АО «Интеграл-Аналитикс»
ИНН 9701237188 КПП 770101001 ОГРН 1237700040324
Юридический адрес: 101000, г. Москва, ул. Покровка, 1/13/6, стр. 2, пом. 1, ком. 1
Тел.: 8 (499) 673-00-93
Веб-сайт: www.integral-analytics.ru
Эл.почта: info@integral-analytics.ru

Integral Analytics

Комплексная система мониторинга «ИНТЕГРАЛ»

Общее руководство по системе

Версия 01.00

Москва 2023

Оглавление

Введение

Данный документ содержит общее описание основных концепций мониторинга с применением Комплексной системы мониторинга «Интеграл».

Общее описание приведено для Системы, развернутой на сервере под управлением ОС Linux.

1 Общее описание

1.1 Назначение Системы

Комплексная система мониторинга «Интеграл» (далее – КСМ, Система) – это модульная система мониторинга, обеспечивающая мониторинг различных типов объектов мониторинга, отслеживание и контроль показателей, визуализацию получаемых данных и уведомление пользователей о сбоях в IT-инфраструктуре. Дополнительно к стандартным возможностям визуализации предусмотрена система отчетности и возможность экспорта данных в сторонние системы. Одна из ключевых особенностей – обеспечение совместимости с базой плагинов свободно распространяемого программного обеспечения Nagios и Icinga.

КСМ «Интеграл» адаптирована для работы с российской платформой виртуализации «Горизонт-ВС». Совместно с разработчиками ООО «ИЦ «Баррикады» был разработан модуль мониторинга, обеспечивающий автоматическое подключение и мониторинг всех компонентов системы виртуализации.

За счет возможностей масштабирования и расширения КСМ позволяет проводить мониторинг больших и сложных IT-инфраструктур, включая территориально распределенные инфраструктуры, частные, общедоступные или гибридные облака.

1.2 Общее описание и архитектура КСМ «Интеграл»

Компоненты КСМ разработаны на языках:

- C++ – ядро ПО;
- PHP – веб-интерфейс ПО;
- иные языки программирования (python, bash, java и т.д.) – для отдельных плагинов могут использоваться специализированные языки программирования, наиболее подходящие для выполнения поставленной задачи, конкретный перечень зависит от используемых плагинов.

Система имеет модульную архитектуру, состоящую из следующих основных компонентов (см. Рисунок 1.1):

1) компонент Ядро;
    а) ПО Ядра КСМ "Интеграл";
    б) Модуль API;
    в) Модуль IntegralDB;
    г) Подключаемые плагины.
2) компонент БД;
    а) СУБД постоянного хранения;
    б) Оперативная СУБД;
    в) Модуль IntegralDB.
3) Компонент Веб:
    а) ПО Веб-интерфейса КСМ "Интеграл";
    б) Веб-сервер (apache2 или nginx);
    в) Модуль IntegralDB.

Система позволяет интегрировать различные дополнения и расширения, что обеспечивает гибкость системы и расширяет первоначальный функционал.

Структура КСМ позволяет разворачивать распределенную систему мониторинга, в рамках которой возможно создать зависимые узлы, осуществляющие проверки и направляющие результаты на основной узел.

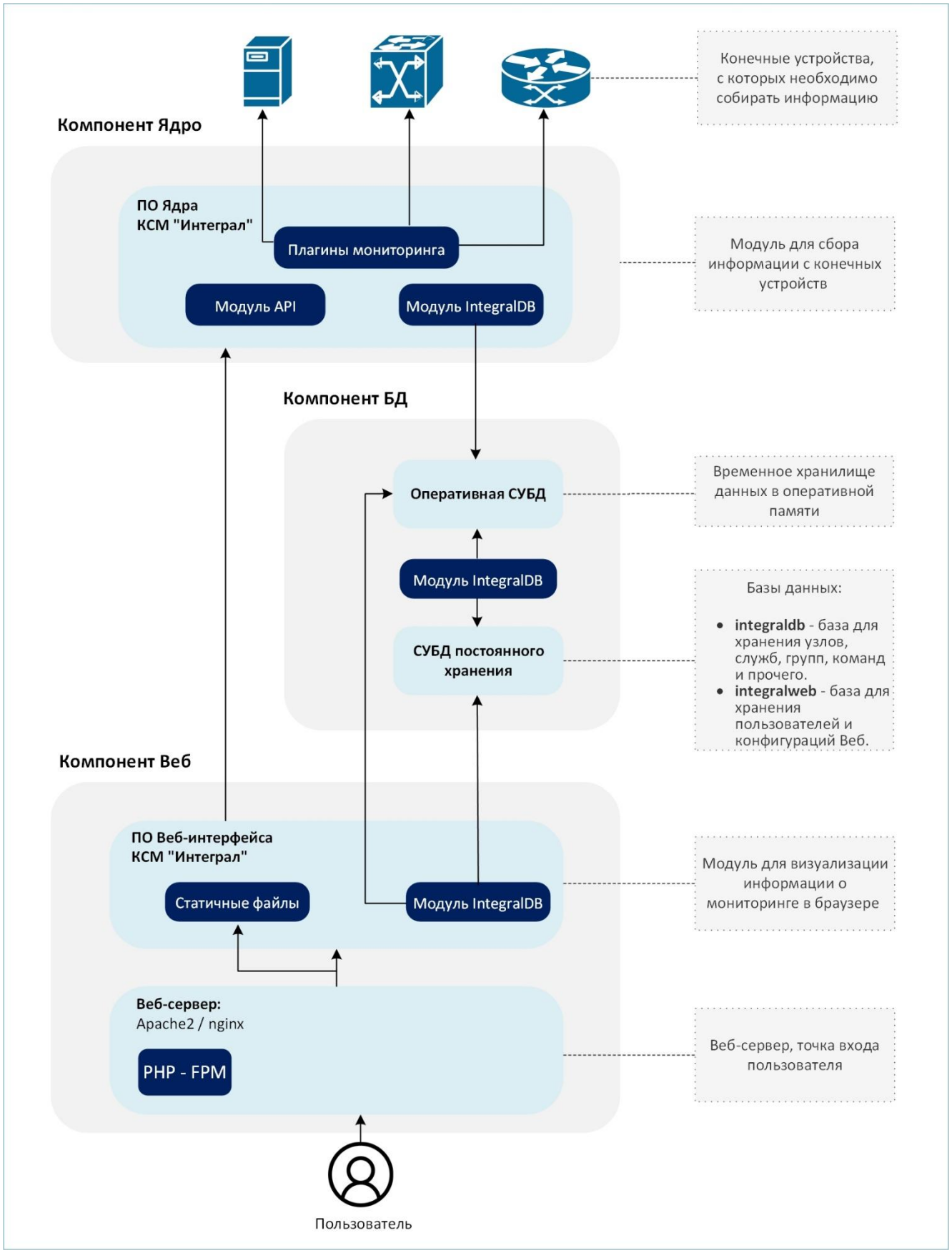

Рисунок 1.1 – Общая архитектура КСМ

Система позволяет использовать для хранения данных мониторинга и конфигурационной информации следующие типы СУБД:

- MySQL;
- PostgreSQL.

Модульный web-интерфейс в состав Системы написан на языке PHP, активно использует AJAX и предоставляет статистику в виде графиков.

В Системе для обеспечения интеграции с внешними сервисами предусмотрены следующие варианты API:

- XML;
- JSON;
- SOAP.

1.3 Обзор возможностей

Основные возможности системы:

- Мониторинг сетевых Служб (SMTP, POP3, HTTP, NNTP, Ping и т. д.);
- Мониторинг ресурсов Узла ИТ-инфраструктуры (загрузка ЦПУ, использование дисков, использование оперативной памяти);
- Мониторинг серверных компонентов (коммутаторы, маршрутизаторы, серверы, СХД, датчики температуры, влажности и т. д.);
- Простое создание плагинов, позволяющее пользователям разрабатывать собственные типы проверок Служб;
- Параллельная проверка Служб;
- Создание иерархии сетевых Узлов, позволяющее отличать нерабочие Узлы от недоступных;
- Возможность назначения обработчиков событий;
- Возможность автоматической отправки уведомлений по E-Mail, через систему мгновенного обмена сообщениям, SMS и т. д.;
- Эскалация уведомлений.

2 Используемые типы атрибутов

Для конфигурации Системы используется заданный набор типов атрибутов, приведенный в Таблице 2.1.

Внимание! Необходимо использовать только указанные типы атрибутов объекта, так как в противном случае проверка конфигурации завершится ошибкой.

Таблица 2.1 – Перечень типов атрибутов, используемых при конфигурации Системы

Тип атрибута	Пример
Число	`5`
Длительность	`1 m`
Строка	`"These are notes"`
Логический	`true`
Массив	`[ "value1", "value2" ]`
Словарь	`{ "key1" = "value1", "key2" = false }`

3 Узлы и Службы

3.1 Общее описание

В Системеа основными типами объектов мониторинга являются **Узел** (Host) и **Служба** (Service).

К Узлам и Службам относятся любые объекты ИТ-инфраструктуры, информацию о состоянии которых можно получить по сети, например:

- сетевые Службы: HTTP, SMTP, SNMP, SSH и т.д.;
- принтеры;
- коммутаторы или маршрутизаторы;
- датчики температуры;
- другие локальные или доступные по сети Службы.

Объект типа Узел предоставляет механизм для группировки объектов типа Службы, запущенных на данном Узле.

Ниже приведен пример Узла, который определяет две дочерние Службы:

```
object Host "my-server1" {
  address = "10.0.0.1"
  check_command = "hostalive"
}

object Service "ping4" {
  host_name = "my-server1"
  check_command = "ping4"
}

object Service "http" {
  host_name = "my-server1"
  check_command = "http"
}
```

В приведенном выше примере создаются две Службы `ping4` и `http`, которые принадлежат Узлу `myserver1`. В нем также указывается, что Узел должен выполнить свою собственную проверку с помощью команды `hostalive check`.

Атрибут `address` используется командами проверки для определения того, какой сетевой адрес связан с объектом Узел.

3.2 Состояния объекта Узел

Возможные состояния Узла приведены в Таблице 3.1.

Таблица 3.1 – Возможные состояния Узла

Название состояния	Описание состояния
UP	Узел работает в штатном режиме
DOWN	Узел выключен или вышел из строя

3.3 Состояния объекта Служба

Возможные состояния Службы приведены в Таблице 3.2.

Таблица 3.2 – Возможные состояния Службы

Название состояния	Описание состояния
OK	Служба работает штатно
WARNING (внимание)	Служба находится в рабочем состоянии, но испытывает некоторые проблемы
CRITICAL (критическое)	Проведенная проверка определила, что Служба находится в критическом состоянии
UNKNOWN (неизвестно)	Проверка не смогла определить состояние Службы

3.4 Таблица номеров состояний Узлов/Служб

Номера состояний для Узлов и Служб, возвращаемые при опросе объектов с помощью средств Системы, приведены в Таблице 3.3.

Таблица 3.3 – Номера состояний

Номер состояния	Состояние Узла	Состояние Службы
0	UP (работает в штатном режиме)	OK
1	UP	WARNING (внимание)
2	DOWN (выключен или вышел из строя)	CRITICAL (критическое)
3	DOWN	UNKNOWN (неизвестно)

Каждое состояние Службы имеет свой уникальный номер. Для Узла каждое состояние может отображаться одним из двух номеров. Например, состояние UP может отображаться как 0 или как 1.

3.5 Состояния HARD и SOFT

При обнаружении проблемы с Узлом/Службой Система повторно проверяет объект несколько раз (на основе настроек `max_check_attempts` и `retry_interval`) перед отправкой уведомлений. Это гарантирует, что при временных сбоях не будут отправляться ненужные уведомления.

В течение этого времени объект находится в состоянии SOFT. После того, как все повторные проверки были выполнены, а объект все еще не функционирует штатно, Узел/Служба переключается в состояние HARD и отправляются соответствующие уведомления.

Описания состояний приведены в Таблице 3.4.

Таблица 3.4 – Описание состояния объектов HARD и SOFT

Состояние	Описание
HARD	Состояние Узла/Службы в последнее время не менялось. Применяется параметр `check_interval` (период проведения проверок в минутах).

Состояние	Описание
SOFT	Узел/Служба недавно изменили состояние и повторно проверяются с помощью параметра `retry_interval` (определяет время ожидания перед повторной проверкой в минутах)

3.6 Определение состояния Узла/Службы

Определение состояния Узла/Службы проводится с помощью регулярных проверок. Например:

```
object Host "router" {
  check_command = "hostalive"
  address = "10.0.0.1"
}
```

В примере выше определение состояния объекта проводится с помощью одной из встроенных команд проверки – `hostalive`. Данная команда отправляет эхо-запросы ICMP на IP-адрес, указанный для объекта в атрибуте `address`, чтобы определить, подключен ли объект к сети.

> Примечание. Команда hostalive аналогична стандартной команде ping, но с другими пороговыми значениями по умолчанию. Обе команды используются для выполнения последовательных проверок из командной строки (CLI) с использованием протокола TCP/IP. Для организации более быстрых проверок рекомендуется использовать команды, работающие по протоколу ICMP.

В Системе доступен ряд других встроенных команд проверки. Так же пользователь может настроить собственные команды проверки.

3.7 Альтернативные варианты проверки Узла

Если Узел недоступен по протоколам ICMP, HTTP и т. д., можно использовать встроенную команду фиктивной проверки для установки состояния по умолчанию (описание функции приведено в отдельном документе).

Например:

```
object Host "dummy-host" {
  check_command = "dummy"
  vars.dummy_state = 0 // штатный режим работы
  vars.dummy_text = "Everything OK."
}
```

Данный метод также используется при отправке результатов внешней проверки.

Существует более продвинутый метод, который заключается в вычислении общего состояния на основе всех Служб.

4 Шаблоны

4.1 Общая информация о шаблонах

Одним из основных принципов взаимосвязей объектов в Системе является то, что сначала создаются шаблоны того-или иного типа объекта, а потом создаются сами объекты с привязкой к этим шаблонам. Шаблоны описывают одно или несколько значений свойств схожих для некоторого множества объектов. Для одного объекта может быть создано любое необходимое количество шаблонов.

Основное назначение шаблона – определить параметры, характерные для некоторого количества однотипных объектов, например, для группы Узлов, чтобы в дальнейшем можно было применить эти параметры к данной группе объектов.

Например, можно создать разные шаблоны для объектов типа Узел с типовыми параметрами проверки внутри каждого шаблона: в одном шаблоне проверки запускаются чаще, в другом шаблоне реже, для третьего шаблона отключена отсылка уведомлений и т.п. Важно помнить, что при определении разных значений данных для одних и тех же параметров шаблонов, к объекту в конечном итоге будет применяется параметр из последнего по списку шаблона, либо параметр, переопределенный на уровне самого объекта.

4.2 Пример использования шаблона

Шаблоны могут использоваться для применения набора одинаковых атрибутов к более чем одному объекту.

Например:

```
template Service "generic-service" {
  max_check_attempts = 3
  check_interval = 5m
  retry_interval = 1m
  enable_perfdata = true
}

apply Service "ping4" {
  import "generic-service"

  check_command = "ping4"

  assign where host.address
}

apply Service "ping6" {
  import "generic-service"

  check_command = "ping6"

  assign where host.address6
}
```

В приведённом выше примере Службы `ping4` и `ping6` наследуют свойства шаблона `generic-service`.

Объекты, а также сами шаблоны, могут импортировать произвольное количество других шаблонов. Атрибуты, унаследованные от шаблона, при необходимости могут быть переопределены непосредственно в объекте.

Также можно импортировать существующие объекты, не являющиеся шаблоном.

> Примечание. Шаблоны и объекты используют одно и то же пространство имен, поэтому невозможно определить шаблон с таким же именем, как у объекта.

4.3 Применение нескольких шаблонов к одному объекту

Рассмотрим пример применения нескольких шаблонов для создания объекта.

В качестве первого шаблона рассмотрим шаблон `web-server`. Данный шаблон будет использоваться в качестве базового шаблона для любого Узла, предоставляющего веб-сервисы. Шаблон содержит настраиваемую переменную `webserver_type`, которой в приведенном ниже примере присвоено значение `apache`. Т.к. данный шаблон является базовым, то импортируем в него шаблон `generic-host`. Импортирование данного шаблона позволяет присвоить базовому шаблону атрибут по умолчанию `check_command` и нам не требуется устанавливать его где-либо позднее:

```
template Host "web-server" {
  import "generic-host"
  vars = {
    webserver_type = "apache"
  }
}
```

В качестве второго шаблона рассмотрим шаблон Узла `wp-server`. Шаблон используется для Узлов `Wordpress`. В примере для данного шаблона устанавливается значение настраиваемой переменной `application_type`.

Для функции `vars` указан оператор «+=», который добавляет элементы словаря, но не переопределяет предыдущие атрибуты данной функции.

```
template Host "wp-server" {
  vars += {
    application_type = "wordpress"
  }
}
```

Конечный Узел импортирует оба шаблона. Здесь важен порядок: сначала к объекту добавляется базовый шаблон `web-server`, затем добавляется объект `wp-server`, из которого импортируются дополнительные атрибуты:

```
object Host "wp.example.com" {
  import "web-server"
  import "wp-server"

  address = "192.168.56.200"
}
```

Если необходимо переопределить отдельные атрибуты, унаследованные от шаблонов, то это можно сделать непосредственно в конечном Узле:

```
object Host "wp1.example.com" {
  import "web-server"
  import "wp-server"

  vars.webserver_type = "nginx" //переопределяет атрибут из базового шаблона

  address = "192.168.56.201"
}
```

5 Настраиваемые переменные

5.1 Общая информация

В дополнение к встроенным атрибутам объекта пользователь можете определить свои собственные настраиваемые переменные внутри функции `vars`.

В приведенном ниже примере ключ `ssh_port` указывается как настраиваемая переменная, которой присваивается целочисленное значение:

```
object Host "localhost" {
  check_command = "ssh"
  vars.ssh_port = 2222
}
```

Функция `vars` – это словарь, в котором можно задать определенные значения для ключей. В приведенном выше примере используется вариант более короткого синтаксиса индексатора.

Так же для определения значений можно использовать следующие варианты записи:

```
  vars = {
    ssh_port = 2222
  }
```

или:

```
  vars["ssh_port"] = 2222
```

5.2 Значения настраиваемых переменных

Допустимые значения для настраиваемых переменных включают:

- Строки, числа и логические значения.
- Массивы и словари
- Функции.

Также можно устанавливать типы данных с вложенностью, такие как словари в словарях.

В приведённом ниже примере настраиваемая переменная `disks` определяется как словарь. Первый ключ `disk /`словаря `disks` устанавливается как словарь с одной парой ключ-значение.

```
  vars.disks["disk /"] = {
    disk_partitions = "/"
  }
```

Приведенное выше выражение можно записать в виде развернутой стандартной структуры следующим образом:

```
  vars = {
    disks = {
      "disk /" = {
        disk_partitions = "/"
      }
    }
  }
```

Не забывайте об этом, когда обращаетесь к определенным вложенным ключам при установке правил или использовании функций.

Рассмотрим ещё один пример использования вложенных словарей в конфигурациях:

```
  vars.notification["mail"] = {
    groups = [ "integraladmins" ]
  }
```

В данном примере настраиваемая переменная `notification` определяется как словарь с ключом `mail`. Её значением является словарь ключей `groups`, который сам по себе имеет массив в качестве значения.

> Примечание: Этот массив точно такой же, как и атрибут `user_groups` для правил применения уведомлений.

```
  vars.notification = {
    mail = {
      groups = [
        " integraladmins"
      ]
    }
  }
```

5.3 Использование функций в качестве настраиваемых переменных

Система позволяет указывать функции в качестве настраиваемых переменных.

При этом, всякий раз, когда Системе требуется значение для такой пользовательской переменной, она запускает функцию и использует любое значение, возвращаемое данной функцией.

```
 object CheckCommand "random-value" {
  command = [ PluginDir + "/check_dummy", "0", "$text$" ]

  vars.text = {{ Math.random() * 100 }}
}
```

В приведенном выше примере используется сокращенный лямбда-синтаксис.

Функции имеют доступ к ряду переменных. Доступные переменные приведены в Таблице 5.1.

Таблица 5.1 – Описание переменных

Переменная	Описание
user	Объект типа Пользователь (для уведомлений)
service	Объект типа Служба (для проверок/уведомлений/обработчиков событий)
host	Объект типа Узел
command	Объект типа Команда (например, объект CheckCommand для проверок)

Например:

```
vars.text = {{ host.check_interval }}
```

В дополнение к этим переменным функция макроса может использоваться для получения значения произвольного выражения макроса:

```
vars.text = {{
  if (macro("$address$") == "127.0.0.1") {
    log("Running a check for localhost!")
  }

  return "Some text"
}}
```

Функция `resolve_arguments` может использоваться для разрешения Команды и ее аргументов схожим образом, как это происходит для атрибутов `command` и `arguments` для Команд. Команда `by_ssh` использует данную функциональность, чтобы обеспечить возможность пользователям указывать Команду и аргументы, которые должны выполняться через SSH:

```
arguments = {
  "-C" = {{
    var command = macro("$by_ssh_command$")
    var arguments = macro("$by_ssh_arguments$")

    if (typeof(command) == String && !arguments) {
      return command
    }

    var escaped_args = []
    for (arg in resolve_arguments(command, arguments)) {
      escaped_args.add(escape_shell_arg(arg))
    }
    return escaped_args.join(" ")
  }}
  ...
}
```

Получение доступа к атрибутам объекта внутри функций на этапе выполнения работы программы приведены в другом документе.

6 Макросы, работающие на этапе выполнения работы программы – Runtime-макросы

6.1 Общее описание и пример Runtime-макроса

Макросы могут использоваться для получения доступа к атрибутам других объектов на этапе выполнения работы программы. Например, они используются в командах, чтобы установить, по какому IP-адресу должна выполняться проверка:

```
object CheckCommand "my-ping" {
  command = [ PluginDir + "/check_ping" ]

  arguments = {
    "-H" = "$ping_address$"
    "-w" = "$ping_wrta$,$ping_wpl$%"
    "-c" = "$ping_crta$,$ping_cpl$%"
    "-p" = "$ping_packets$"
  }

  // Получение значения из атрибута Узла или настраиваемой переменной
  vars.ping_address = "$address$"

  // значение по умолчанию
  vars.ping_wrta = 100
  vars.ping_wpl = 5

  vars.ping_crta = 250
  vars.ping_cpl = 10

  vars.ping_packets = 5
}

object Host "router" {
  check_command = "my-ping"
  address = "10.0.0.1"
}
```

В приведенном выше примере был использован макрос `$address$` для обращения к атрибуту Узла `address`. Так же можно напрямую обращаться к настраиваемым переменным, например, используя макрос `$ping_wrta$`. Система автоматически пытается найти наиболее близкое соответствие указанному атрибуту Детальные правила приведены в следующих разделах.

> Примечание. При использовании знака `$` в качестве одиночного символа, его необходимо экранировать дополнительным символом доллара – `($$)`.

6.2 Порядок/очерёдность определения/поиска

Для поиска макросов и их соответствующих значений Система при выполнении команд проверяет объекты в следующем порядке:

1) Объект типа Пользователь (только для уведомлений);
2) Объект типа Служба;

3) Объект типа Узел;
4) Объект типа Команда;
5) Объект типа Глобальная настраиваемая переменная в константе `vars`.

Данный порядок выполнения проверки используется для установки значения по умолчанию для настраиваемых переменных в пользовательских командных объектах.

Ниже приведен пример перезаписи настраиваемой переменной `ping_packets` из предыдущего примера (см. п.6.1):

```
object Service "ping" {
  host_name = "localhost"
  check_command = "my-ping"

  vars.ping_packets = 10 // Перезаписывает значение по умолчанию 5,
прописанное в команде
}
```

Если настраиваемая переменная нигде не установлена, то используется пустое значение и в журнал Системы записывается предупреждение.

Также можно напрямую ссылаться на определенный атрибут, указав полное имя атрибута. Таким образом игнорируется заданный порядок определения/поиска:

```
$service.vars.ping_wrta$
```

В приведенном примере извлекается значение настраиваемой переменной `ping_write` для Службы. Если у Службы нет такой настраиваемой переменной, то возвращается пустое значение, независимо от того, есть ли такой же атрибут у объекта другого типа – например у Узла.

6.3 Макросы для Узлов, выполняемые во время работы программы

В Таблице 6.1 приведены настраиваемые переменные Узла, которые могут быть использованы во всех командах, выполняемых для Узлов или Служб.

Таблица 6.1 – Перечень настраиваемых переменных для объекта мониторинга с типом Узел

№	Имя настраиваемой переменной	Описание
1	host.name	Имя Узла
2	host.display_name	Значение атрибута `display_name`
3	host.state	Текущее состояние Узла. Возможные значения: - `UNREACHABLE` (недоступен); - `UP` (работает в штатном режиме) - `DOWN` (выключен или вышел из строя).
4	host.state_id	Идентификатор текущего состояния Узла. Возможные значения: - `0` (работает в штатном режиме); - `1` (выключен или вышел из строя); - `2` (недоступен).
5	host.state_type	Тип текущего состояния Узла. Возможные значения:

№	Имя настраиваемой переменной	Описание
		- SOFT; - HARD.
6	host.check_attempt	Номер текущей попытки проверки
7	host.max_check_attempts	Максимальное количество проверок, которые выполняются перед переходом Узла в состояние с типом HARD
8	host.last_state	Предыдущее состояние Узла. Возможные значения: - UNREACHABLE (недоступен); - UP (работает в штатном режиме) - DOWN (выключен или вышел из строя).
9	host.last_state_id	Идентификатор предыдущего состояния Узла. Возможные значения: - 0 (работает в штатном режиме); - 1 (выключен или вышел из строя); - 2 (недоступен).
10	host.last_state_type	Тип предыдущего состояния Узла. Возможные значения: - SOFT; - HARD.
11	host.last_state_change	Временная метка последнего изменения состояния Узла
12	host.downtime_depth	Количество Узлов в режиме обслуживания
13	host.duration_sec	Время с момента последнего изменения состояния
14	host.latency	Задержка проверки Узла
15	host.execution_time	Время выполнения проверки Узла
16	host.output	Результаты последней проверки
17	host.perfdata	Данные о производительности последней проверки
18	host.last_check	Временная метка выполнения последней проверки
19	host.check_source	Экземпляр системы мониторинга, который выполнил последнюю проверку
20	host.num_services	Количество Служб, связанных с Узлом
21	host.num_services_ok	Количество Служб, связанных с Узлом, которые находятся в состоянии OK (штатное)
22	host.num_services_warning	Количество Служб, связанных с Узлом,, находящихся в состоянии WARNING (внимание)
23	host.num_services_unknown	Количество Служб, связанных с Узлом, которые находятся в состоянии UNKNOWN (неизвестно)
24	host.num_services_critical	Количество Служб, связанных с Узлом, которые находятся в состоянии CRITICAL (критическое)

В дополнение к приведенным в таблице runtime-макросам также можно получить доступ к атрибутам объекта типа Узел.

6.4 Макросы для Служб, выполняемые во время работы программы

В Таблице 6.2 приведены настраиваемые переменные Службы, которые могут быть использованы во всех командах, выполняемых для Служб.

Таблица 6.2 – Перечень настраиваемых переменных для объекта мониторинга с типом Служба

№	Имя настраиваемой переменной	Описание
1	service.name	Краткое имя Службы
2	service.display_name	Значение атрибута `display_name`
3	service.check_command	Краткое имя команды вместе с любыми аргументами, которые будут использоваться для проверки
4	service.state	Текущее состояние Службы. Возможные значения: - OK (штатное); - WARNING (внимание); - CRITICAL (критическое); - UNKNOWN (неизвестно).
5	service.state_id	Идентификатор текущего состояния Службы. Возможные значения: - 0 (штатное); - 1 (внимание); - 2 (критическое); - 3 (неизвестно)
6	service.state_type	Тип текущего состояния Службы. Возможные значения: - SOFT; - HARD
7	service.check_attempt	Номер текущей проверки
8	service.max_check_attempts	Максимальное количество проверок, которые выполняются перед переходом в состояние HARD
9	service.last_state	Предыдущее состояние Службы. Возможные значения: - OK (штатное); - WARNING (внимание); - CRITICAL (критическое); - UNKNOWN (неизвестно).
10	service.last_state_id	Идентификатор предыдущего состояния Службы. Возможные значения: - 0 (штатное); - 1 (внимание); - 2 (критическое); - 3 (неизвестно)
11	service.last_state_type	Тип предыдущего состояния Службы. Возможные значения: - SOFT; - HARD
12	service.last_state_change	Временная метка последнего изменения состояния.
13	service.downtime_depth	Количество Служб в режиме обслуживания
14	service.duration_sec	Время с момента последнего изменения состояния
15	service.latency	Задержка проверки Службы
16	service.execution_time	Время выполнения проверки
17	service.output	Результаты последней проверки
18	service.perfdata	Данные о производительности последней проверки
19	service.last_check	Временная метка выполнения последней проверки
20	service.check_source	Экземпляр системы мониторинга, который выполнил последнюю проверку

В дополнение к приведенным в таблице runtime-макросам также можно получить доступ к атрибутам объекта типа Служба.

6.5 Макросы для Команд, выполняемые во время работы программы

В Таблице 6.3 приведены настраиваемые переменные, которые могут быть использованы во всех Командах.

Таблица 6.3 – Перечень настраиваемых переменных для объектов мониторинга типа Команда

№	Имя настраиваемой переменной	Описание
1	command.name	Название Команды

6.6 Макросы для Пользователя, выполняемые во время работы программы

В Таблице 6.4 приведены настраиваемые переменные, которые могут быть использованы во всех командах, выполняемых для Пользователей.

Таблица 6.4 – Перечень настраиваемых переменных для объектов мониторинга типа Пользователь

№	Имя настраиваемой переменной	Описание
1	user.name	Имя Пользователя
2	user.display_name	Значение атрибута `display_name`

В дополнение к приведенным в таблице runtime-макросам, также можно получить доступ к атрибутам объекта типа Пользователь.

6.7 Макросы для Уведомлений, выполняемые во время работы программы

В Таблице 6.5 приведены настраиваемые переменные, которые могут быть использованы во всех командах, выполняемых для Уведомлений.

Таблица 6.5 – Перечень настраиваемых переменных для команд для объектов мониторинга типа Уведомление

№	Имя настраиваемой переменной	Описание
1	notification.type	Тип уведомления.
2	notification.author	Автор комментария уведомления, если он существует.
3	notification.comment	Комментарий к уведомлению, если он существует.

В дополнение к приведенным в таблице runtime-макросам также можно получить доступ к атрибутам объекта типа Уведомление.

6.8 Глобальные макросы, выполняемые во время работы программы

В Таблице 6.6 приведены макросы, которые предоставляют информацию о датах и времени и доступны для всех исполняемых команд.

Таблица 6.6 – Перечень глобальных макросов, предоставляющих информацию о датах и времени

№	Имя макроса	Описание
1	icinga.timet	Текущая временная метка UNIX
2	icinga.long_date_time	Текущая дата и время, включая информацию о часовом поясе. Пример: `2014-01-03 11:23:08 +0000`
3	icinga.short_date_time	Текущая дата и время. Пример: `2014-01-03 11:23:08`
4	icinga.date	Текущая дата. Пример: `2014-01-03`
5	icinga.time	Текущее время, включая информацию о часовом поясе. Пример: `11:23:08 +0000`
6	icinga.uptime	Текущее время безотказной работы процесса в Системе.

В Таблице 6.7 приведены макросы, которые предоставляют статистическую информацию и доступны для всех исполняемых команд.

Таблица 6.7 – Перечень глобальных макросов, предоставляющих статистическую информацию

№	Имя макроса	Описание
1	icinga.num_services_ok	Текущее количество Служб в состоянии `OK` (штатное)
2	icinga.num_services_warning	Текущее количество Служб в состоянии `WARNING` (внимание)
3	icinga.num_services_critical	Текущее количество Служб в состоянии `CRITICAL` (критично)
4	icinga.num_services_unknown	Текущее количество Служб в состоянии `UNKNOWN` (неизвестно)
5	icinga.num_services_pending	Текущее количество Служб в состоянии ожидания
6	icinga.num_services_unreachable	Текущее количество недоступных Служб
7	icinga.num_services_flapping	Текущее количество Служб в состоянии флаппинга (flapping)
8	icinga.num_services_in_downtime	Текущее количество Служб, находящихся в режиме обслуживания
9	icinga.num_services_acknowledged	Текущее количество подтвержденных проблем в работе Служб
10	icinga.num_hosts_up	Текущее количество Узлов в состоянии UP (работает в штатном режиме)
11	icinga.num_hosts_down	Текущее количество Узлов в состоянии DOWN (выключен или вышел из строя)
12	icinga.num_hosts_unreachable	Текущее количество Узлов в состоянии UNREACHABLE (недоступен)
13	icinga.num_hosts_pending	Текущее количество Узлов в состоянии ожидания
14	icinga.num_hosts_flapping	Текущее количество Узлов в состоянии флаппинга (flapping)
15	icinga.num_hosts_in_downtime	Текущее количество Узлов, находящихся в режиме обслуживания

№	Имя макроса	Описание
16	icinga.num_hosts_acknowledged	Текущее количество подтвержденных проблем в работе Узлов

7 Установка правил

7.1 Общее описание правил

Некоторые типы объектов требуют наличия связи с другими типами объектов, например, Служба, Уведомление, Зависимость, Запланированное Обслуживание (Scheduleddowntime). Объектные отношения задокументированы ниже в соответствующих разделах.

Например, при создании объекта типа Служба, необходимо указать атрибут `host_name` и сослаться на атрибут существующего Узла:

```
object Service "ping4" {
  check_command = "ping4"
  host_name = "agent1.localdomain"
}
```

Для управления большим количеством конфигураций, которые имеют общий признак, который можно использовать для обращения и управления ими как группой, предпочтительным является использование функционала установки правил.

Например, если нужно обеспечить базовый мониторинг для всех Узлов ИТ-инфраструктуры, то можно установить правило, настраивающее Службу `ping4` для всех Узлов, у которых указан атрибут `address`. Достаточно установить одно правило для 1000 Узлов вместо создания 1000 объектов типа Служба с ручным указанием атрибута. Правило автоматически сгенерирует за вас объекты и заполнит соответствующие атрибуты.

```
apply Service "ping4" {
  check_command = "ping4"
  assign where host.address
}
```

Дополнительная информация по применению условия "assign where" приведена в п.7.4.

7.2 Установка правил: Предварительная подготовка

Прежде чем приступить к установке правил, необходимо учесть следующие моменты:

- Определить оптимальный вариант группировки:
  - набора уникальных настраиваемых переменных для рассматриваемых Узлов/Служб;
  - членство в соответствующих группах, например, рассматриваемый Узел является членом группы Узлов, для которых должен быть установлен заданный набор Служб;
  - единая структура или общий признак в имени Узла/Службы;
  - наличие нескольких выражений, объединённых операторами `&&` или `||` .
- Все выражения должны возвращать логическое значение (например, пустая строка приравнивается к 0 (ложь (false)).

Более конкретные требования к типам объектов описаны в следующих главах:

- Применение Служб к Узлам – см. п. 7.5.
- Применение Уведомлений к Узлам и Службам – см. п.7.6.

- Применение Зависимостей к Узлам и Службам – см. п.7.7.
- Применение Запланированного Обслуживания к Узлам и Службам – см. п. 7.8.

7.3 Установка правил: Примеры использования

Можно установить/перезаписать атрибуты объекта посредством правил, используя доступные в пределах этого правила объекты (объекты Узла и/или Службы):

```
vars.application_type = host.vars.application_type
```

Настраиваемые переменные также могут хранить вложенные словари и массивы. Переменные данного вида можно использовать не только в качестве признака для обращения при установке правил, но и в качестве значений, которые будут присваиваться объектам, созданным в результате установленных Правил.

Рассмотрим на примере применения настраиваемых переменных из п. 5.2:

```
  vars.notification["mail"] = {
    groups = [ "integraladmins" ]
  }
```

В данном примере можно сделать следующее:

- Проверить существование настраиваемой переменной `notification` и ее вложенного словарного ключа `mail`. Если возвращаемое логическое значение 1 (истина), то будет создан объект типа Уведомление.
- Присвоить значение ключа `groups` атрибуту `user_groups`.

```
apply Notification "mail-integraladmin" to Host {
  [...]

  user_groups = host.vars.notification.mail.groups

  assign where host.vars.notification.mail
}
```

Более сложный пример — использование правил применения в цикле для массивов или словарей, предоставляемых настраиваемыми атрибутами или группами.

Рассмотрим на примере применения настраиваемых переменных из п. 5.2:

```
  vars.disks["disk /"] = {
    disk_partitions = "/"
  }
```

В данном случае можно выполнить проход по всем ключам словаря, определенным в атрибуте `disk`. Так же можно использовать получаемые из словаря значения для указания дополнительных атрибутов объекта.

```
apply Service for (disk => config in host.vars.disks) {
  [...]
```

```
  vars.disk_partitions = config.disk_partitions
}
```

Более подробная информация приведена ниже – см. п. 7.9.

> Примечание. Создание конфигураций в подобном динамическом режиме требует подробной информации о сгенерированных объектах. Рекомендуется использовать команду CLI `object list` для получения списка объектов после успешной проверки конфигурации.

7.4 Применение выражений из правил

Можно использовать простые или продвинутые комбинации из выражений правил применения. Каждое выражение должно при оценке возвращать логическое значение `true/false`. Пустая строка будет, например, интерпретирована как `false`. Аналогичным образом неопределенные атрибуты буду возвращать значение `false`.

Например, выражение ниже возвращает значение `false`:

```
assign where host.vars.attribute_does_not_exist
```

Несколько условий в одном `assign where` будут работать через логическое ИЛИ (`or`).

Можно комбинировать несколько выражений для обращения к конкретному подмножеству объектов. В отдельных случаях необходимо добавить более одного выражения присваивания/игнорирования (assign/ignore where) для обеспечения соответствия определенному условию. Для таких случаев можно использовать логические операторы И (`and`) и ИЛИ (`or`).

7.4.1 Примеры использования выражений из Правил

Рассмотрим пример применения выражения из правил.

Назначить Службу определенному Узлу в массиве группы узлов с помощью оператора «in»:

```
assign where "hostgroup-dev" in host.groups
```

Назначить объект, когда настраиваемая переменная равна значению:

```
assign where host.vars.application_type == "database"

assign where service.vars.sms_notify == true
```

Назначить объект, если словарь содержит заданный ключ:

```
assign where host.vars.app_dict.contains("app")
```

Сопоставить имя Узла (используя не чувствительное к регистру сравнение):

```
assign where match("webserver*", host.name)
```

Сопоставить имя Узла с помощью регулярного выражения (обратить внимание на экранированный символ обратной косой черты – `\\`):

```
assign where regex("^webserver-[\\d+]", host.name)
```

Результаты сопоставления должны содержать все варианты использования маски *mysql* в имени Узла и (&&) настраиваемая переменная prod_mysql_db должна соответствовать маске db-*. Следует игнорировать все Узлы, у которых для настраиваемой переменной test_server установлено значение true. Так же следует игнорировать любое имя Узла, оканчивающееся маской *internal.

```
object HostGroup "mysql-server" {
  display_name = "MySQL Server"

  assign where match("*mysql*", host.name) && match("db-*",
host.vars.prod_mysql_db)
  ignore where host.vars.test_server == true
  ignore where match("*internal", host.name)
}
```

Ниже приведен аналогичный пример для применения правил расширенной фильтрации Уведомлений:

- Уведомления отправляются пользователю, если в карточке пользователя атрибут Службы notes соответствует строке "has gold support 24x7" (режим поддержки 24x7) **и** (and) выполняется одно из двух условий:
  - настраиваемой переменной Узла customer присвоено значение customer-xy (у пользователя определённое имя);

    или (or)
  - настраиваемой переменной Узла always_notify присвоено значение true (в карточке включен режим "уведомлять всегда").
- Уведомления игнорируются для Служб, имя Узла которых оканчивается шаблоном *internal **или** (or) настраиваемая переменная priority меньше 2.

```
template Notification "cust-xy-notification" {
  users = [ "noc-xy", "mgmt-xy" ]
  command = "mail-service-notification"
}

apply Notification "notify-cust-xy-mysql" to Service {
  import "cust-xy-notification"

  assign where match("*has gold support 24x7*", service.notes) &&
(host.vars.customer == "customer-xy" || host.vars.always_notify == true)
  ignore where match("*internal", host.name) || (service.vars.priority < 2
&& host.vars.is_clustered == true)
}
```

Более продвинутые примеры приведены в отдельном документе.

7.5 Применение Службы к Узлам

В данном примере применение Службы `ssh` создает соответствующий объект типа Служба всем Узлам, где задано значение атрибута `address` и настраиваемая переменная `os` в `vars` установлена как `Linux`.

```
apply Service "ssh" {
  import "generic-service"

  check_command = "ssh"

  assign where host.address && host.vars.os == "Linux"
}
```

Другие подробные примеры приведены в соответствующих главах, например, применение Служб с настраиваемыми аргументами Команд.

7.6 Применение Уведомления к Узлам и Службам

Уведомления применяются аналогичным образом к заданным объектам, Узлам или Службам:

```
apply Notification "mail-noc" to Service {
  import "mail-service-notification"

  user_groups = [ "noc" ]

  assign where host.vars.notification.mail
}
```

В приведенном выше примере Уведомление `mail-noc` будет создано как объект для всех Служб, у которых установлена настраиваемая переменная `notification.mail`. Команда для Уведомления установлена как `mail-service-notification`, и все члены группы пользователей `noc` получат уведомление.

В общем случае также можно применять шаблон Уведомления и динамически перезаписывать значения в шаблоне, проверяя наличие настраиваемых переменных. Этого можно достичь с помощью условных операторов:

```
apply Notification "host-mail-noc" to Host {
  import "mail-host-notification"

  // заменить интервал уведомления, наследованный от шаблона `mail-host-
notification` новым интервалом уведомления, заданным в настраиваемой
переменной Узла
  if (host.vars.notification_interval) {
    interval = host.vars.notification_interval
  }

  // также с периодом уведомления
  if (host.vars.notification_period) {
    period = host.vars.notification_period
  }
```

```
  // посылать смс вместо е-мейла, если настраиваемая переменная
`notification_type` установлена как `sms`
  if (host.vars.notification_type == "sms") {
    command = "sms-host-notification"
  } else {
    command = "mail-host-notification"
  }

  user_groups = [ "noc" ]

  assign where host.address
}
```

В приведенном выше примере шаблон Уведомления `mail-host-notification` содержит все соответствующие настройки Уведомлений. Правило применения применяется ко всем объектам Узла, для которых задан `host.address`.

Если объект типа Узел содержит определенный набор настраиваемых переменных, их значения наследуются и доступны локально для объекта типа Уведомления, например, `host.vars.notification_interval`, `host.vars.notification_period` и `host.vars.notification_type`. В данном примере настраиваемые переменные перезаписывают атрибуты, уже указанные в импортированном шаблоне `mail-host-notification`.

Соответствующий Узел может выглядеть следующим образом:

```
object Host "host1" {
  import "host-linux-prod"
  display_name = "host1"
  address = "192.168.1.50"
  vars.notification_interval = 1h
  vars.notification_period = "24x7"
  vars.notification_type = "sms"
}
```

7.7 Применение Зависимости к Узлам и Службам

Подробные примеры применения Зависимости к Узлам и Службам приведены в главе о Зависимостях.

7.8 Применение Повторяющегося обслуживания к Узлам и Службам

Настройка конфигурации режима Повторяющегося обслуживания производится в файле конфигурации downtimes.conf, который описан в отдельном документе, посвященном конфигурации Системы.

7.9 Применение циклов для установки правил

Помимо стандартного способа применения правил, существует возможность установки правил на основе наборов переменных (массив или словарь) с использованием цикла «for».

Возьмем следующий пример: Узел предоставляет SNMP OIDs для проведения различных типов проверки Служб. Ниже приведен пример такого представления:

```
object Host "router-v6" {
  check_command = "hostalive"
  address6 = "2001:db8:1234::42"

  vars.oids["if01"] = "1.1.1.1.1"
  vars.oids["temp"] = "1.1.1.1.2"
  vars.oids["bgp"] = "1.1.1.1.5"
}
```

Пусть требуется создать объекты для Служб `if01` и `temp`, но не для `bgp`. Значение OID следует использовать в качестве настраиваемой переменной Службы `snmp_oid`. Это аргумент, необходимый для проверки объекта по данным, получаемым по протоколу SNMP. Атрибуту `Display_name` Службы должно быть присвоено значение идентификатора внутри словаря, например: `if01`.

```
apply Service for (identifier => oid in host.vars.oids) {
  check_command = "snmp"
  display_name = identifier
  vars.snmp_oid = oid

  ignore where identifier == "bgp" // не создавать службу для проверки bgp
}
```

Система применяет правило `apply for` для всех объектов с установленной настраиваемой переменной `oids`. Система перебирает все элементы словаря внутри цикла `for` и проверяет условия в выражениях `assign/ignore where`. При необходимости можно получить доступ к переменной внутри цикла в данных выражениях, например, для того, чтобы игнорировать определенные значения.

В данном примере игнорируется идентификатор `bgp`. Это позволяет избежать создания нежелательных Служб. Другой подход заключается в том, чтобы сопоставить значение OID, например, с шаблоном в виде регулярного выражения или подстановочного знака соответствия (regex/wildcard).

```
  ignore where regex("^\d.\d.\d.\d.5$", oid)
```

> **Примечание**. В данном случае нет необходимости использовать выражение `assign where`, которое проверяет существование настраиваемой переменной `oids`.
>
> Этот метод избавляет от необходимости создавать несколько правил применения. Он также перемещает логику спецификации атрибутов из Службы на Узел.

7.9.1 Переопределение настраиваемых переменных в цикле

Рассмотрим другой, более сложный пример. Пусть осуществляется мониторинг сетевого устройства (Узла) с множеством интерфейсов (Служб). Существуют следующие требования/проблемы:

- Каждая служба интерфейса должна быть названа с префиксом и именем, определенным в объекте Узла (которое может быть сгенерировано из вашей CMDB и т.д.).
- Каждый интерфейс имеет свой собственный тег VLAN.
- На некоторых интерфейсах включена функция контроля качества QoS.
- Дополнительные атрибуты, такие как `display_name` или `notes`, `notes_url` и `action_url` должны генерироваться динамически.

Рекомендуется определить пароль для SNMP (комьюнити) как глобальную константу в файле constants.conf:

```
const IftrafficSnmpCommunity = "public"
```

1) Объявите настраиваемую переменную `interfaces` в объекте Узла `cisco-catalyst-6509-34` и добавьте три примера интерфейсов в качестве ключей словаря.

Укажите дополнительные атрибуты во вложенном словаре:

```
object Host "cisco-catalyst-6509-34" {
  import "generic-host"
  display_name = "Catalyst 6509 #34 VIE21"
  address = "127.0.1.4"

  /* "GigabitEthernet0/2" имя интерфейса
   * и имя ключа для службы в примерах ниже
   */
  vars.interfaces["GigabitEthernet0/2"] = {
     /* установка всех настраиваемых переменных
      * с одинаковыми именами (требуется для параметров/аргументов
      * команд при установке Службы (можно посмотреть
      * в определении CheckCommand)
      */
     iftraffic_units = "g"
     iftraffic_community = IftrafficSnmpCommunity
     iftraffic_bandwidth = 1
     vlan = "internal"
     qos = "disabled"
  }
  vars.interfaces["GigabitEthernet0/4"] = {
     iftraffic_units = "g"
     //iftraffic_community = IftrafficSnmpCommunity
     iftraffic_bandwidth = 1
     vlan = "remote"
     qos = "enabled"
  }
  vars.interfaces["MgmtInterface1"] = {
     iftraffic_community = IftrafficSnmpCommunity
     vlan = "mgmt"
     interface_address = "127.99.0.100" #special management ip
  }
}
```

2) Начните с использования цикла для установки значений и пройдите по словарю `host.vars.interfaces`. Это словарь, который должен использовать переменные

interface_name в качестве ключа и interface_config в качестве значения для каждого сгенерированного объекта.

Конструкция "if-" определяет префикс имени объекта для каждой Службы, что приводит к имени вида if-<имя_интерфейса> в каждой итерации:

```
/* итерирование по словарю host.vars.interfaces,
 * конструкция вида for (key => value in dict), обозначает
 * использование `interface_name в качестве ключа,
 * а `interface_config` - значения. Доступ к атрибутам
 * конфигурации осуществляется с помощью символа индексатора (`.`).
 */
apply Service "if-" for (interface_name => interface_config in
host.vars.interfaces) {
```

3) Импортируйте шаблон generic-service и установите для Команды check_command значение iftraffic. Используйте ключ interface_name для установки значения строки display_name для внешних интерфейсов.

```
  import "generic-service"
  check_command = "iftraffic"
  display_name = "IF-" + interface_name
```

4) Значение ключа interface_name – это та же строка, которая используется в качестве параметра команды для iftraffic:

```
  /* использование ключа в качестве аргумента команды (без дублирования
значений в host.vars.interfaces) */
  vars.iftraffic_interface = interface_name
```

5) Необходимо помнить, что словарь interface_config является вложенным словарем и при первой итерации он будет иметь следующий вид::

```
interface_config = {
  iftraffic_units = "g"
  iftraffic_community = IftrafficSnmpCommunity
  iftraffic_bandwidth = 1
  vlan = "internal"
  qos = "disabled"
}
```

6) Получите доступ к ключам словаря с помощью синтаксиса индексатора и присвойте их настраиваемым переменным, используемым в качестве параметров команды iftraffic.

```
  /* использование настраиваемых переменных в качестве аргументов команды */
  vars.iftraffic_units = interface_config.iftraffic_units
  vars.iftraffic_community = interface_config.iftraffic_community
```

7) Если необходимо просто наследовать все атрибуты, указанные в словаре interface_config, то добавьте их в настраиваемые переменные Службы следующим образом:

```
  /* вышеуказанное может быть достигнуто более коротким способом, если имена
```

```
   * внутри host.vars.interfaces будут точно соответствовать требованиям к
   * параметрам команды в определении команды проверки.
   */
  vars += interface_config
```

8) Если не были указаны значения по умолчанию для требуемых настраиваемых переменных Службы, то их можно задать прямо здесь. Установка значений данным образом также дополнительно позволит избежать нежелательных ошибок проверки конфигурации или сбоев во время выполнения. Подробнее об условных операторах написано в отдельном документе .....

```
  /* установить значение по умолчанию для единиц измерения и пропускной
   * способности */
  if (interface_config.iftraffic_units == "") {
    vars.iftraffic_units = "m"
  }
  if (interface_config.iftraffic_bandwidth == "") {
    vars.iftraffic_bandwidth = 1
  }
  if (interface_config.vlan == "") {
    vars.vlan = "not set"
  }
  if (interface_config.qos == "") {
    vars.qos = "not set"
  }
```

9) Если в Узле отсутствует пароль SNMP (комьюнити), установите значение по умолчанию, заданное глобальной константой `IftrafficSnmpCommunity`:

```
  /* установить глобальную константу, если она
   * явно не указана в словаре `interfaces` на Узле
   */
  if (len(interface_config.iftraffic_community) == 0 ||
len(vars.iftraffic_community) == 0) {
    vars.iftraffic_community = IftrafficSnmpCommunity
  }
```

10) Используя предоставленные значения, сформируйте дополнительные атрибуты объекта, которые будут видны, например, во внешних интерфейсах:

```
  /* Формирование дополнительных атрибутов объекта после заполнения словаря
`vars` */
  notes = "Interface check for " + interface_name + " (units: '" +
interface_config.iftraffic_units + "') in VLAN '" + vars.vlan + "' with '
QoS '" + vars.qos + "'"
  notes_url = "https://foreman.company.com/hosts/" + host.name
  action_url = "https://snmp.checker.company.com/" + host.name + "/if-" +
interface_name
}
```

11) Рекомендация.

Построение конфигурации таким динамическим способом требует детальной информации о созданных объектах. Для получения нужной информации после успешной проверки конфигурации можно использовать команду CLI `object list`.

Убедитесь, что цикл for успешно создал объекты Служб с унаследованными настраиваемыми переменными:

```
# integralCore daemon -C
# integralCore object list --type Service --name *catalyst*

Object 'cisco-catalyst-6509-34!if-GigabitEthernet0/2' of type 'Service':
......
  * vars
    % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 59:3-59:26
    * iftraffic_bandwidth = 1
    * iftraffic_community = "public"
      % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 53:3-53:65
    * iftraffic_interface = "GigabitEthernet0/2"
      % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 49:3-49:43
    * iftraffic_units = "g"
      % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 52:3-52:57
    * qos = "disabled"
    * vlan = "internal"


Object 'cisco-catalyst-6509-34!if-GigabitEthernet0/4' of type 'Service':
...
  * vars
    % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 59:3-59:26
    * iftraffic_bandwidth = 1
    * iftraffic_community = "public"
      % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 53:3-53:65
      % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 79:5-79:53
    * iftraffic_interface = "GigabitEthernet0/4"
      % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 49:3-49:43
    * iftraffic_units = "g"
      % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 52:3-52:57
    * qos = "enabled"
    * vlan = "remote"

Object 'cisco-catalyst-6509-34!if-MgmtInterface1' of type 'Service':
...
  * vars
    % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 59:3-59:26
    * iftraffic_bandwidth = 1
      % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 66:5-66:32
    * iftraffic_community = "public"
      % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 53:3-53:65
    * iftraffic_interface = "MgmtInterface1"
      % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 49:3-49:43
    * iftraffic_units = "m"
      % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 52:3-52:57
      % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 63:5-63:30
    * interface_address = "127.99.0.100"
    * qos = "not set"
      % = modified in '/etc/integralCore/conf.d/iftraffic.conf', lines 72:5-72:24
    * vlan = "mgmt"
```

7.10 Использование атрибутов объекта в правилах применения

Поскольку правила применения вычисляются после общих объектов, то можно ссылаться на существующие атрибуты объектов Узла и/или Службы как на значения для любого атрибута объекта, указанного в этом правиле применения.

```
object Host "opennebula-host" {
  import "generic-host"
  address = "10.1.1.2"

  vars.hosting["cust1"] = {
    http_uri = "/shop"
    customer_name = "Customer 1"
    customer_id = "7568"
    support_contract = "gold"
  }
  vars.hosting["cust2"] = {
    http_uri = "/"
    customer_name = "Customer 2"
    customer_id = "7569"
    support_contract = "silver"
  }
}
```

`hosting` – это настраиваемая переменная с типом значения «словарь» (Dictionary). Это обязательно для проведения итерации с обозначением `key => value` (ключ => значение) в приведенном ниже примере правила применения:

```
apply Service for (customer => config in host.vars.hosting) {
  import "generic-service"
  check_command = "ping4"

  vars.qos = "disabled"

  vars += config

  vars.http_uri = "/" + customer + "/" + config.http_uri

  display_name = "Shop Check for " + vars.customer_name + "-" +
vars.customer_id

  notes = "Support contract: " + vars.support_contract + " for Customer " +
vars.customer_name + " (" + vars.customer_id + ")."

  notes_url = "https://foreman.company.com/hosts/" + host.name
  action_url = "https://snmp.checker.company.com/" + host.name + "/" +
vars.customer_id
}
```

Для каждой итерации цикла указываются разные значения атрибутов `customer` и `config` в теле цикла:

```
1.
customer = "cust 1"
```

```
config = {
  http_uri = "/shop"
  customer_name = "Customer 1"
  customer_id = "7568"
  support_contract = "gold"
}
2.
customer = "cust2"
config = {
  http_uri = "/"
  customer_name = "Customer 2"
  customer_id = "7569"
  support_contract = "silver"
}
```

После этого можно добавить словарь `config` в `vars`:

```
vars += config
```

Теперь в первой итерации тело цикла будет выглядеть следующим образом:

```
customer = "cust 1"
vars = {
  http_uri = "/shop"
  customer_name = "Customer 1"
  customer_id = "7568"
  support_contract = "gold"
}
```

Напомним, что эта структура уже встречалась ранее. Доступ к изменяемым атрибутам также можно получить с помощью синтаксиса индексатора:

```
  vars.http_uri = ... + config.http_uri
```

либо записать в виде:

```
  vars += config
  vars.http_uri = ... + vars.http_uri
```

8 Группы

8.1 Общие сведения о группах

Группа – это совокупность подобных объектов. Группы в основном используются для визуализации в веб-интерфейсах.

Принадлежность к группе определяется непосредственно на самом объекте. Предположим, есть список Узлов с именем `windows`. Надо объединить определенные Узлы из данного списка в группу для последующего просмотра группы на панели оповещения системы мониторинга.

1) Для этого сначала создается объект с типом Группа Узлов (HostGroup):

```
object HostGroup "windows" {
  display_name = "Windows Servers"
}
```

2) Затем следующим образом в группу добавляются Узлы:

```
template Host "windows-server" {
  groups += [ "windows" ]
}

object Host "mssql-srv1" {
  import "windows-server"

  vars.mssql_port = 1433
}

object Host "mssql-srv2" {
  import "windows-server"

  vars.mssql_port = 1433
}
```

3) Аналогичным образом можно создать группы для объектов другого типа, например, создать группы для Служб, Пользователей и т.д.:

```
object UserGroup "windows-mssql-admins" {
  display_name = "Windows MSSQL Admins"
}

template User "generic-windows-mssql-users" {
  groups += [ "windows-mssql-admins" ]
}

object User "win-mssql-noc" {
  import "generic-windows-mssql-users"

  email = "noc@example.com"
}

object User "win-mssql-ops" {
  import "generic-windows-mssql-users"

  email = "ops@example.com"
}
```

8.2 Назначение объекта в группу

Вместо того чтобы добавлять каждый объект в группу вручную, можно добавить объекты в группу на основе их атрибутов:

```
object HostGroup "prod-mssql" {
```

```
  display_name = "Production MSSQL Servers"

  assign where host.vars.mssql_port && host.vars.prod_mysql_db
  ignore where host.vars.test_server == true
  ignore where match("*internal", host.name)
}
```

В приведенном примере все Узлы с `vars`-атрибутом `mssql_port` будут добавлены в Группу Узлов `mssql`. Однако в эту группу не будут добавлены те Узлы, чье имя соответствует строке `\*internal` или их атрибут `test_server` имеет значение `true`.

Подробное описание синтаксиса выражения `assign where` приведены в отдельном документе Справочнике по языку.

9 Уведомления

9.1 Общие сведения об Уведомлениях

Уведомления о проблемах в работе Служб и Узлов являются неотъемлемой частью пользовательской настройки мониторинга.

По умолчанию, когда Узел или Служба находятся в состоянии обслуживания, определена проблема или логика зависимости определила, что Узел/Служба недоступны, Уведомления не отправляются. Можно настроить дополнительные фильтры для типов объектов и состояний, которые уточнят, какие уведомления и в каких случаях должны быть отправлены.

Спецификация Уведомления требует указать одного или нескольких пользователей (и/или групп пользователей), которые получат данное Уведомление в случае возникновения проблем. У данных пользователей должны быть определены все настраиваемые атрибуты, которые будут использоваться в команде `NotificationCommand` при выполнении.

Рассмотрим следующий пример.

Пусть пользователь `integraladmin` получает Уведомления только о проблемах, приводящих объекты в состояния Внимание (`Warning`) и Критическое (`Critical`). Кроме того, он так же получает Уведомления о восстановлении – для них требуется штатное состояние (`OK`).

```
object User "integraladmin" {
  display_name = "integral Admin"
  enable_notifications = true
  states = [ OK, Warning, Critical ]
  types = [ Problem, Recovery ]
  email = "integral@localhost"
}
```

Если для объекта Пользователь не установить атрибуты конфигурации состояний и типов, то будут отправляться уведомления для всех состояний и типов.

> Примечание. Для работы с уведомлениями в системе мониторинга должна быть включена функция уведомления.

Должно быть определено, какая информация интересует адресатов уведомлений в случае возникновения чрезвычайной ситуации, а какая информация не представляет никакой ценности.

Ниже приведен и описан пример команды уведомления.

Можно добавить все общие атрибуты в шаблон уведомления, от которого наследуются заданные Уведомления. Таким образом они сохранятся для каждого объекта Уведомления созданного на основе шаблона. Сами атрибуты могут быть переопределены локально.

```
template Notification "generic-notification" {
  interval = 15m

  command = "mail-service-notification"

  states = [ Warning, Critical, Unknown ]
  types = [ Problem, Acknowledgement, Recovery, Custom, FlappingStart,
            FlappingEnd, DowntimeStart, DowntimeEnd, DowntimeRemoved ]

  period = "24x7"
}
```

В рассматриваемом выше примере для Системы включен временной режим рассылки 24x7 с.

Далее для формирования Уведомлений для указанных Служб необходимо использовать ключевое слово `apply`:

```
apply Notification "notify-cust-xy-mysql" to Service {
  import "generic-notification"

  users = [ "noc-xy", "mgmt-xy" ]

  assign where match("*has gold support 24x7*", service.notes) &&
(host.vars.customer == "customer-xy" || host.vars.always_notify == true
  ignore where match("*internal", host.name) || (service.vars.priority < 2
&& host.vars.is_clustered == true)
}
```

Вместо того чтобы назначать пользователей в качестве адресатов для Уведомлений, можно к объекту `Notification` добавить атрибут `user_groups` со списком групп пользователей. В таком случае система будет отправлять Уведомления всем членам группы.

> Примечание. Уведомления о восстановлении получают только те пользователи, которые ранее получили уведомление о самой проблеме непосредственно (для Служб это состояния: Внимание (`Warning`), Критическое (`Critical`) и Неизвестно (`Unknown`), для узлов это состояние Недоступен (`Down`)).

Также возможно настроить для Пользователя получение Уведомлений с типами Подтверждение проблемы и/или Восстановление, без уведомления о проблеме. Такие Уведомления будут отправляться без предварительного уведомления о проблеме и могут быть использованы, например, в системах управления заявками.

```
object User "ticketadmin" {
  display_name = "Ticket Admin"
  enable_notifications = true
  states = [ OK, Warning, Critical ]
  types = [ Acknowledgement, Recovery ]
  email = "ticket@localhost"
```

```
}
```

9.2 Уведомления от Узла/Службы Пользователю

Стандартной практикой является включение данных пользователей и групп пользователей в состав объектов Узла или Службы, а не в состав объекта Уведомление.

> Примечание. Рекомендуется для полного понимания приведенного ниже примера обязательно прочитать раздел о правилах применения (см. п.7) и раздел о настраиваемых переменных (см. п.5).

Для понимания процесса отправки Уведомлений от Узлов/Служб пользователям системы рассмотрим следующий пример.

Необходимо указать пользователя и группы как вложенную настраиваемую переменную на объекте Узла:

```
object Host "agent1.localdomain" {
  [...]

  vars.notification["mail"] = {
    groups = [ "integraladmins" ]
    users = [ "integraladmins" ]
  }
  vars.notification["sms"] = {
    users = [ "integraladmins" ]
  }
}
```

Как видно из примера, здесь есть возможность использовать два разных правила применения Уведомлений: одно для электронной почты (`mail`) и одно для СМС (`sms`).

В данном примере вложенные ключи `users` и `groups` из настраиваемой переменной `notification` назначаются фактическим атрибутам объекта Уведомление.

Поскольку ошибки трудно отлаживать, если конфигурация не указана в атрибутах объектов Узлов, можно добавить дополнительное условие, которое регистрирует, какой Узел затронут:

```
critical/config: Host 'client3.localdomain' does not specify required
user/user_groups configuration attributes for notification 'mail-
integraladmins'.
```

Также для получения более подробной информации об Узле можно использовать отладчик скриптов:

```
apply Notification "mail-host-notification" to Host {
  [...]

  /* Зафиксируйте, у какого узла нет обязательных атрибутов
   * user/user_groups. Это сразу приведет к сбою во время проверки
   * конфигурации и очень поможет.
   */
  if (len(host.vars.notification.mail.users) == 0 &&
len(host.vars.notification.mail.user_groups) == 0) {
```

```
    log(LogCritical, "config", "Host '" + host.name + "' does not specify
required user/user_groups configuration attributes for notification '" +
name + "'.")
  }

  users = host.vars.notification.mail.users
  user_groups = host.vars.notification.mail.groups

  assign where host.vars.notification.mail &&
typeof(host.vars.notification.mail) == Dictionary
}

apply Notification "sms-host-notification" to Host {
  [...]

  /* Зафиксируйте, какой узел не указывает обязательные атрибуты
   * user/user_groups. Это приведет к сбою сразу же во время проверки
   * конфигурации и очень поможет.
   */
  if (len(host.vars.notification.sms.users) == 0 &&
len(host.vars.notification.sms.user_groups) == 0) {
    log(LogCritical, "config", "Host '" + host.name + "' does not specify
required user/user_groups configuration attributes for notification '" +
name + "'.")
  }

  users = host.vars.notification.sms.users
  user_groups = host.vars.notification.sms.groups

  assign where host.vars.notification.sms &&
typeof(host.vars.notification.sms) == Dictionary
}
```

В приведенном выше примере функция `typeof` используется в качестве меры предосторожности, которая гарантирует, что ключ `mail` действительно предоставляет словарь в качестве значения. В ином случае проверка конфигурации может завершится неудачей, например если администратор добавит на другом Узле:

```
  vars.notification.mail = "yes"
```

Также можно выполнить более детализированную настройку для объекта типа Служба:

```
apply Service "http" {
  [...]

  vars.notification["mail"] = {
    groups = [ "integraladmins " ]
    users = [ "integraladmins" ]
  }

  [...]
}
```

Рассмотрим другой вариант правила применения для Уведомлений.

Пользователи и группы пользователей, являющиеся адресатами для уведомлений Службы, наследуются от объекта Службы, а если они там не установлены, то наследуются от соответствующего объекта Узла. Также задается пользователь по умолчанию.

```
apply Notification "mail-service-notification" to Service {
  [...]

  if (service.vars.notification.mail.users) {
    users = service.vars.notification.mail.users
  } else if (host.vars.notification.mail.users) {
    users = host.vars.notification.mail.users
  } else {
    /* Пользователь по умолчанию, который получает все уведомления. */
    users = [ "integraladmins" ]
  }

  if (service.vars.notification.mail.groups) {
    user_groups = service.vars.notification.mail.groups
  } else if (host.vars.notification.mail.groups) {
    user_groups = host.vars.notification.mail.groups
  }

  assign where ( host.vars.notification.mail &&
typeof(host.vars.notification.mail) == Dictionary ) || (
service.vars.notification.mail && typeof(service.vars.notification.mail) ==
Dictionary )
}
```

9.3 Эскалация уведомлений

Если уведомление о проблеме отправлено, а проблема все еще существует на момент повторного уведомления, то, при необходимости, можно перевести проблему на следующий уровень поддержки. Другой подход заключается в настройке уведомления по умолчанию по электронной почте и эскалации проблемы с помощью SMS, если данная проблема еще не решена.

Возможно определить время начала и окончания отправки Уведомлений в качестве дополнительных атрибутов конфигурации, для дальнейшего использования в эскалации уведомлений. Используя шаблоны, можно предоставить общий доступ к основным атрибутам Уведомлений, таким как `users` (пользователи) или `interval` (интервал), и затем переопределить их для эскалации.

Используя приведенный выше пример, можно определить дополнительных пользователей, которым будут передаваться SMS-уведомления в период времени между началом и окончанием действия Уведомления.

```
object User "integral-oncall-2nd-level" {
  display_name = "integral 2nd Level"

  vars.mobile = "+7 555 424642"
}

object User "integral-oncall-1st-level" {
```

```
  display_name = "integral 1st Level"

  vars.mobile = "+7 555 424642"
}
```

Так же необходимо определить дополнительную команду уведомлений (NotificationCommand) для SMS-уведомлений. Пример приведен ниже.

> Примечание. Приведенный пример является частным случаем, так как существует множество различных SMS-провайдеров. Следует помнить, что для использования функционала отправки SMS-уведомлений требуется поставщик услуг SMS или местное оборудование с активной SIM-картой.

```
 object NotificationCommand "sms-notification" {
   command = [
     PluginDir + "/send_sms_notification",
     "$mobile$",
     "..."
}
```

Рассмотрим приведенный ниже пример. Две новые эскалации уведомлений добавляются на локальный Узел и его Службу «ping4» с помощью шаблона `generic-notification`. Пользователь `integral-oncall-2nd-level` (пользователь 2-ого уровня) будет получать уведомления по SMS (команда `sms-notification`) через 30 мин и до достижения 1 часа с момента фиксации события.

> Примечание. В примере универсального шаблона `generic-notification` атрибут `interval` (интервал) установлен как 15 минут. Можно уменьшить это значение при эскалациях, используя дополнительный шаблон или переопределив атрибут непосредственно в структуре данных уведомлений для `escalation-sms-2nd-level`.

Если проблема не будет решена или принята, что остановит дальнейшую рассылку уведомления, то пользователь `escalation-sms-1st-level` получит уведомление через 1 час (`1h`) после первоначального уведомления о проблеме, но только в течение одного часа (2 часа (`2h`) установлены как ключ `end` для словаря `times`).

```
apply Notification "mail" to Service {
  import "generic-notification"

  command = "mail-notification"
  users = [ "integraladmin" ]

  assign where service.name == "ping4"
}

apply Notification "escalation-sms-2nd-level" to Service {
  import "generic-notification"

  command = "sms-notification"
  users = [ "integral-oncall-2nd-level" ]

  times = {
```

```
    begin = 30m
    end = 1h
  }

  assign where service.name == "ping4"
}

apply Notification "escalation-sms-1st-level" to Service {
  import "generic-notification"

  command = "sms-notification"
  users = [ "integral-oncall-1st-level" ]

  times = {
    begin = 1h
    end = 2h
  }

  assign where service.name == "ping4"
}
```

9.4 Задержка уведомления

В некоторых ситуациях о проблеме с объектом надо уведомить пользователя не в момент возникновения проблемы (когда объект перейдет в состояние HARD, см. п.3.5), а с некоторой задержкой.

Для задания временной задержки для отправки уведомления можно использовать словарь times и установить значение ключа `begin` как `15m (begin = 15m)`, если, например, требуется отложить отправку уведомления на 15 минут. Если вам не требуется устанавливать интервал окончания уведомлений – не используйте ключ `end`, в таком случае система не будет проверять время окончания отправки уведомлений.

> Примечание. Если установить ключ `end` в `0`, то при возникновении проблемы на объекте отправка уведомлений немедленно прекратится, что равнозначно отключению уведомления.

Для быстрого повторного получения уведомления необходимо для атрибута `interval` указать относительно короткий интервал уведомления:

```
apply Notification "mail" to Service {
  import "generic-notification"

  command = "mail-notification"
  users = [ "integraladmin" ]

  interval = 5m

  times.begin = 15m // окно уведомления о задержке

  assign where service.name == "ping4"
}
```

Стоит обратить внимание, что данный механизм не учитывает время обслуживания и т.д., имеет значение только время изменения состояния HARD. Например, предположим, что период обслуживания объекта с 14:00 до 16:00. В середине данного интервала в 15:00 на объекте произошла проблема. При значении атрибута `times.begin = 2h` (2ч.) это означает, что отправка уведомления произойдет в 17:00, а не 18:00.

9.5 Отключение повторных уведомлений

Если требуется получать уведомления только один раз, то можно отключить повторные уведомления, присвоив атрибуту `interval` значение `0`:

```
apply Notification "notify-once" to Service {
  import "generic-notification"

  command = "mail-notification"
  users = [ "integraladmin" ]

  interval = 0 // отключить повторное уведомление

  assign where service.name == "ping4"
}
```

9.6 Фильтрация уведомлений по состоянию и типу

Если в объекте Уведомление или Пользователь не определена фильтрацию по состоянию и типу события, система предполагает, что уведомления направляются для всех состояний и типов событий.

Ниже приведены доступные фильтры состояния (`states`) и типа (`types`) для уведомлений:

```
template Notification "generic-notification" {

  states = [ OK, Warning, Critical, Unknown ]
  types = [ Problem, Acknowledgement, Recovery, Custom, FlappingStart,
            FlappingEnd, DowntimeStart, DowntimeEnd, DowntimeRemoved ]
}
```

10 Команды

10.1 Общие сведения о Командах

Система использует три типа объектов Команда:

- Команды, обеспечивающие выполнение проверок (Команды проверки, Check Command);
- Команды, обеспечивающие отправку Уведомлений (Команды Уведомлений, Notification Commands);
- Команды, обеспечивающие обработку событий на Узлах и Службах (Event command).

10.2 Команды проверки

Объекты типа Команда проверки определяют порядок вызова проверки в интерфейсе командной строки.

Объекты типа Команда Проверки вызываются объектами типа Узел и Служба. Вызываемая команда проверки устанавливается в атрибуте `check_command`.

> Примечание. Для работы с Командами проверки необходимо чтобы в Системе была включена функция выполнения проверок – checker.

10.2.1 Использование плагинов в командах проверки

Сначала необходимо загрузить в Систему пользовательский плагин проверки (если он еще не загружен) и поместить его в каталог **PluginDir**. Далее в примере используется плагин `check_mysql` из пакета плагинов мониторинга (Monitoring Plugins).

Путь к плагину и все аргументы команды рекомендуется передавать в виде списка строковых аргументов, заключенных в двойные кавычки для экранирования.

Для просмотра всех доступных опций команды – используйте параметр –help. Для примера вызовем плагин `check_mysql` с параметром `--help`, чтобы увидеть все доступные опции.

```
integral@integral $ /usr/lib64/nagios/plugins/check_mysql --help
...
This program tests connections to a MySQL server

Usage:
check_mysql [-d database] [-H host] [-P port] [-s socket]
[-u user] [-p password] [-S] [-l] [-a cert] [-k key]
[-C ca-cert] [-D ca-dir] [-L ciphers] [-f optfile] [-g group]
```

Следующий шаг – понять, как параметры команды передаются от Узла или Службы, и установить определение для Команды проверки на основе требуемых для неё параметров и/или значений по умолчанию.

10.2.2 Передача Команде проверки параметров от Узла или Службы

Параметры Команды проверки определяются как настраиваемые переменные, к которым выполняемая Команда проверки может обращаться как к runtime-макросам (см. п. 6.1).

Для того чтобы попрактиковаться в передаче параметров Команды, пользователю следует интегрировать свой собственный плагин.

Рассмотрим пример с плагином `check_mysql`. Необходимо установить настраиваемые переменные Команды проверки по умолчанию, например, `mysql_user` и `mysql_password` (свободно определяемая схема именования), и указать их опциональные пороговые значения по умолчанию. Затем вы можете использовать данные настраиваемые переменные как runtime-макросы для аргументов команд в командной строке.

> Примечание. Рекомендуется использовать общий тип команды в качестве префикса для аргументов команды, чтобы повысить удобочитаемость. Аргумент `mysql_user` помогает лучше понять контекст, чем просто `user` в качестве аргумента.

Настраиваемые переменные по умолчанию могут быть переопределены настраиваемыми переменными, установленными на объектах Узла или Службы, с помощью Команды проверки

my-mysql. Настраиваемые переменные также могут быть унаследованы от родительского шаблона с использованием аддитивного наследования (+=).

```
# vim /etc/integral/conf.d/commands.conf

object CheckCommand "my-mysql" {
  command = [ PluginDir + "/check_mysql" ] //constants.conf -> const
PluginDir

  arguments = {
    "-H" = "$mysql_host$"
    "-u" = {
      required = true
      value = "$mysql_user$"
    }
    "-p" = "$mysql_password$"
    "-P" = "$mysql_port$"
    "-s" = "$mysql_socket$"
    "-a" = "$mysql_cert$"
    "-d" = "$mysql_database$"
    "-k" = "$mysql_key$"
    "-C" = "$mysql_ca_cert$"
    "-D" = "$mysql_ca_dir$"
    "-L" = "$mysql_ciphers$"
    "-f" = "$mysql_optfile$"
    "-g" = "$mysql_group$"
    "-S" = {
      set_if = "$mysql_check_slave$"
      description = "Проверка работы слейвов."
    }
    "-l" = {
      set_if = "$mysql_ssl$"
      description = "Использовать шифрование SSL"
    }
  }

  vars.mysql_check_slave = false
  vars.mysql_ssl = false
  vars.mysql_host = "$address$"
}
```

В приведенном выше примере определения Команды проверки для переменной mysql_host в качестве значения по умолчанию устанавливается $address$. Можно переопределить данный параметр Команды, если, например, узел MySQL запущен на ip-адресе отличном, от адреса, на котором запущена система мониторинга.

Обязательно требуется передать все необходимые параметры Команды, такие как mysql_user, mysql_password и mysql_database. В приведенном ниже примере параметры MysqlUsername и MysqlPassword указаны как глобальные константы.

```
# vim /etc/integral/conf.d/services.conf

apply Service "mysql-integral-db-health" {
  import "generic-service"
```

```
  check_command = "my-mysql"

  vars.mysql_user = MysqlUsername
  vars.mysql_password = MysqlPassword

  vars.mysql_database = "integral"
  vars.mysql_host = "192.168.33.11"

  assign where match("integral*", host.name)
  ignore where host.vars.no_health_check == true
}
```

Рассмотрим другой пример: В примере конфигурации Узла устанавливается проверка Службы `ssh`. Порт ssh Узла по умолчанию не равен `22`, а установлен на `2022`. Можно передать параметр Команды как настраиваемую переменную `ssh_port` непосредственно в правиле применения Службы внутри файла конфигурации:

```
apply Service "ssh" {
  import "generic-service"

  check_command = "ssh"
  vars.ssh_port = 2022 //настраиваемый параметр команды

  assign where (host.address || host.address6) && host.vars.os == "Linux"
}
```

Если требуется, чтобы эта настройка выполнялась на объекте Узла, а не Службы, измените конфигурацию объекта в конфигурационном файле Узла. Порядок/очерёдность исполнения runtime-макроса приведены в п.6.2.

```
object Host "integral-agent1.localdomain {
...
  vars.ssh_port = 2022
}
```