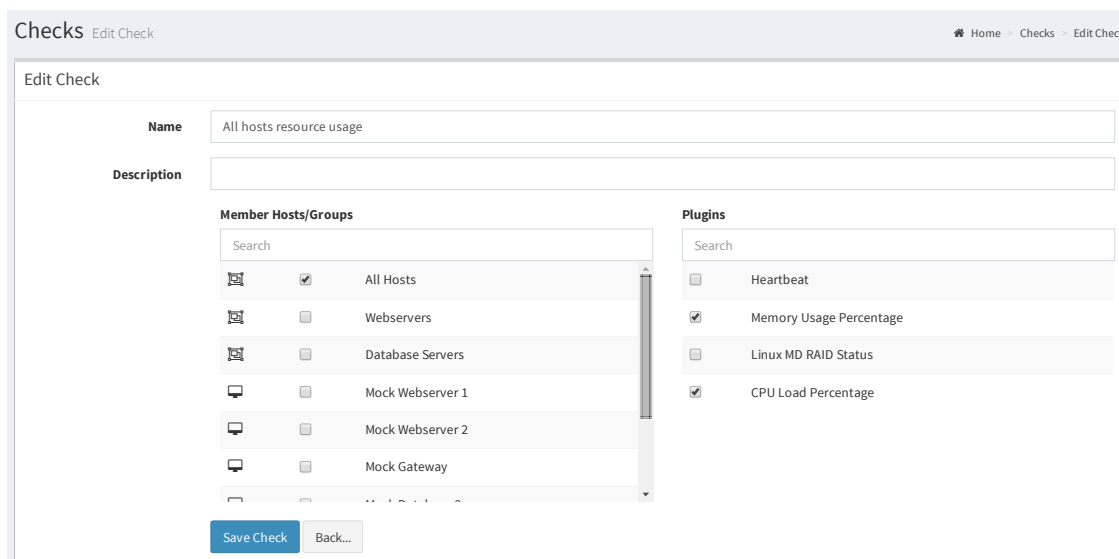
Figure 3.3: The alerts index page



Name	Entity Type	Actions
Any host becomes ok	All Hosts	<a href="#">Edit</a> <a href="#">Delete</a> <a href="#">Test</a>
Any host becomes non-ok	All Hosts	<a href="#">Edit</a> <a href="#">Delete</a> <a href="#">Test</a>
Any State Change	All Hosts	<a href="#">Edit</a> <a href="#">Delete</a> <a href="#">Test</a>

When the user chooses to add or edit an object a form is loaded. The add and edit forms are both rendered using the exact same template files ensuring that both forms are consistent. A `method` parameter is passed into these templates which is set to either "add" or "edit". This is used to adjust the template. Figure 3.4 shows the form for editing a check.

Figure 3.4: The Edit Check form



This form has a pair of text boxes to specify a name and description for the check and then a pair of lists of checkboxes: the first lists hosts and host groups that the check will be executed on and the second lists plugins that will be executed on those



hosts when the check is run. The search boxes above each of these lists will filter the list to entries that contain the search term. This search system is provided entirely in Javascript and is generic enough to be applied to any HTML table by simply specifying some classes and HTML5 data attributes. These search boxes are applied to any tables in the system which may contain a lot of items.

**Code editing** As described in Section 2.2.2.3, Lua code is used to classify the results retrieved from plugins. In order to maintain the ability to manage everything through the web interface, it needs to be possible to edit this code through the web interface. For this, the CodeMirror<sup>1</sup> editor is used which provides an editor with syntax highlighting and support for using the tab key to indent blocks of code as well as intelligently handling indentation when pressing return. The form for editing classification code can be seen in Figure 3.5.

Figure 3.5: The form for editing classification Lua code

Plugins Set Thresholds Home Plugins Set Thresholds

Default Thresholds

Number of historical values  How many previously collected values should be aggregated

Classification Code

```

1 -- The following arrays are provided which contain data collected by the plugin
2 -- values: Array of "values" returned by the plugin
3 -- messages: Array of "messages" returned by the plugin
4 -- result_types: Array with "plugin" for a successful check or the following error values:
5 --   "command_unsuccessful", "authentication_error", "request_error", "connection_error", "connection_timeout"
6 if arrayContains(values, "none") then
7   return "unknown"
8 end
9
10 maxValue = arrayMax(values)
11
12 if maxValue >= 100 then
13   return "critical"
14 elseif maxValue >= 90 then
15

```

Save Thresholds Add Custom Threshold

**Schedules** As described in Section 2.3.4, a schedule is used to define when to execute one or more checks. The interface for this includes the usual form fields for a name for and description of the schedule as well as a checkbox list of available checks to execute. This page also has additional functionality to configure the time intervals at which the schedule is run. This can be seen in Figure 3.6. Each interval has fields to specify the start time for the schedule as well as a text box and drop down menu to enter the interval period. The green "Add Interval" button will add another row to this

<sup>1</sup><https://codemirror.net/>

list allowing more intervals to be added and the red "X" button will remove the row it's in from the table. This is all handled purely in frontend Javascript.

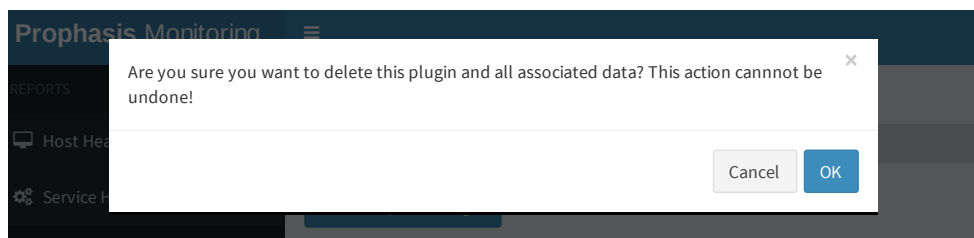
Figure 3.6: Form for managing the intervals for a schedule

Starting from	2016-03-19 00:00:00	execute every	7	Days	X
Starting from	2016-03-16 00:00:00	execute every	7	Days	X

Add Interval

In order to display confirmation messages to users, `Bootbox.js`<sup>2</sup> is used which provides a convenient abstraction around Bootstrap's build in modal functionality. This allows confirmation messages to be displayed to users without any manual HTML markup and responses to be collected straight into Javascript. An example of a confirmation message can be seen in Figure 3.7.

Figure 3.7: The confirmation message displayed when deleting a plugin



**Services** As described in Section 2.3.5, services are comprised of one or more dependencies or redundancy groups. Redundancy groups in turn are comprised of one or more hosts or host groups. Building a user interface to represent this was particularly challenging as data needed to be represented clearly and be intuitive to edit. The finished interface can be seen in Figure 3.8.

<sup>2</sup><http://bootboxjs.com/>

Figure 3.8: The user interface for editing services

In this interface, dependencies and redundancy groups are represented as boxes each containing the host(s) and host group(s) that are a member of that dependency or redundancy group. Individual hosts can be added to redundancy groups by clicking the "Add Item" button or removed by clicking the red "X" next to it. Entire dependencies can be removed by clicking the red "X" in the top right of the box. To select hosts or host groups to create a dependency or to add to a redundancy group, a Bootstrap modal dialog is displayed as shown in Figure 3.9. On this form, HTML5 data attributes are used to attach data such as IDs to the actual DOM elements that represent the structure of the service. When the form is saved, Javascript is used to serialise this structure for processing. This massively simplifies the code as adding and removing dependencies, redundancy groups, hosts.etc can be handled entirely in the DOM without having to also maintain a Javascript data structure at the same time.

Figure 3.9: The modal dialog for selecting hosts and host groups

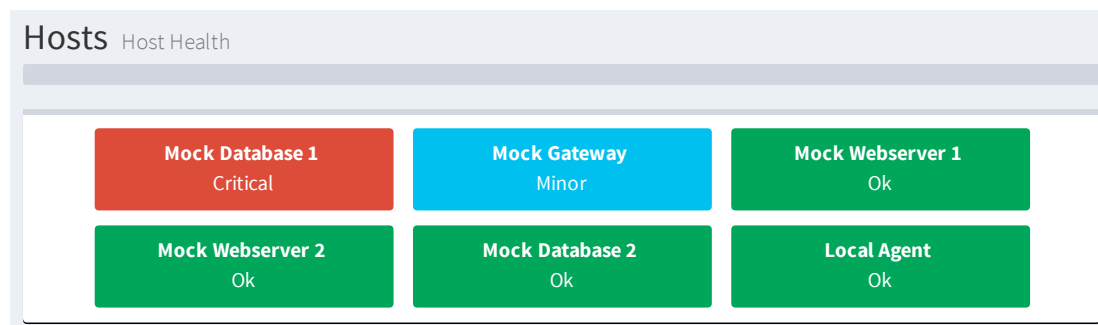
### 3.7.2 Reporting

In addition to being used to configure the system, the web interface is also used to visualise the collected data. There are currently two reports, one to show the health of all hosts in the system and another to show the health of all services. There is scope to add more reports to give different views of the data.

#### 3.7.2.1 Host Health

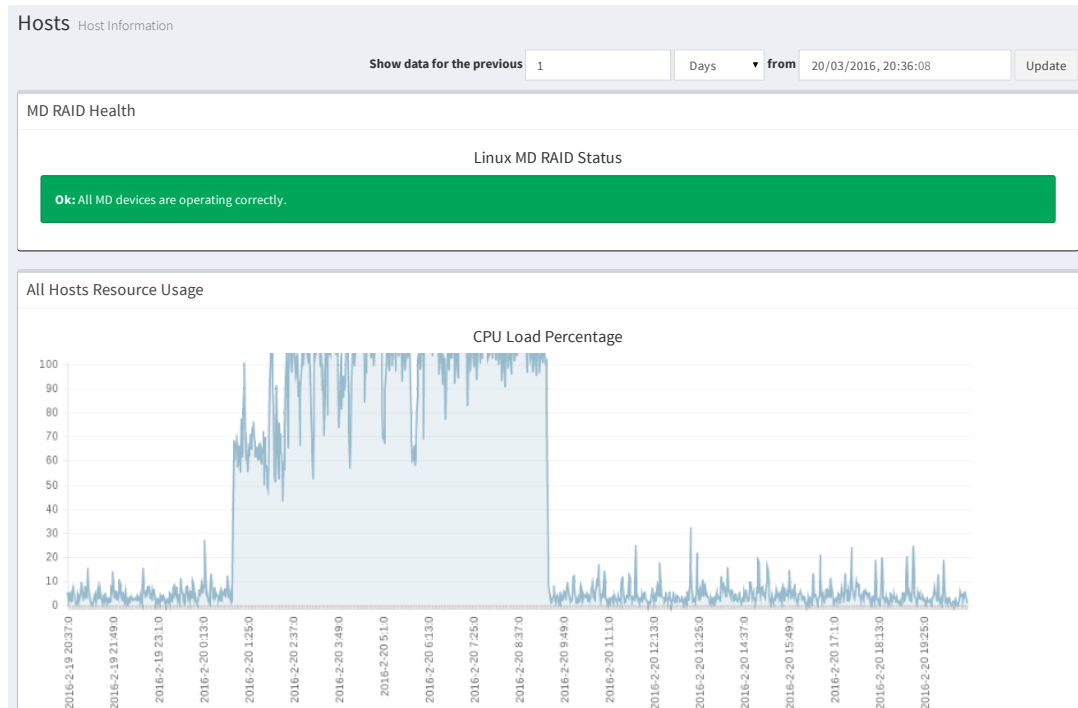
Figure 3.10 shows the host health report for a selection of machines. This report lists all the machines in the system as well as displaying their health in both a textual and colour representation. The results here are sorted in order from least to most healthy. This ensures that even in a network with a large number of systems, unhealthy hosts will not go unnoticed due to being buried deep inside a list, especially on the small screens of mobile devices.

Figure 3.10: The index page for the Host Health report



Clicking on any of these hosts will open a page displaying more detailed breakdowns of the data stored for that host. This is where the timeseries data can be visualised. Figure 3.11 shows a sample output of timeseries data collected from a running server. This shows both the status of the Linux MD RAID devices as well as a graph of CPU load showing a large spike due to a RAID consistency check that occurred on the machine. At the top of this report are a pair of inputs allowing the time period that the data represents to be specified.

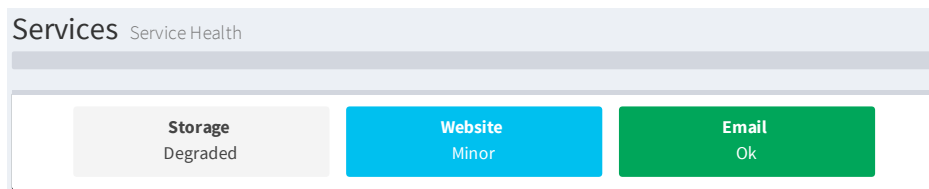
Figure 3.11: The host information page showing timeseries data for a single host



### 3.7.2.2 Service Health

In order to give a view of the overall health of the services running on a system or to show the impact that a fault with a host has on the overall functionality of a system, a report is available to list the health of all services. This is shown in Figure 3.12. Like the Host Health report, the services are ordered to show the services with the most serious condition first. As can be seen, services have an additional health status of "Degraded" which means that while a service is operating normally, one of its redundant components is experiencing an issue.

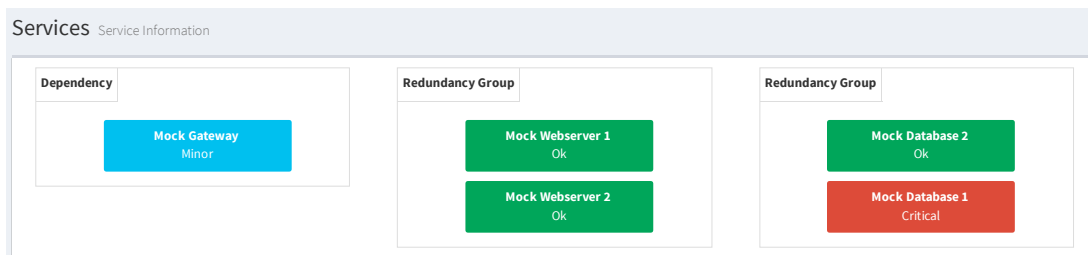
Figure 3.12: Page showing the overall health of all services in the system



Clicking on each of these services will load a view that shows the status of all hosts in the service as well as how they are structured in terms of dependencies and redundancy groups. This is shown in Figure 3.13. Clicking on each of the hosts on this page

loads the host information view as shown above. This view clearly shows the reason for a service's health status and allows issues affecting a service to be clearly seen.

Figure 3.13: Page showing the overall health of all services in the system



# Chapter 4

## Testing

### 4.1 Unit Testing

Unit tests are used to test the more complex functionality of the system such as getting all member hosts in a host group or determining the overall health of a host or service. These algorithms are somewhat complex so having unit tests is useful to both verify the functionality during development as well as to ensure this functionality is not broken by changes to other parts of the system. The Python `unittest` library is used as it included as part of a standard Python install and is well documented.

Each test case is implemented as a single class with each different test in a separate function within this. Each `TestCase` has a `setUp()` method which is called before each test in the class is executed. The `setUp()` method will drop all database tables, recreate the database structure from the models file and will then insert sample data for use in the tests. This ensures that every test starts on a clean database.

In order to prevent tests from affecting the main database or requiring a new database to be set up, SQLite is used. A temporary, in memory database is created when the tests are executed which is then destroyed when the tests are completed. The SQLAlchemy ORM completely transparently handles the differences between PostgreSQL and SQLite therefore requiring no changes to the database code for use during testing.

## 4.2 Mock Agent

When developing and testing Prophasis, it is important to be able to simulate a network of multiple machines all being monitored and in addition to this it is useful to be able to simulate faults with these machines. While it would be possible to run multiple agents and then use plugins to return data and to simulate faults this is impractical to set up, run and maintain.

As a solution to this, a mock agent was developed. This is a simple application that implements enough of the agent's functionality to allow the core to communicate with it as though it is a regular agent. It does not use plugins and instead reads the data to respond with from a JSON file stored on disk. The agent's `ping` method is implemented as normal, the `check_plugin_version` method always responds saying that an update for the plugin is not required (this means we don't need to implement the `update_plugin` method). The `get_plugin_data` method is then used to read the JSON file and respond with the appropriate data. Figure 4.1 shows an example of the JSON file used by the agent.

Figure 4.1: An example of the JSON file read by the mock agent

```
{
  "127.0.0.1:4048": {
    "me.camerongray.proj.cpu_load_percentage": {
      "value": 20, "message": ""},
    "me.camerongray.proj.memory_usage_percentage": {
      "value": 20, "message": ""}
  },
  "127.0.0.2:4048": {
    "me.camerongray.proj.cpu_load_percentage": {
      "value": 90, "message": ""},
    "me.camerongray.proj.memory_usage_percentage": {
      "value": 50, "message": ""}
  }
}
```

In order to keep the mock agent simple, it listens on a single port (4048, the same as the regular agent). The mock agent binds to all available IP addresses on the system and it uses the destination IP address to determine which data it is to respond with.



In my usage I simply ran the mock agent on the same machine as the core and used loopback IP addresses picked from 127.0.0.0/8 to add multiple mock agents to the system giving each a different IP addresses. Each of these IP addresses is used as a key in the JSON file which maps to a JSON schema containing plugin IDs mapped to the data that should be sent in response to requests for that plugin.

The JSON file is read every time a request reaches the mock agent. This means that the test data can be changed on the fly while the mock agent is running by simply changing the data in the file and saving it. For example, to test what happens if the CPU load on a machine is too high one can simply change the value in the JSON file to the new CPU load and save it, the next time the CPU load plugin is executed on that specific agent, the new CPU load will then be sent out.

- Unit testing
- Use within Tardis & Lynchpin



# **Chapter 5**

## **Evaluation**



## **Chapter 6**

## **Conclusion**