МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН

САТБАЕВ УНИВЕРСИТЕТІ

Институт кибернетики и информационных технологий
Кафедра «Программной инженерии»

Алтаев Дэннис Радикович

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

На соискание академической степени магистра

Название диссертации	Создание инфраструктуры и перенос программных решений на базе Asp.Net в кластер Kubernetes для автоматизации развертывания и повышения отказоустойчивости систем
Направление подготовки	6М100200 – Системы информационной безопасности

Научный руководитель
канд. тех. наук
_____ А.А. Мухамедгалиев
«__» _____ 2020 г.

Оппонент
Доктор PhD, ассистент
профессор кафедры
_____А. Богданчиков
«__» _____ 2020 г.

ДОПУЩЕН К ЗАЩИТЕ
Заведующий кафедрой ПИ
доктор PhD
_____ М. Тұрдалыұлы
«__»_____ 2020 г.

Нормоконтроль
лектор
_____Ж.М.Алибиева
«__» _____ 2020 г.

Алматы 2020

Институт информационных и телекоммуникационных технологий

Кафедра Программная инженерия

Специальность: 6М100200 – Системы информационной безопасности

УТВЕРЖДАЮ
Заведующий кафедрой ПИ
доктор PhD
_____ М. Тұрдалыұлы
«___» _____ 2020 г.

ЗАДАНИЕ
на выполнение магистерской диссертации

магистранту Алтаеву Дэннису Радиковичу

Тема диссертации: «Создание инфраструктуры и перенос программных решений на базе Asp.Net в кластер Kubernetes для автоматизации развертывания и повышения отказоустойчивости систем»

Срок сдачи законченной диссертации «___» _____

Исходные данные к магистерской диссертации. Даны физические сервера и 2 типа запускаемых приложений: Asp.Net и Angular. Поставленные цели и задачи диссертационной работы направлены на улучшение текущей инфраструктуры путем внедрения новых инструментов и технологий для оркестрации приложений, написанных с использованием технологии Asp.Net.

Перечень подлежащих разработке в магистерской диссертации вопросов или краткое содержание магистерской диссертации:

а) определение требований – анализ целей, задач и назначения разработки;

б) исследование инструментов для оркестрации контейнерных служб и выбор подходящего оркестратора;

в) установка и конфигурирование кластера Kubernetes;

г) контейнеризация необходимых приложений в контейнеры Docker и запуск этих приложений в Kubernetes;

д) изучение лучших практик и паттернов развертывания и управления приложениями в кластере;

е) добавление автомасштабирования приложений для повышения отказоустойчивости систем при увеличении нагрузок на приложения.

Рекомендуемая основная литература:

1 Gigi Sayfan, Mastering Kubernetes. – Packt Publishing, 2017. – 400 с.
2 Марко Лукша, Kubernetes в действии / пер. с англ. А. В. Логунов. – М.: ДМК Пресс, 2019. – 672с.
3 Билджин Ибрам, Роланд Хасс, Паттерны Kubernetes: Шаблоны разработки собственных облачных приложений. – СПб.: Питер, 2020.

ГРАФИК
подготовки магистерской диссертации

Наименование разделов, перечень разрабатываемых вопросов	Сроки представления научному руководителю	Примечание
Раздел 1. Сравнение Kubernetes с другими инструментами оркестрации контейнеров.		
Раздел 2. Установка и конфигурирование кластера Kubernetes.		
Раздел 3. Контейнеризация и запуск приложений в кластере.		
Раздел 4. Масштабирование приложений в Kubernetes.		
Раздел 5. Упаковка приложений в пакет для удобного развертывания с помощью Helm.		

Консультации по проекту с указанием относящихся к ним разделов проекта

Раздел	Консультант, (уч. степень, звание)	Сроки	Подпись
Нормоконтроль	Ж.М.Алибиева, Лектор кафедры программная инженерия		

Дата выдачи задания "____"_____ 2020 г.

Заведующий кафедрой _____ М. Тұрдалыұлы

Научный руководитель _____ А.А. Мухамедгалиев

Задание принял к исполнению магистрант _____ Д.Р. Алтаев

Дата "_____"_____ 2020 г.

АННОТАЦИЯ

В данной работе рассматривается создание инфраструктуры для развертывания контейнеризованных приложений путем создания кластера Kubernetes.

Для выбора лучшего оркестратора было проведено исследование трех из них, в том числе Docker Swarm, Nomad и Kubernetes.

Рассмотрены различные виды ресурсов, доступные для использования разработчиком, использующим Kubernetes.

Написаны файлы Docker для контейнеризации приложений, разработанных с помощью фреймворка Angular, а также приложений на базе Asp.net.

Разработан шаблон chart’а для пакетного менеджера Helm с целью автоматизации развертывания новых версий приложений, разработанных на базе Asp.net.

Рассмотрены различные варианты развертывания приложений, позволяющих производить обновление системы с нулевым временем простоя.

Также исследованы различные паттерны для правильного запуска приложений в облачной среде, что позволит увеличить качество программного обеспечения.

Приведен пример обеспечения отказоустойчивости на платформе Kubernetes с помощью развертывания горизонтального автомасштабирования модулей, который контролирует число реплик приложения в зависимости от нагрузки на процессор и оперативную память.

АҢДАТПА

Бұл жұмыста Kubernetes кластерін құру арқылы контейнерлік қосымшаларды өрістету үшін инфрақұрылым құру қарастырылады.

Үздік оркестр таңдау үшін олардың үшеуі зерттеу жүргізілді, соның ішінде Docker Swarm, Nomad және Kubernetes.

Kubernetes-ді қолданатын әзірлеуші пайдалану үшін қол жетімді ресурстардың әртүрлі түрлері қарастырылған.

Қосымшаны контейнерлеу Docker файлдары Angular фреймворкімен әзірленген, сондай-ақ Asp.net негізіндегі қосымшаларға әзірленген .

Helm пакеттік менеджеріне арналған және Asp.net базасында құрылған жаңа нұсқаларын өрістетуін автоматизациялауға арналған chart'а шаблоны құрылды.

Жүйені нөлдік уақытпен жаңартуды жүргізуге мүмкіндік беретін қосымшаларды орналастырудың әртүрлі нұсқалары қарастырылған.

Сондай-ақ, бұлт ортасында қолданбаларды дұрыс іске қосу үшін әртүрлі паттерлер зерттелді, бұл бағдарламалық жасақтаманың сапасын арттырады.

Kubernetes платформасында процессор мен жедел жадқа жүктемеге байланысты қосымшалар репликаларының санын бақылайтын модульдерді көлденең автомасштауды қолдану арқылы бас тартуға төзімділікті қамтамасыз ету мысалы келтірілген.

SUMMARY

In this paper, we consider creating an infrastructure for deploying containerized applications by creating A Kubernetes cluster.

To select the best Orchestrator, a study was conducted on three of them, including Docker Swarm, Nomad, and Kubernetes.

Various types of resources available for use by a developer using Kubernetes are considered.

Docker files are written for containerization of applications developed using the Angular framework, as well as applications based on Asp.net.

The chart template for the Helm package Manager has been developed to automate the deployment of new versions of applications developed on the basis of Asp.net.

Various options for deploying applications that allow you to update the system with zero downtime are considered.

Various patterns for correctly running applications in the cloud environment are also studied, which will increase the quality of software.

An example of providing fault tolerance on the Kubernetes platform by deploying horizontal module autoscaling, which controls the number of replicas of the application depending on the load on the CPU and RAM.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

За последние несколько лет разработка ПО развивается очень быстрыми темпами. Появились новые методологии разработки, такие как Lean и Agile. И так как процесс разработки должен соответствовать данному течению, были придуманы новые практики разработки ПО, вошедшие в методологию DevOps.

DevOps – методология разработки, при которой налажено взаимодействие разработчиков и специалистов по информационно–технологическому обслуживанию [1].

В DevOps используются различные инструменты для создания инфраструктуры для запуска и работы приложений разного стека технологий. Цикл разработки программного обеспечения при использовании методологии DevOps представлен на рисунке 1.

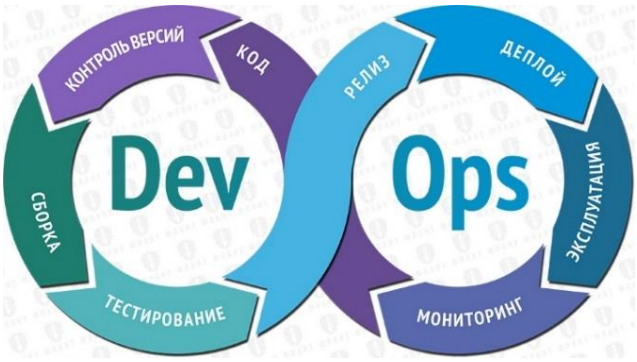

Рисунок 1 – Методология DevOps

Быстрое развитие виртуализации и контейнеризации, а также облачных технологий, и следование методологиям гибкой разработки Agile и DevOps–практик привело к потребности создания нового подхода в разработке программного обеспечения, и эта архитектура получила название микросервисной.

До микросервисов большинство приложений разрабатывалось с использованием монолитной архитектуры, а это означает, что все приложение – один большой тяжеловесный монолит, который занимается всеми функциями бизнеса, и один решает все необходимые задачи. Также в данном подходе к проектированию систем минусом является зависимость одних компонентов от других, и таким образом правки в одной части системы требуют развертывания всего приложения. В вопросах масштабируемости тоже все не так удобно, нельзя масштабировать одну часть системы отдельно от всей системы.

С приходом микросервисов настало время, когда разные команды компании могут разрабатывать свои сервисы на совершенно разных технологиях, развертывать их тогда, когда будет нужно именно этой команде,

и эти развертывания могут происходить по несколько раз в сутки. Отказоустойчивость всей системы от такого подхода тоже повысилась: если перестанет работать одна часть системы, то это всего лишь один микросервис, который можно перезапустить, но остальная часть системы будет продолжать работать как прежде.

Для обеспечение одинакового окружения в момент разработки, и в момент работы в промышленной среде, а также для изоляции приложений друг от друга, приложение нужно собрать в образ, который при запуске становится контейнером. Контейнеры очень легко запускать, в нем уже установлены все необходимые зависимости для работы приложения, и это даёт уверенность в том, что после обновления в production не будет отсутствовать библиотека, которую забыли установить на машину, а она необходима для правильной работы.

Контейнеры – виртуальное разделение ресурсов, поэтому разные контейнеры можно запускать в пределах одной и той же операционной системы, но они не будут знать о существовании друг друга. Это очень удобно, так как экономит ресурсы.

Для управления приложениями, упакованными в контейнеры, существует специальный класс программного обеспечения, именуемый оркестраторами.

Оркестраторы позволяют объединять серверы в кластеры и размещать на них нужные разработчикам рабочие нагрузки. Разработчик или администратор больше не будет заботиться о том, на какой именно сервер публиковать свое приложение.

В настоящее время существует несколько инструментов для оркестрации контейнерных служб: Docker Swarm, Nomad, Kubernetes и др.

В современном мире сложных и распределенных систем все больше требуется инструмент, который позволил бы управлять системой проще и гибче. И в данной ситуации именно Kubernetes подходит как нельзя кстати. С его помощью можно настроить очень удобное масштабирование системы при пиковых нагрузках и всегда вести мониторинг целостной системы. Данная платформа на данный момент является стандартом де–факто и выбором всех крупных мировых компаний–лидеров в информационном мире, поэтому и для построения инфраструктурной платформы для компании, он был выбран изначально.

Предпосылкой для перехода на новую инфраструктуру стало увеличение нагрузки на проекты компании, а также увеличение количества заказчиков и соответственно самих проектов.

Актуальность работы и темы исследования подтверждается тем, что во всем мире сейчас все больше компаний обращаются к данной технологии, так как она изначально была разработана именно для высоконагруженных систем и систем, требующих надежной, отказоустойчивой работы.

Далее рассмотрим подробнее, что из себя представляет каждый оркестратор, и выберем лучший из них.

1 Постановка задач

В текущей инфраструктуре компании все приложения работают на физических серверах. Это дает полный контроль над выполняемыми приложениями, но не обеспечивает должного уровня отказоустойчивости с помощью горизонтального масштабирования при повышениях нагрузки на приложения. В таком варианте инфраструктуры можно настроить балансировщик нагрузки, распределяющий поступающий трафик на несколько серверов, но нельзя на ходу изменить число этих приложений на серверах. Для изменения количества приложений нужно непосредственное вмешательство администратора.

Чтобы обеспечить больший уровень отказоустойчивости и масштабирования было принято решение о создании инфраструктурной платформы, основанной на оркестрации контейнеров. Также такая платформа, помимо запуска обычных приложений, должна иметь возможность запуска различных консольных приложений по расписанию, например, для проведения синхронизации данных при интеграции с другими компаниями.

2 Решение

Решением поставленной задачи является развертывание инфраструктуры с помощью кластера на базе платформы Kubernetes, созданной компанией Google в 2015 году. Данная платформа удовлетворяет всем необходимым требованиям в обеспечении бесперебойной работы различных систем компании, что позволит повысить качество работы компании.

Также в будущем платформа позволит внедрить практики CI/CD, с помощью Azure DevOps Server, так как сейчас в компании используется Team Foundation Server, являющийся предшественником.

3 Сравнение Kubernetes с другими инструментами оркестрации контейнеров

Одними из главных оркестраторов на сегодняшний день являются Kubernetes (или K8S) от компании Google, Docker Swarm от одноименной компании Docker, а также Nomad от компании HashiCorp. Логотипы данных инструментов показаны на рисунке 2.

Рисунок 2 – Логотипы основные системы оркестрации контейнеров

Время выхода каждого из оркестраторов:
– Kubernetes – 21 июля 2014 году;
– Nomad – 1 июня 2015 года;
– Docker Swarm – 28 июля 2016 года;

Как видно Kubernetes тут был первым, и возможно он задал определенный стандарт для дальнейших подобных разработок.

Рассмотрим коротко каждый инструмент.

3.1 Docker Swarm

Docker Swarm mode – абстракция Докера, которая привносит возможность объединения нескольких серверов для работы в режиме кластера. Docker Swarm объединяет множество Docker хостов в один виртуальный хост. И все клиенты, которые работают с Docker API могут продолжать работать с этим API, даже не подозревая, что обращение происходит не к одному хосту, а уже к кластеру.

Этот режим доступен, начиная с версии 1.12. Общая архитектура работы Docker в режиме Swarm представлена на рисунке 3, а схема архитектуры с несколькими Manager серверами в кворуме представлена на рисунке 4.

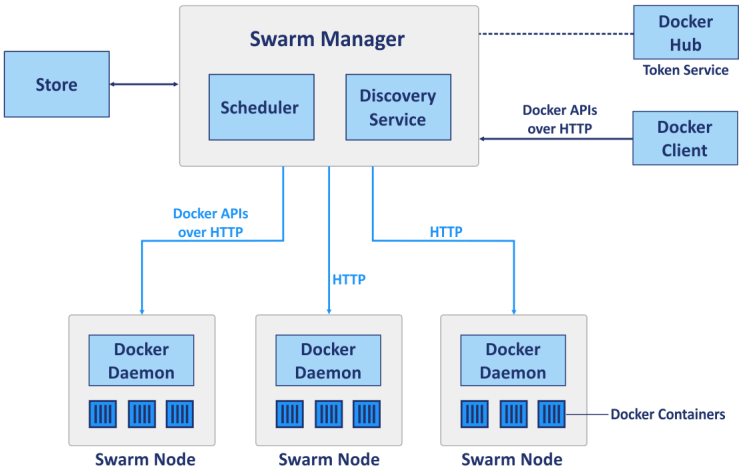

Рисунок 3 – Архитектура кластера на базе Docker Swarm [2]

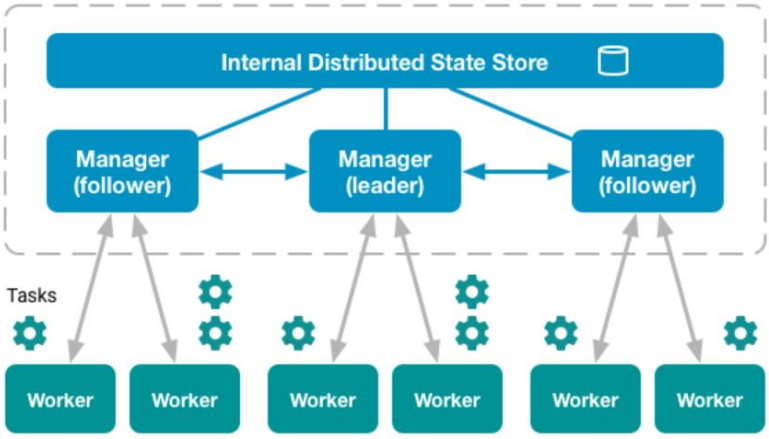

Рисунок 4 – Архитектура кластера на базе Docker Swarm [3]

Основные функции, предлагаемые оркестратором Docker Swarm:

– Управление кластерами, интегрированное с Docker Engine;

– Масштабирование (можно указать количество реплик сервиса);

– Безопасность по умолчанию – создаются самоподписанные сертификаты, можно создавать свои (например, каждый час будут генерироваться и обновляться сертификаты);

– Отказоустойчивость благодаря нескольким manager, находящимся в кворуме;

– Балансировка нагрузки – настраивается Ingress;

– Service discovery – обнаружение сервисов происходит внутри оркестратора, без прямого участия разработчика;

– Можно менять топологию кластера прямо в runtime;

– Плавное обновление.

В кластере существует Manager node, worker node. Manager node находится в сети поддержания согласованности по алгоритму Raft. Raft – алгоритм, использующийся для решения задач нахождения консенсуса в сети

ненадёжных вычислений. Для обеспечения отказоустойчивости в Docker Swarm обычно создают несколько серверов с ролью менеджера, и из них будет избираться лидер по алгоритму Raft, и эта нода теперь имеет роль Leader node.

Рабочие нагрузки запускаются на worker node. Соответственно, если у нас для разработки есть только одна нода, то она будет иметь сразу 3 роли: и manager, и leader, и worker.

Также Swarm работает как балансировщик нагрузки между несколькими рабочими узлами, равномерно распределяя запросы, приходящие с любой из сторон кластера. Например, если в кластере существует 3 сервера, на которых исполняется рабочая нагрузка, а определенное приложение есть только на 2, то если запрос придет на 3 сервер, Ingress с Load Balancer'ом перенаправит трафик на ноду с нужным приложением. Пример балансировки нагрузки представлен на рисунке 5.

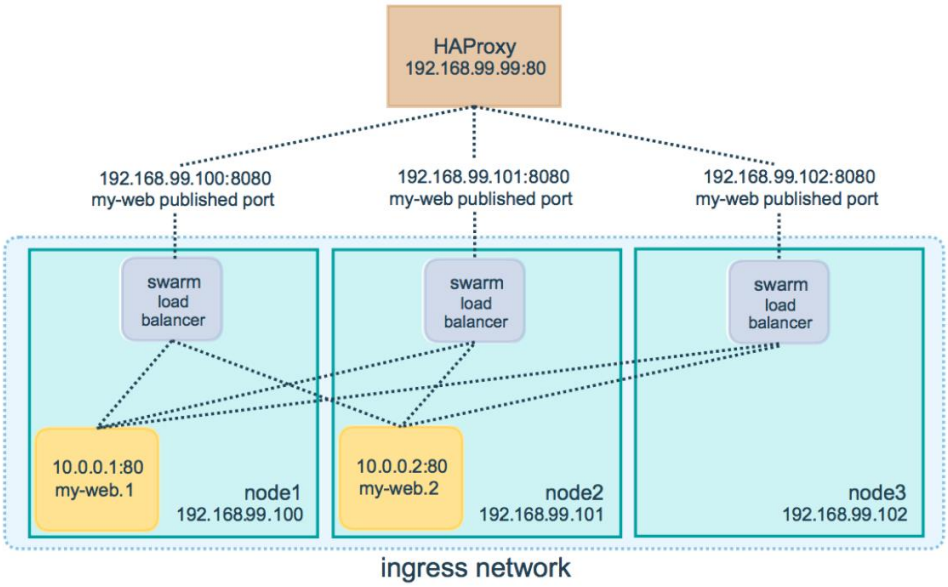

Рисунок 5 – Пример работы балансировщика нагрузки на основе HAProxy [3]

Чтобы присоединить новый сервер в роли менеджера, нужно выполнить команду, показанную на рисунке 6.

```
docker swarm join-token manager
```

Рисунок 6 – Команда для присоединения сервера в роли менеджера

И движок докера сгенерирует команду, показанную на рисунке 7, для добавления новой ноды в кластер.

```
docker swarm join --token <join_token> <manager_ip>:<manager_port>
```

Рисунок 7 – Сгенерированная команда для присоединения сервера в роли рабочего

А чтобы сгенерировать команду для присоединения нового сервера в роли рабочего, нужно выполнить команду, показанной на рисунке 8.

```
docker swarm join-token worker
```

Рисунок 8 – Команда для присоединения сервера в роли рабочего

В данном оркестраторе существует 2 вида сервисов: глобальный и реплицируемый.

Проведя небольшую аналогию с Kubernetes, можно увидеть, что глобальный сервис в Swarm – Daemon set в K8S. Реплицируемый сервис – обычный под или ReplicaSet, либо Replication Controller в K8S.

Плюсы использования кластера на базе Docker Swarm:

– простота использования – если разработчик уже знаком с Docker и Docker Compose, то с помощью всего одной команды можно запустить свои сервисы в режиме кластера;

– нет необходимости в дополнительном ПО, так как Swarm режим уже включен по умолчанию в Docker Engine;

– имеет более легкую кривую обучения, что позволяет за несколько дней, или недель воссоздать инфраструктуру кластера со своей рабочей нагрузкой.

Минусы использования Docker Swarm:

– Docker Swarm дает ограниченные возможности разработки, например, в нем нет автоматического горизонтального масштабирования сервисов;

– нужно самому создавать сети для работы приложений в разных сетях;

– нет поддержки blue–green deployments, из коробки доступна только поддержка плавного развертывания.

3.2 Nomad

Если посмотреть на архитектуру Nomad – станет понятно, что она намного проще, чем в Kubernetes. Nomad – это всего один двоичный файл, как для клиента, так и для сервера, и для него не требуется никаких внешних зависимостей. Архитектура Nomad изображена на рисунке 9.

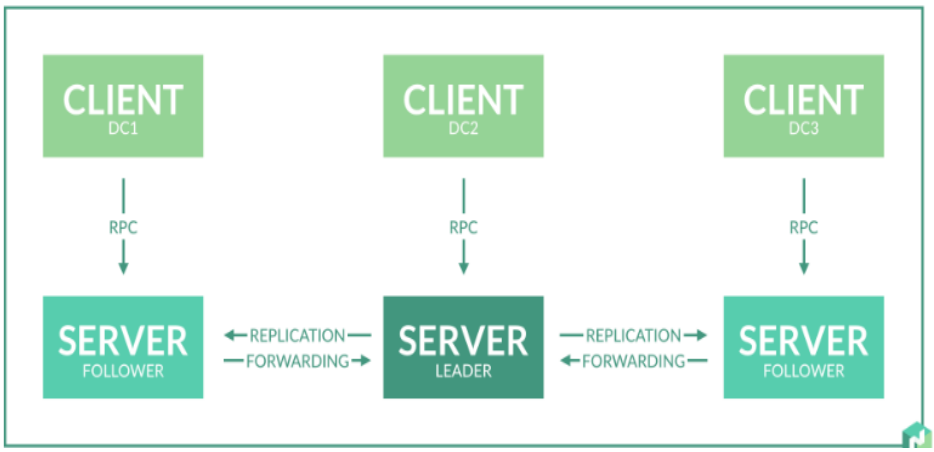

Рисунок 9 – Архитектура проекта Nomad

Nomad сочетает в себе легкий менеджер ресурсов и сложный планировщик в единую систему. По умолчанию Nomad является распределенным, высокодоступным, и функционально простым [4].

Посмотрим, какие ресурсы используются в Nomad:

– Задание – это спецификация в формате HCL (HashiCorp Configuration Language, построен на базе yaml формата), описываемая разработчиком, в которой есть вся информация для запуска рабочей нагрузки в кластере. Задание – это описание того состояния кластера, которое хочет получить разработчик после развертывания сервисов. Аналогом задания является Под в K8S, но он может включать в себя несколько контейнеров, поэтому и группа задач – тоже подходит к этому определению. Одно задание может включать в себя сотни и тысячи групп задач;

– Драйвер – представление, какое средство выполнения задач нужно заданию. Если разворачивается контейнер – используется Docker, также есть драйвер Java или статические двоичные файлы;

– Задача – это минимальная единица работы в Nomad. Задачи выполняются драйверами, которые позволяют Nomad быть гибким в типах задач, которые он поддерживает. Именно в задачах указываются требуемые ресурсы и их ограничения;

– Группа задач – это набор задач, которые требуется запускать и исполнять вместе;

– Клиент Nomad – это сервер, на котором запускаются задачи. На всех клиентах запускаются агенты Nomad. Агент несет ответственность за регистрацию на серверах, следит за любыми работа по назначению и выполнению заданий. Агент Nomad – это процесс, который взаимодействует с серверами для получения и исполнения заданий;

– Распределение – это описание связей между группой задач в задании и сервером–клиентом, на котором это задание будет развернуто;

– Оценка – это механизм, с помощью которого Nomad решает о планировании. В случае различия текущего состояния, и того, которое должно

получиться, Nomad создает новую оценку, и ее результатом может стать изменение кластера;

– Сервер Nomad – то, из чего состоит весь кластер. Серверы выполняют репликацию данных, а также выбор лидера среди себе подобных при помощи известного алгоритма Raft. У серверов можно указывать регион, где он находится, чтобы Nomad имел данные для лучшего разворачивания того или иного сервиса.

Чтобы обеспечить Service discovery (обнаружение сервисов) нужно использовать отдельный инструмент под названием Consul, который разработала та же компания HashiCorp.

В документации проекта есть следующее описание:

«Consul – это сетевой инструмент, который обеспечивает полнофункциональную плоскость управления сеткой обслуживания, обнаружение служб, конфигурацию и сегментацию» [5].

То есть для того, чтобы запустить полноценный кластер Nomad, нужно будет все–таки разобраться в работе еще одного инструмента, который также поставляется единственным бинарным файлом.

Но Consul – это не только Service discovery, это еще и Service Mesh и хранилище key–value. Таким образом, Consul – аналог etcd в Kubernetes. Он позволяет настраивать взаимодействие между различными сервисами, обеспечивать авторизацию и аутентификацию.

Также чтобы хранить секреты в кластере (логины и пароли к базам данных и подобные данные), можно использовать инструмент опять же этой компании под названием Vault. Оба этих инструмента отлично ложатся на инфраструктуру проекта Nomad.

Плюсы использования Nomad относительно Kubernetes:

– требует меньше ресурсов для работы control plane. У Kubernetes это обязательно 3 мастера 2 ядра /4 gb памяти, что подходит для большинства систем;

– не нужна плоская сетевая модель;

– возможность запуска различных артефактов, в том числе бинарных файлов, контейнеров;

– не нужно отдельно устанавливать minikube для создания окружения для разработки, требуется всего лишь одна команда: nomad agent –dev.

Минусы Nomad относительно Kubernetes:

– экосистема еще не так полноценна, поэтому нет определенного стиля разработки инструментов, и разработчику приходится учить совершенно разные инструменты для включения их в свою инфраструктуру;

– пока что мало готовых решений, как и что запускать;

– за безопасность ответственность несет администратор кластера;

– пока нет инструментов для запуска stateful приложений.

Nomad подходит для небольших команд, где архитектура взаимодействия приложения не столь обширна. Но со временем, если система

разрастается, появляются новые заказчики, то скорее всего компания захочет переехать на Kubernetes.

Явным преимуществом над Kubernetes в том, что у него можно запускать различные артефакты, будь то контейнеры, Java файлы или бинарные исполняемые файлы. У Kubernetes есть поддержка различных движков контейнеров, таких как Docker, ContainerD, rkt (проект уже закрылся, но как поддержка старых проектов работает) и все приложения нужно будет упаковывать. А это бывает очень сложно сделать с legacy проектами, в которых не предполагалась возможность контейнеризации. Если сравнивать Nomad с Kubernetes, сообщество Nomad не настолько велико. У Kubernetes уже больше 91 000 коммитов и 2500 контрибьюторов, у Nomad же чуть больше 18 000 коммитов и 368 контрибьюторов. Nomad скорее всего всегда будет отставать по скорости от развития экосистемы вокруг Kubernetes! Но это немного другой инструмент, это более узконаправленная система.

Данные из репозитория Kubernetes представлены на рисунке 10.

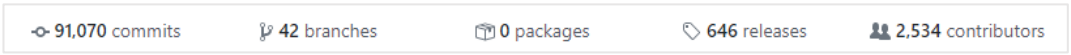

Рисунок 10 – Данные из репозитория Kubernetes

Данные из репозитория Nomad представлены на рисунке 11.

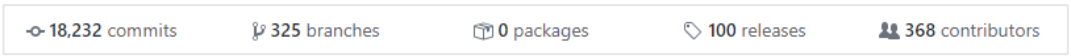

Рисунок 11 – Данные из репозитория Nomad компании HashiCorp

3.3 Общий вывод по трем оркестраторам

Docker swarm – лучший вариант, когда у вас в компании используется Docker Compose и приложения без сложных связей между собой. Он требует меньше всего прилагаемых усилий для изучения, особенно удобен, так как на сегодняшний день платформа Docker планомерно внедряется во все большее и большее количество организаций.

Nomad – очень хороший вариант, в котором уже можно запускать production кластера с множеством сервисов. При тестировании этого оркестратора контейнерных служб разработчики развертывают кластер с 10 000 виртуальных серверов, что доказывают готовность к промышленным нагрузкам. Здесь отсутствует общий стиль разработки инструментов, что делает изучение новых более сложным.

Kubernetes – это фреймворк, который имеет наибольшее сообщество разработчиков, и темпы разработки не сбавляют обороты. Это один

оркестратор, который «из коробки» уже имеет горизонтальное автомасштабирование подов, и в недавних версиях уже появилось вертикальное автомасштабирование. И сама платформа спроектирована для развития, с очень удобным механизмом для расширения возможностей (custom resource definition и операторы), что позволяет сохранять его более модульным. Также большим преимуществом является то, что K8S или Kubernetes стал первым проектом – выпускником в организации CNCF (Cloud Native Computing Foundation). Также в этой организации есть много проектов, которые используются во всей экосистеме Kubernetes (Prometheus, Helm и т.д.). Данный оркестратор уже обзавелся собственной экосистемой из операторов для различных инструментов, а также Helm–chart'ов, поэтому для изучения нового инструмента уже не потребуется так много времени.

Также стоит отметить работу Kubernetes в сфере безопасности. Изначально у платформы политика «разрешено все, что не запрещено», поэтому каждый администратор кластера должен заботиться об этом сам. Безопасность осуществляется с помощью RBAC (Role Based Access Control).

Управление доступом на основе ролей (RBAC) – это метод регулирования доступа к компьютерам и сетевым ресурсам, опирающийся на роли отдельных пользователей в компании. RBAC можно использовать со всеми ресурсами Kubernetes, которые поддерживают CRUD (Create, Read, Update, Delete) [6].

В Kubernetes есть предел, описанный разработчиками в документации, который имеет следующие границы кластера: не более 5000 узлов, не более 150 000 подов, не более 300 000 полных контейнеров, не более 100 подов на узел [7]. Сравнительная таблица возможностей разных оркестраторов представлена в таблице 1.

Возможность	Docker Swarm	Nomad	Kubernetes
Обнаружение сервисов	+	При помощи Consul	+
Горизонтальное автомасштабирование	-	-	+
Вертикальное автомасштабирование	-	-	+
Безопасность	Генерация сертификатов по умолчанию	Ответственность администратора	Ответственность администратора
Простота в изучении	Просто	Не очень сложно	Сложно
Планирование на определенных узлах	-	Указание дата центра	NodeSelector, NodeAffinity, PodAffinity, PodAntiAffinity

Таблица 1 – Сравнительная таблица возможностей 3 оркестраторов

4 Базовая архитектура Kubernetes

Kubernetes – платформа с открытым исходным кодом для автоматизации развертывания, масштабирования и управления контейнерными приложениями. Данная платформа позволяет оркестрировать приложениями, настраивая взаимодействие между разными частями приложения (в случае с микросервисами), а также среду выполнения. Kubernetes имеет свой DNS, который используется всем кластером, и это позволяет обращаться к сервисам напрямую, не зная IP адресов, и не заботясь о том, что они могут меняться.

Как показано на рисунке 12, на аппаратном уровне кластер состоит из множества узлов, которые можно поделить на 2 основных типа:

– Master узел;

– Рабочий (worker) узел.

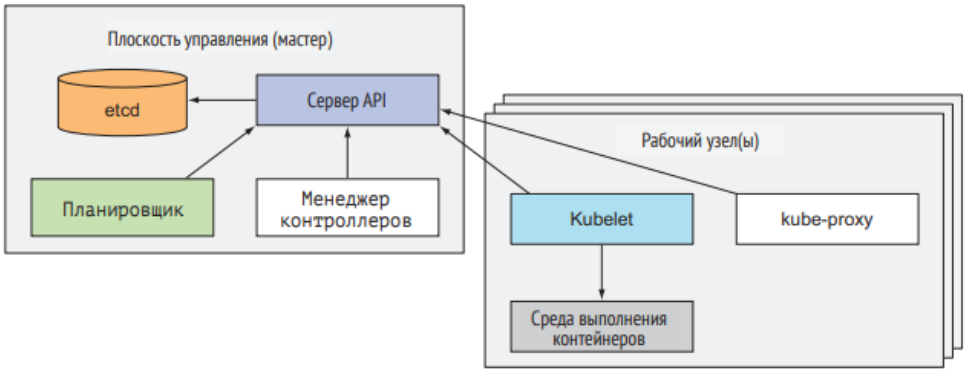

Рисунок 12 – Аппаратный уровень кластера

4.1 Master узел

На Master узле размещена плоскость управления (control plane), которая управляет всем кластером Kubernetes. К нему можно обращаться через командную строку, Dashboard, или Rest API.

Плоскость управления состоит из нескольких компонентов, которые могут быть реплицированы для обеспечения высокодоступности, и отказоустойчивости кластера:

– API сервер – сердце всей системы, ведь с ним взаимодействует разработчик и остальные компоненты, находящиеся в плоскости управления. это компонент, который отвечает за политики, предоставляя ограниченный доступ к etcd;

– Планировщик (scheduler) – планирует работу по распределению модулей по разным рабочим узлам. Это осуществляется благодаря тому, что

планировщик содержит сведения об использовании ресурсов на каждом рабочем узле и требуемые ресурсы для работы каждого конкретного модуля. Также пользователем могут быть заданы метки для определенных модулей, чтобы они размещались на нужных узлах, например, с ssd диском вместо hdd, и планировщик берет все эти данные в расчет [8];

– Менеджер контроллеров (controller manager) – выполняет функции на уровне кластера, такие как репликация модулей, отслеживание всех рабочих узлов, а также обработка аварийных сбоев узлов. В Controller Manager есть код, реализующий поведение ReplicaSet (поддерживает заданное количество одновременно выполняющихся реплик пода). Этот контроллер следит за ресурсом ReplicaSet и набором подов, основанных на селекторе в этом ресурсе. Он создает/удаляет поды, поддерживая таким образом стабильный набор подов, согласно описанию, в ReplicaSet. Большинство контроллеров действуют по такому же принципу. Как писал в своей книге «DevOps with Kubernetes» Hideto Saito, диспетчер контроллеров управляет множеством различных вещей в кластере. Диспетчер контроллеров репликации гарантирует, что все контроллеры репликации работают на требуемом объеме реплик. Диспетчер контроллера узлов отвечает, когда узлы выходят из строя, он затем выселяет поды. Контроллер конечных точек используется для связывания отношений между службами и подами. Service Account и контроллер токенов используются для управления учетной записью по умолчанию и токенами доступа API [9];

– Хранилище etcd – распределенное хранилище кластера высокой доступности в виде пар ключ–значения, в котором хранится состояние и конфигурация кластера. Согласованность (consistency) считается одним из самых важных качеств этой базы данных. Часто можно увидеть, как в production кластерах запущены несколько реплик etcd, объединенных в собственный кластер. Это позволяет создать кворум, лидер в котором будет выбираться, как чаще всего это происходит, по алгоритму нахождения консенсуса Raft.

Если в системе более одного главного узла только один из них будет ведущим, выполняющим все операции. Остальные master ноды будут ведомыми.

4.2 Рабочий узел

Рабочие узлы – серверы, на которых будут запускаться и работать контейнеризованные приложения. Данных узлов намного больше, чем управляющих. Компания Google советует для повышения отказоустойчивости создавать 3-7 мастер узлов в очень больших средах, остальные узлы будут рабочими. Для обеспечения полноценной работы у узла есть следующие компоненты:

– Среда выполнения контейнеров;

– kubelet – агент на каждом узле, через который идет обмен информацией с сервером API для управления всеми контейнерами на узле. Также он периодично проверяет, что контейнеры в поде, работают правильно и без сбоев;

– kube–proxy – служебный прокси K8S, который балансирует нагрузку сетевого трафика между компонентами приложения.

Для запуска и управления жизненным циклом контейнера необходима среда выполнения контейнера (container runtime) на рабочем узле.

Docker – это целая платформа, которая в своей реализации использует containerd как среду выполнения контейнеров.

Базовым строительным блоком в Kubernetes являются Поды(Pods).

Далее рассмотрим, какие ресурсы существуют для размещения своих рабочих нагрузок, чем они отличаются, и когда какой ресурс оптимально использовать.

5 Ресурсы, доступные для использования разработчиком

5.1 Pod

Pod представляет собой запрос на запуск одного или более контейнеров на одном узле. Все рабочие нагрузки в Kubernetes – Deployments, ReplicaSets и Jobs – могут быть выражены в виде pod'ов. Pod – единственный объект в k8s, который приводит к запуску контейнеров. Контейнер никак не может существовать без пода [10].

То есть под – минимальный блок, с которым можно работать в Kubernetes.

Обычно под содержит внутри себя от 1 до 5 контейнеров. Эти контейнеры могут взаимодействовать различными способами. В одном из разделов данной работы будут приведены 3 паттерна для настройки взаимодействия контейнеров внутри пода.

Далее будут рассмотрены ресурсы Kubernetes, позволяющие по–разному управлять набором подов в зависимости от требуемой задачи.

5.2 Replication controller

Контроллер репликации (Replication controller) – ресурс K8S, который занимается обеспечением постоянной работоспособности его модулей. Именно этот ресурс следит за теми случаями, когда под исчезает по причине отказа ноды кластера, либо вытеснения его с узла, и в этот момент он создает замену данному поду.

Контроллер следит за количеством реплик «своих» модулей. Когда в конфигурационном файле записывается определения контроллера репликации должны быть указаны селекторы (метки) подов, которые должны попасть под управление. Контроллер репликации следит за тем, чтобы количество реплик в его определении совпадало с реальным количеством реплик подов, которые он находит по селектору. Алгоритм работы контроллера представлен на рисунке 13.

В поле spec.selector указывается селектор, по которому контроллер будет находить поды. А в поле template.metadata.labels указываются метки самих подов. Разумеется, для корректной работы они должны совпадать.

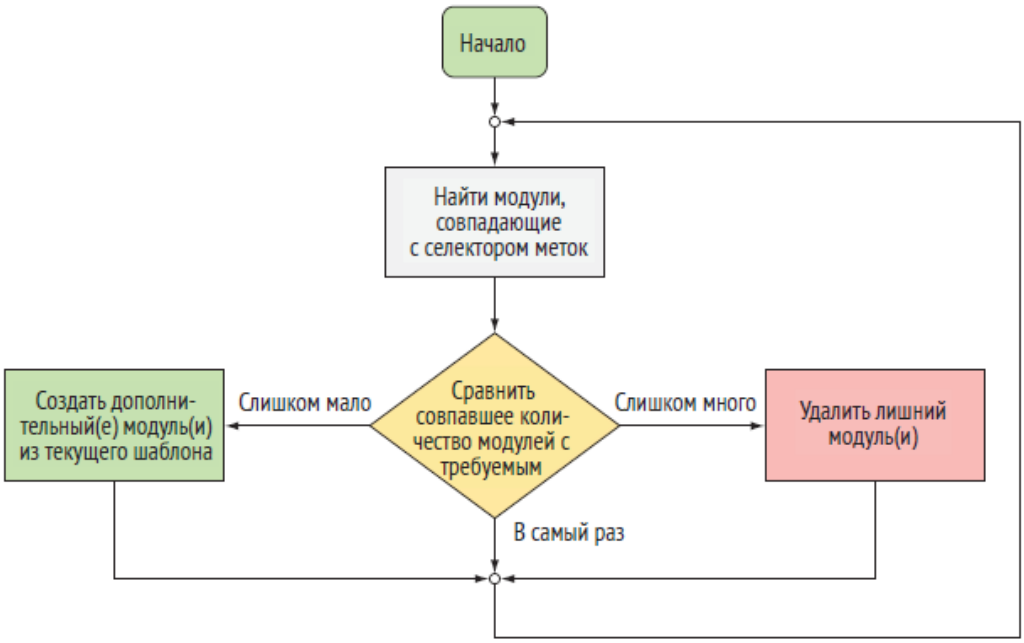

Рисунок 13 – Схема работы по приведению количества реплик подов к норме [11]

Пример файла конфигурации для определения Replication Controller’а представлен на рисунке 14.

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: example-app
spec:
  replicas: 3
  selector:
    app: example-app
  template:
    metadata:
      labels:
        app: example-app
    spec:
      containers:
      - name: example-app
        image: dennis/example-app
        ports:
        - containerPort: 8080
```

Рисунок 14 – Пример определения Replication Controller'a

В K8S была создана альтернатива для Replication Controller, и она называется ReplicaSet.

В чем же различие, и для чего нужно было создавать ресурс с почти такой же функциональностью?

5.3 ReplicaSet

ReplicaSet (набор реплик) – новое поколение контроллера репликации, его естественное развитие. Дело в том, что селекторы контроллера репликации позволяют выбирать поды только один способом: под должен включать в себя метку со значением. Такой гибкости не хватало, и в ReplicaSet были введены дополнительные возможности:

– matchLabels – Проверяет строгое соответствие ключа и значения меток модуля (как и у контроллера репликации);

– matchExpressions – обеспечивает поддержку более гибкой выборки подов. Описание каждого оператора приведено в таблице 2.

Оператор	**Выполняемая функция**
In	значение метки **должно** совпадать с одним из указанных значений values
NotIn	значение метки **не должно** совпадать с любым из указанных значений values
Exists	модуль **должен** содержать метку с указанным ключом (при этом значение не важно). При использовании этого оператора не следует указывать поле values
DoesNotExist	модуль **не должен** содержать метку с указанным ключом (значение не важно). При использовании этого оператора не следует указывать поле values

Таблица 2 – операторы matchExpressions

Пример использования оператора In в селекторе matchExpressions представлен на рисунке 15.

```
selector:
 matchExpressions:
 - key: app
   operator: In
   values:
   - example-app
```

Рисунок 15 – Пример использования оператора In в селекторе matchExpressions

В настоящее время всегда вместо контроллера репликации нужно использовать набор реплик, так как в будущем в K8S контроллер станет устаревшим и объявлен deprecated.

Обычно при работе с кластером ресурс ReplicaSet не создается вручную, этот набор используется в ресурсе более высокого уровня – Deployment.

Контроллер репликации (Replication controller) и набор реплик (Replica Set) при запуске подов не гарантируют нахождение последних на фиксированных узлах, потому что Scheduler (планировщик) распределяет поды в зависимости от загруженности узлов, и запросов на память и центральный процессор у подов. Но в K8S существует еще один ресурс для управления набором подов под названием DaemonSet (набор демонов).

5.4 DaemonSet

DaemonSet – такой ресурс Kubernetes, который позволяет запускать поды по одному на определенных узлах. По умолчанию, если никак не ограничивать его, то он будет запускаться на всех узлах, и даже тех, которые помечены как unschedulable (не назначаемые), потому что DaemonSet не управляется планировщиком для размещения подов.

Чтобы ограничить подмножество узлов, на которых нужно запускать поды, нужно установить свойство nodeSelector в шаблоне модуля, который описывается в .yaml файле DaemonSet.

Хорошим примером использования набора демонов является прокси самого K8S – kube–proxy. Ведь он должен работать на каждом узле. Также можно размещать в виде набора демонов системные процессы наподобие сбора метрик на всех узлах.

Сравнение работы запуска подов ReplicaSet и DaemonSet изображен на рисунке 16.

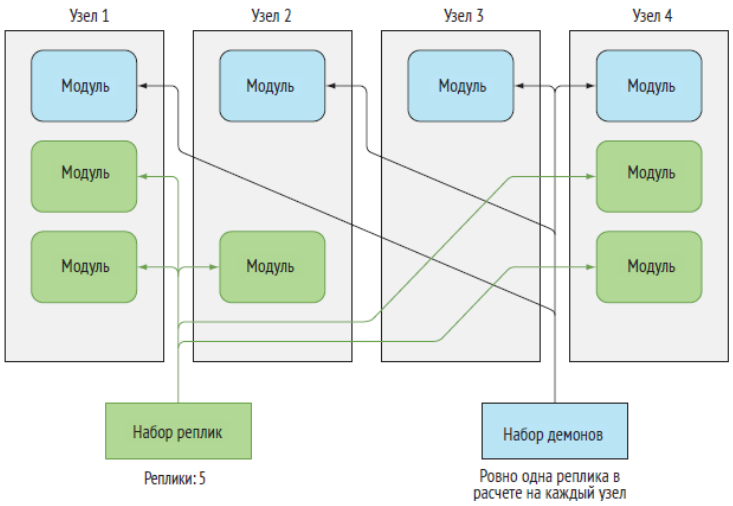

Рисунок 16 – Сравнение работы запуска подов ReplicaSet и DaemonSet

Пример описания набора демонов с селектором узлов, на которых установлен ssd disk, представлен на рисунке 17.

```yaml
apiVersion: apps/v1beta2
kind: DaemonSet
metadata:
  name: ssd-metrics-monitor
spec:
  selector:
    matchLabels:
      app: ssd-metrics-monitor
  template:
    metadata:
      labels:
        app: ssd-metrics-monitor
    spec:
      nodeSelector:
        disk: ssd
      containers:
      - name: main
        image: dennis/ssd-metrics-monitor
```

Рисунок 17 – Конфигурационный файл DaemonSet для размещения подов на узлах с ssd диском

5.5 StatefulSet

StatefulSet – ресурс в Kubernetes, позволяющий запускать приложения с сохранением состояния. Данный ресурс, в отличие от Replication Controller и ReplicaSet, по–другому работает с подами. Он позволяет запускать приложения, и обеспечить им стабильное хранилище, а также стабильное сетевое имя.

В случае, когда приложение не отвечает, либо же узел, на котором запущен данный под перестал работать – Kubernetes создаст другой под, но с таким же IP–адресом и внедрит то же самое хранилище. Это позволяет разворачивать любые приложения, даже кластеры баз данных в кластере Kubernetes.

Наличие именно этого ресурса в платформе заметно отличает Kubernetes от других платформ.

5.6 Service

Так как в мире Kubernetes модули (поды) это такое эфемерное понятие, разработчик не знает точно ни его ip адреса, ни dns имени, потому что они могут удаляться и создаваться очень часто – нужен механизм, позволяющий

как–то управлять подами, объединив их в группы. Таким механизмом являются сервисы или службы.

Службы следят за «своими» подами, как и все в кластере, с помощью селектора меток.

При создании пода, который соответствует селектору меток какой–либо службы, Kubernetes изменяет объект Endpoints, тем самым добавляя новый под в наблюдаемые для службы. Если есть такая необходимость, можно самому создать объект ресурса Endpoints и связать его со службой. Также можно посмотреть список IP адресов подов в объектах Endpoints простой командой: kubectl get endpoints <название службы>.

Существует несколько типов служб:

– NodePort – для такой службы на каждом узле кластера открывается порт и когда трафик поступает на данный порт узла, он перенаправляет полученный запрос в эту службу;

– ExternalName – для доступа к внешнему источнику, например API. Для этого типа службы необходимо выставить поле spec.externalName со значением того адреса, куда мы хотим отправлять трафик. Данный тип особенно удобен, потому что позволяет изменять поставщика какого–либо API всего лишь в одном месте, без затрагивания настроек других рабочих нагрузок кластера;

– LoadBalancer – балансировщик нагрузки.

Service Discovery (обнаружение служб) осуществляется с помощью кластерного DNS сервера, под названием kube–dns, запущенного в пространстве имен kube–system. DNS позволяет обращаться к службам с помощью полностью квалифицированного доменного имени (Fully Qualified Domain Name).

FQDN имеет следующий вид:

<название службы>.<namespace>.<доменный суффикс кластера>

Примером FQDN является следующее доменное имя:

asp–net–backend.production.svc.cluster.local

где svc.cluster.local – доменный суффикс кластера для локальных служб кластера Kubernetes.

При использовании разных технологий для взаимодействия в реальном времени, наподобие web sockets, нужно чтобы службы сохраняли сессию, и запросы перенаправлялись от одного и того же клиента к той же самой службе. Это возможно сделать с помощью свойства spec.sessionAffinigty со значением ClusterIP.

5.7 Job и CronJob

В работе любой системы существуют специальные типы приложений, который запускаются для определенной цели. Часто нужно запускать консольное приложение для какой–либо синхронизации, если произошла

интеграция с внешним ресурсом. Такие задачи нужно запускать, например, раз в день. Но такой вид приложения не должен оставаться запущенным после проведения своей основной работы, оно должно быть остановлено.

Именно для таких целей в Kubernetes предусмотрен отдельный вид ресурсов, именуемый Job.

Job (задание) также запускает модуль (под), и приложение не перезапускается, если модуль отработал успешно. В случае же неуспешного завершения работы можно настроить, что будет происходить дальше – либо перезапуск контейнера, либо нет.

Также стоит учитывать, что если узел, на котором запущен модуль, управляемый Job, завершает свою работу аварийно, то данный модуль будет перезапущен на другом узле, как если бы модуль управлялся ReplicaSet.

В определение задания есть поле restartPolicy, которое указывает Kubernetes, перезапускать ли под при завершении задания. У обычного пода по умолчанию значение поля было бы равно Always, но поды, управляемые Job, не могут использовать это значение, так как этот ресурс, не задумывался как процесс, который работает постоянно. Поэтому нужно указывать либо значение OnFailure, либо Never.

Пример файла конфигурации описания Job представлен на рисунке 18.

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: data-synchronization
spec:
  template:
    metadata:
      labels:
        app: data-synchronization
    spec:
      restartPolicy: OnFailure
      containers:
      - name: main
        image: dennis/data-synchronization
```

Рисунок 18 – Пример файла конфигурации Job

В спецификации задания можно указывать поле completions со значением равным числу. Это поле указывает Kubernetes, что данный под нужно запустить последовательно указанное количество раз. Сначала создается один под, и когда он успешно отработает, то при его уничтожении создается новый, и так будет продолжаться, пока все задания не будут завершены успешно.

Также если разрешено запускать несколько подов задания параллельно, то нужно указать свойство spec.parallelism с числом возможных параллельных заданий.

В платформе K8S есть еще один тип задания, которое нужно для запуска задания по расписанию, и он называется CronJob.

CronJob – ресурс Kubernetes, в котором в поле spec.schedule указывается время запуска задания в специальном формате.

Формат поля schedule имеет следующий вид: * * * * *.

Если смотреть слева направо, то задаются 5 значений: минута, час, день месяца, месяц, день недели.

* – означает «в каждый» (каждую минуту, каждый час и т.д.).

С помощью такого формата можно записать следующее расписание: запускать задание каждое воскресенье, только в 4 часа ночи.

Поле Schedule будет выглядеть следующим образом:

0 4 * * 0.

Такая запись очень удобна, ведь с ее помощью можно задать абсолютно любое расписание, и любые синхронизации будут выполнены в точный срок.

5.8 Deployment

Для того, чтобы запускать приложения в кластере, нужно оперировать различными ресурсами, например, можно это делать с помощью набора реплик. Но для того, чтобы запустить обновленный вариант своего приложения, нужно как–то управлять трафиком, который поступает к приложению. Сначала нужно запустить набор реплик с контейнерами новой версии приложения, затем перенаправить трафик на эту версию, и потом выполнять удаление старой версии набора реплик. Чтобы упростить и автоматизировать этот процесс был создан специальный ресурс под названием Deployment.

Deployment (развертывания) – ресурс более высшего уровня, чем набор реплик или контроллер репликации, позволяющий проводить развертывания и обновления приложения с помощью декларативного подхода. Внутри своего механизма работы он использует наборы реплик.

Схема отношения ресурсов представлена на рисунке 19.

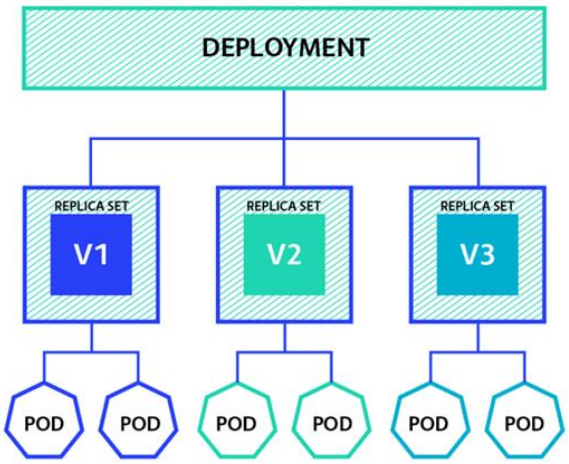

Рисунок 19 – Схема отношения используемых ресурсов в Deployment

Чтобы развернуть приложение с 3 репликами нужно описать конфигурацию, показанную на рисунке 20.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: asp-net-app
spec:
  replicas: 3
  template:
    metadata:
      name: asp-net-app
      labels:
        app: asp-net-app
    spec:
      containers:
      - image: dennis/asp-net-app:v1
        name: asp-net-app
```

Рисунок 20 – Пример файла конфигурации развертывания, создающего приложение с 3 репликами

5.9 ConfigMap

ConfigMap – ресурс Kubernetes, позволяющий устанавливать переменные окружения для приложения. ConfigMap – ассоциативный массив, содержащий пары ключ–значение.

Этот ресурс дает преимущество в том, что теперь можно не задавать жестко переменные окружения, а для разных окружений задавать разные ConfigMap, но с одним и тем же именем, и тогда можно использовать один и тот же yaml файл, но при запуске приложения будут разные настройки. Схема того, как можно использовать разные конфигурации с одним именем ConfigMap’а представлена на рисунке 21.

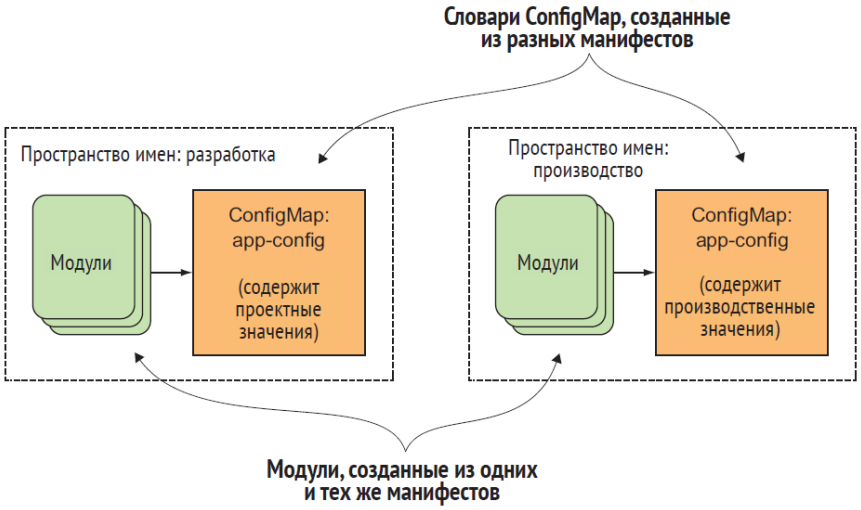

Рисунок 21 – Пример настройки установки переменных окружения для разных namespace’ов

5.10 Secret

Для разных систем существует необходимость в хранении данных, которые нужно защитить от доступа других лиц. Это могут быть учетные данные для доступа к базам данных или закрытые ключи шифрования. Именно для таких нужд в Kubernetes был создан отдельный объект под названием секреты. Они также содержат пары ключ–значение, как и ConfigMap и также можно передавать в качестве переменных окружения записи секретов. Для обеспечения безопасности Kubernetes распространяет секреты только на те серверы, где запущены поды, которые нуждаются в секретах. Также они никогда не записываются на жесткий диск, а хранятся всегда в оперативной памяти. Раньше секреты хранились в etcd в незашифрованном виде, однако, начиная с версии Kubernetes 1.7 эту проблему исправили.

В секрете информация кодируется с помощью base64, и раскодируется только тогда, когда идет монтирование файла в модуль, либо задаются переменные среды. Получается для конечного пользователя нет разницы в получении данных из ConfigMap и из Secret. Стоит помнить, что объем секрета не может превышать порог в 1Мб.

5.11 Namespace

Namespace – объект, который позволяет разграничить кластер, разбив его на разные пространства, и как бы создавая виртуальный кластер в одном физическом. Часто пространства имен используются для разделения разных окружений, например, dev, stage, production. Это позволяет легче смотреть ресурсы, запущенные в разных окружениях просто добавляя параметр –

namespace или –n, либо установив один раз контекст исполнения команд в kubectl.

Данный объект позволяет выделить необходимый объем ресурсов кластера для определенной системы микросервисов либо приложения с помощью ResourceQuota, т.е. можно задать ограничение для всего «виртуального кластера».

6 Установка и конфигурирование кластера Kubernetes

На сегодняшний день существует множество утилит для создания кластера Kubernetes. К ним относится Kind, который позволяет развернуть кластер в контейнере Docker. Стоит заметить, что именно с помощью этой утилиты сами разработчики Kubernetes тестируют новые версии K8S, эмулируя кластер с 5000 нод (серверов). Для локальной разработки подойдет minikube, который состоит из всего одной ноды, и позволяет быстро проверить работоспособность какой–либо гипотезы. Для production кластера принято использовать утилиту kubeadm, или Kubespray, который является некоей надстройкой над kubeadm, и позволяет развернуть кластер сразу на многих серверах, так как в его работе используется Ansible, позволяющий совершать монотонные действия на разных машинах автоматизировано.

В своей работе я использовал утилиту kubeadm, она буквально за несколько шагов позволяет развернуть production–ready кластер и сразу же приступить к работе с ним.

Для того, чтобы можно было начать создание кластера, нужен сервер, с установленной операционной системой Ubuntu, или CentOS, а также доступ к ней по ssh. В данном случае на сервере установлена Ubuntu 18.04 (LTS).

Далее приведены шаги для разворачивания кластера Kubernetes:

1. Для начала нужно отключить раздел подкачки, командой, показанной на рисунке 22.

```
swapoff -a
```

Рисунок 22 – Команда для отключения раздела подкачки

2. Далее добавляем ключ Google apt репозитория gpg, как показано на рисунке 23.

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
```

Рисунок 23 – Добавление ключа Google apt репозитория gpg

3. Также нужно добавить Kubernetes apt репозиторий в локальный лист репозиториев, как показано на рисунке 24.

```
sudo bash -c 'cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF'
```

Рисунок 24 – Добавление Kubernetes apt репозитория

4. Далее нужно обновить пакеты, как показано на рисунке 25.

```
sudo apt-get update
```

Рисунок 25 – Обновление пакетов

Также можно посмотреть, какие версии пакетов доступны для установки kubelet и docker, командой, изображенной на рисунке 26.

```
apt-cache policy kubelet | head -n 20
apt-cache policy docker.io | head -n 20
```

Рисунок 26 – Просмотр доступных версий пакетов

5. Устанавливаем необходимые пакеты для работы кластера, как изображено на рисунке 27.

```
sudo apt-get install -y docker.io kubelet kubeadm kubectl
```

Рисунок 27 – Установка kubelet, docker, kubeadm, kubectl

6. Далее эти пакеты нужно пометить для того, чтобы они не могли обновляться как все остальные пакеты командой apt–get update, как

изображено на рисунке 28. Это важно, так как версии должны оставаться теми, которые мы задали при установке.

```
sudo apt-mark hold docker.io kubelet kubeadm kubectl
```

Рисунок 28 – Пометка установленных пакетов для сохранности версии

7. Нужно проверить, что службы kubelet и docker имеют статус «включен», это проверяется командой, изображенной на рисунке 29.

```
sudo systemctl status kubelet.service
sudo systemctl status docker.service
```

Рисунок 29 – Проверка статуса служб docker и kubelet

Результат выполнения команд показан на рисунках 30 и 31.

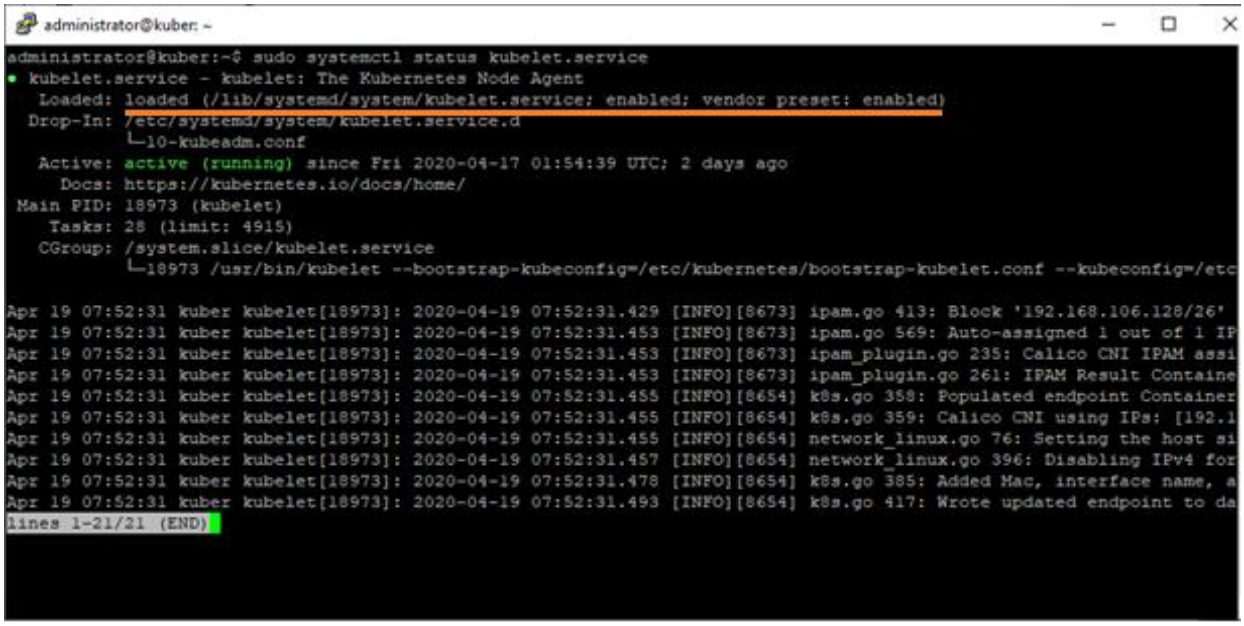

Рисунок 30 – Просмотр статуса службы kubelet

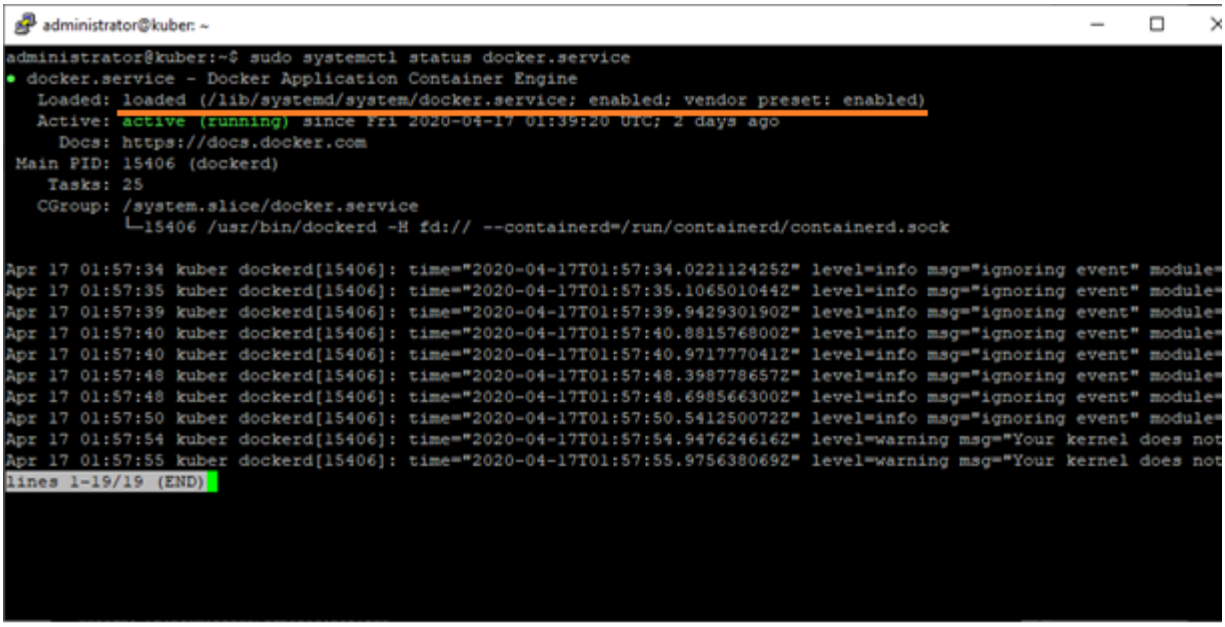

Рисунок 31 – Просмотр статуса службы docker

В случае если статус указан «disabled», то нужно запустить службы командами, показанными на рисунке 32.

```
sudo systemctl enable kubelet.service
sudo systemctl enable docker.service
```

Рисунок 32 – Команды для запуска служб

8. Устанавливаем Docker daemon, как показано на рисунке 33.

```
sudo bash -c 'cat > /etc/docker/daemon.json <<EOF
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2"
}
EOF'
```

Рисунок 33 – Установка Docker daemon

Как изображено на рисунке 34, нужно сразу же перезапустить Docker.

```
sudo systemctl daemon-reload
sudo systemctl restart docker
```

Рисунок 34 – Перезапуск Docker

Подготовительные работы закончились, теперь можно приступить непосредственно к созданию кластера и настройке master node.

1. Чтобы настроить сеть в кластере существует готовое решение под названием Calico, и нужно скачать yaml файл для применения настроек к кластеру, как показано на рисунке 35.

```
wget https://docs.projectcalico.org/manifests/calico.yaml
```

Рисунок 35 – Скачивание файла проекта Calico

Нужно открыть скачанный файл calico.yaml и найти там поле CALICO–IPV4POOL–CIDR, оно понадобится для создания кластера.

2. Инициализируем кластер командой, показанной на рисунке 36.

```
sudo kubeadm init --pod-network-cidr=192.168.0.0/16
```

Рисунок 36 – Инициализация кластера

где 192.168.0.0/16 – значение из поля CALICO_IPV4POOL_CIDR файла calico.yaml

После инициализации можно увидеть сообщение об успешности процесса, как показано на рисунке 37.

```
Your Kubernetes master has initialized successfully!
```

Рисунок 37 – Сообщение об успешности инициализации кластера

3. Далее нужно создать пользователя для Kubernetes, как показано на рисунке 38.

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Рисунок 38 – Создание пользователя admin для работы в кластере

4. И теперь нужно применить файл calico.yaml для установки конфигурации сети в кластере командой, изображенной на рисунке 39.

```
kubectl apply -f calico.yaml
```

Рисунок 39 – Установка конфигурации

Кластер работает, и теперь можно разворачивать в нем необходимые приложения.

Для качественной работы можно настроить dashboard, и тогда можно будет следить за процессами, происходящими в кластере, из удобного интерфейса, как показано на рисунке 40.

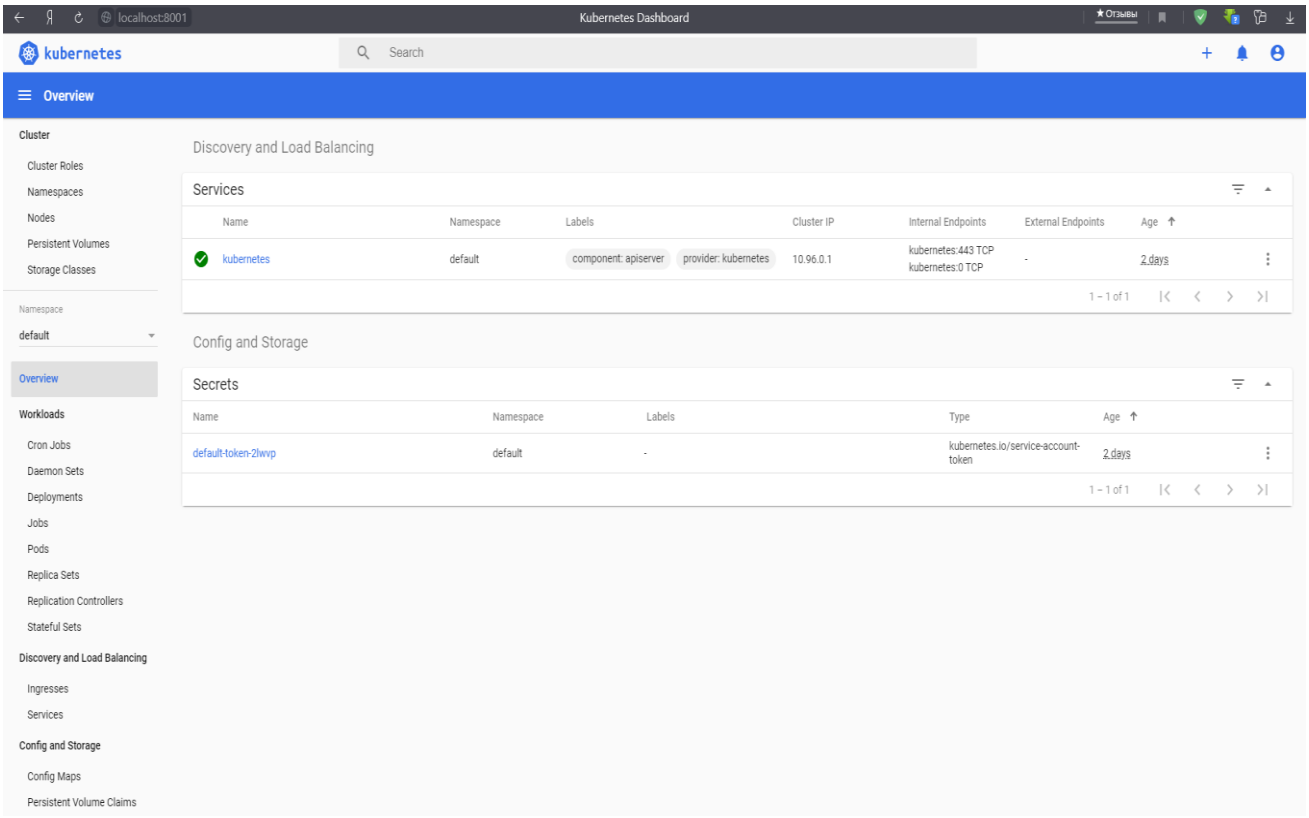

Рисунок 40 – Сообщение об успешности инициализации кластера

7 Концепции оптимального построения приложений в Kubernetes

Для того, чтобы система, развернутая в Kubernetes, была отказоустойчивой, ее нужно правильно спроектировать. В частности, в каждом приложении нужно указывать проверки готовности, и проверки «живучести» приложения.

Проверка живучести нужна для того, чтобы платформа могла узнавать периодически, «живо» ли приложение или нужно его перезапустить. Проверка готовности же нужна для того, чтобы трафик, который идет на определенный маршрут, шел именно на готовые к приему данного трафика подам. Потому что часто запуск приложения занимает немалое время, и, если за этим не следить, трафик пойдет в под, который еще не успел запуститься.

В Kubernetes существует несколько вариантов создания проверки живучести:

– проверка HTTP GET выполняет запрос на IP адрес, порт и путь контейнера, которые указываются в описании пода;

– проверка сокета TCP пытается открыть подключение к указанному порту контейнера. Если подключение установлено успешно, то проверка считается сработавшей. В противном случае контейнер внутри пода будет перезапущен;

– проверка EXEC выполняет указанную произвольную команду внутри контейнера пода, и проверяет ответ, код, который возвращает команда. Проверка считается сработавшей только в случае, если код равен 0 [11].

Ниже на рисунке 41 представлен фрагмент кода, с помощью которого указывается проверка живучести.

```
1   apiVersion: v1
2   kind: Pod
3   metadata:
4     name: container-liveness
5   spec:
6     containers:
7     - image: dennisaltaev/kubia-unhealthy
8       name: kubia
9       livenessProbe:
10        httpGet:
11          path: /
12          port: 8080
```

Рисунок 41 – Пример проверки живучести для контейнера

Также можно установить задержку перед началом «опроса» контейнера. Свойство называется initialDelaySeconds. Как понятно из названия, время задаётся в секундах.

С точки зрения быстродействия нужно помнить, что проверки живучести не должны быть слишком ресурсозатратными и должны отрабатывать в кратчайшие сроки.

Проверка готовности имеют такие же типы, как и проверки живучести. Нужно всегда добавлять проверки готовности в модули, потому что, если не добавить проверку, то модуль почти сразу станет конечной точкой, которая должна обрабатывать входящий трафик, а фактически модуль будет не готов, и это приведет к серьезным проблемам. А ведь бывают приложения, процесс инициализации которых требует много времени, пока пройдет подготовка базы данных, прогреется кэш приложения и т.д.

Иногда стоит задача размещения подов на узлах, с определенными характеристиками, например, на которых стоит высокопроизводительная графическая карта (GPU) для ускорения проведения вычислений. В таком случае нужно установить метку для узла: GPU: true, и далее можно в подах указывать поле spec.nodeSelector с таким же значение GPU: true. После этих действий планировщик будет планировать такие поды только на соответствующих узлах.

Также в Kubernetes есть продвинутые практики, позволяющие размещать поды на серверах, с определенной временной зоной, или определенной страной, или нужным дата–центром, и даже на нужной стойке в дата–центре и все это в зависимости от расположение других подов. Такие техники позволяют выразить близость и удаленность подов, которая иногда необходима, когда разработчик хочет разместить поды рядом, чтобы, например, микросервисы одного приложения работали в одной группе. Такой функционал возможно реализовать с помощью полей spec.affinity.podAffinity и spec.affinity.podAntiAffinity полей в определении подов:

– podAffinity – задаются условия для выборки подов, рядом с которыми нужно размещать данный под;

– podAntiAffinity – задаются условия для выборки подов, рядом с которыми нельзя размещать данный под.

Иногда взаимодействие по сети является критичным в плане времени исполнения. При такой необходимости можно указать с помощью поля spec.affinity.nodeAffinity. Для этого поля существуют обязательные требования и предпочтительные: requiredDuringSchedulingIgnoredDuringExecution и preferredDuringSchedulingIgnoredDuringExecution соответственно. Окончание обоих требований IgnoredDuringExecution указывает на то, что под не будет перемещен во время выполнения, если узел изменится.

В следующей главе будут рассмотрены паттерны, позволяющие управлять взаимодействием между разными контейнерами, разворачиваемыми в одном поде.

8 Паттерны расположения контейнеров в подах.

Существует несколько паттернов для настраивания взаимодействия контейнеров в одном поде:

– Sidecar;
– Adapter;
– Ambasador.

8.1 Паттерн Sidecar

Паттерн Sidecar (прицеп) является базовым паттерном для Адаптера и Посредника (Ambassador).

Паттерн Sidecar (Прицеп) заключается в определении контейнера, который расширяет возможности существующего контейнера без его изменения. Это один из основополагающих паттернов контейнеров, который позволяет создавать узкоспециализированные контейнеры, тесно взаимодействующие друг с другом [12].

На рисунке 42 представлена схема работы контейнеров при использовании sidecar паттерна.

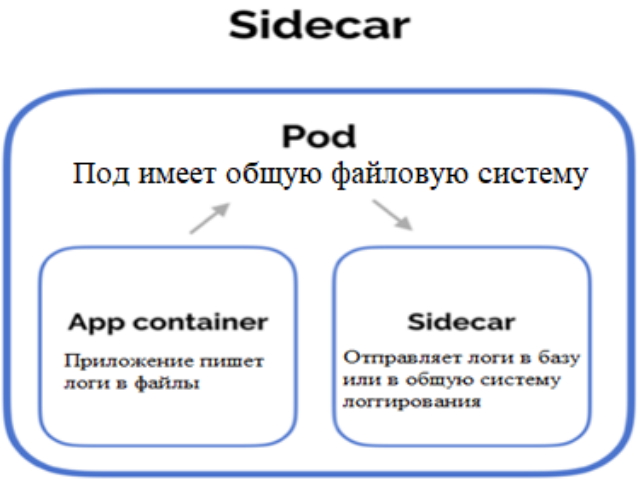

Рисунок 42 – Схема работы контейнеров в sidecar паттерне

Так как контейнеры – сущность, содержащая определенную функциональность, у которой есть своя среда выполнения, свои ресурсы. Контейнер инкапсулирует в себе работу одной команды, и приложение, находящееся внутри, развертывается самостоятельно, следовательно, лучшим вариантом добавления каких–либо дополнительных функций будет вынос этих функций в другой контейнер, запуск нового контейнера в поде рядом с основным, и предоставления возможности взаимодействия между основным и

дополнительным контейнером. Таким образом можно выделять однотипные функции в дополнительные контейнеры, и компоновать свои приложения с помощью нескольких необходимых. Паттерн Sidecar как раз предоставляет такую возможность в разработке различных систем.

При развертывании в Kubernetes пода с несколькими контейнерами внутри они имеют в своем распоряжении общие ресурсы в виде общих томов (жесткий диск). Также запросы могут обрабатываться через локальную сеть, что позволяет задать фиксированный адрес для отправки и приема запросов через localhost:<номер порта>.

Хорошим примером использования паттерна Sidecar является проект Istio. В нем реализованы множество функций в том числе авторизация, аутентификация, трассировка и повторные вызовы именно с помощью этого паттерна.

8.2 Паттерн Adapter

Паттерн Adapter (Адаптер) позволяет привести гетерогенную контейнерную систему в соответствие с единообразным унифицированным интерфейсом, имеющим стандартизованный и нормализованный формат, который может использоваться внешним миром. Паттерн Adapter (Адаптер) наследует все свои характеристики от паттерна Sidecar (Прицеп), но имеет единственную цель – предоставить адаптированный доступ к приложению [12].

На рисунке 43 представлена схема работы контейнеров при использовании паттерна адаптер.

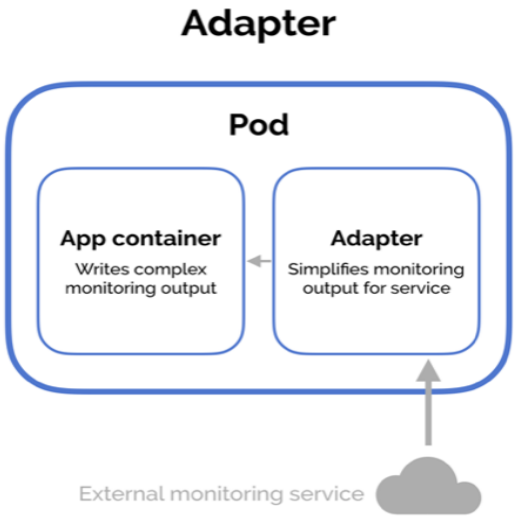

Рисунок 43 – Схема работы контейнеров в паттерне адаптер [12]

Одним из преимуществ разработки с использованием микросервисной архитектуры является то, что каждая команда, разрабатывающая отдельный микросервис, может выбирать нужный именно ей стек технологий. Это очень удобно, ведь один язык программирования может быть лучшим в определенных задачах, и почему же не использовать именно его. Во времена, когда микросервисы были не так сильно развиты, разработчики были вынуждены решать все свои задачи на одном стеке, одном языке, и это очень сильно ограничивало в действиях и вариантах.

Так вот такая возможность приносит кое-какие сложности в работе облачного решения. К примеру, если в системе много разных технологий на разных проектах, но мониторинг происходит централизовано, например, с помощью Prometheus или любой другой технологии, то нужно как–то стандартизировать формат логирования. Именно паттерн Адаптер позволяет это сделать. То есть при каждом запросе сервера Prometheus вспомогательный контейнер будет читать логи, преобразовывать их в нужный формат, и только после этого возвращать ответ серверу.

Паттерн Adapter (Адаптер) – это специализированный вариант паттерна Sidecar (Прицеп). Он действует как обратный прокси для гетерогенной системы, скрывая ее сложность за унифицированным интерфейсом [12].

8.3 Паттерн Ambassador

Паттерн Ambassador (Посредник) – это специализированный вариант паттерна Sidecar (Прицеп), отвечающий за сокрытие сложности и предоставляющий унифицированный интерфейс для доступа к службам за пределами пода [12].

Мир микросервисов очень сложен, и один сервис не может существовать без зависимостей от других служб. Так как микросервисы проектируются таким образом, чтобы выполнять лишь одну важную функциональность, то будет неверным решением реализовывать логику работы с внешними службами в этом же микросервисе. В мире облачных разработок принято такую логику выносить в отдельный контейнер, чтобы не перегружать основной, и иметь возможность переиспользовать данный функционал. Также бывает полезным во время разработки, когда сервис запущен для тестирования функций системы, подменять сервисы на другие, отличные от промышленного окружения. Именно этим целям и служит паттерн Посредник (Ambassador).

На рисунке 44 представлена схема работы контейнеров при использовании паттерна посредник.

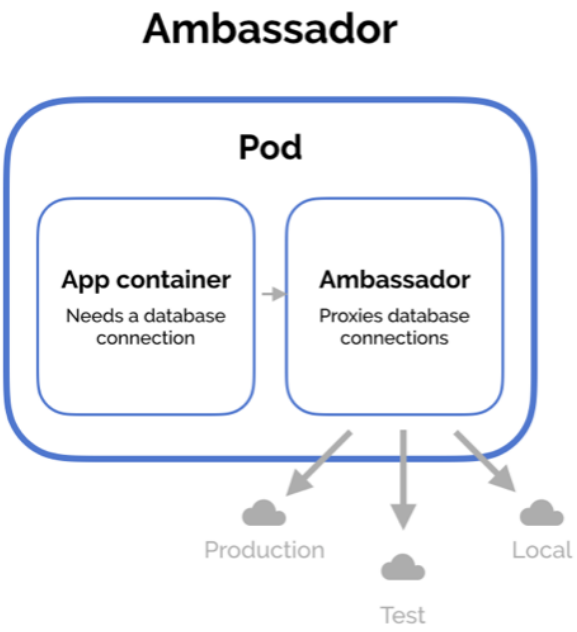

Рисунок 44 – Схема работы контейнеров в паттерне посредник [12]

В следующем разделе будут рассмотрены стратегии развертывания, которые позволяют максимально улучшить пользовательский опыт работы с системой, и убрать простои при обновлении на новую версию.

9 Стратегии развертывания микросервисов в Kubernetes

В современной разработке одной из самых частых и больших проблем при разработке приложений на основе микросервисной архитектуры является ускорение процесса развертывания системы.
В Kubernetes существует несколько основных типов стратегий развертывания систем. К ним можно отнести:

– rolling update;
– recreate;
– blue/green.

Давайте разберем каждую из них подробнее.

9.1 Rolling (постепенный деплой)

Эта стратегия является стандартной. Она работает, заменяя поды со старой версией на поды с новой версией, причем без простоя кластера. Этого позволяют добиться проверки готовности, так называемые readiness пробы. Kubernetes ждет, когда только что запущенный под будет несколько раз стабильно возвращать успешное выполнение проверки готовности, и только после этого начинается уничтожение подов со старой версией [13].

Схема работы постепенной стратегии развертывания показана на рисунке 45.

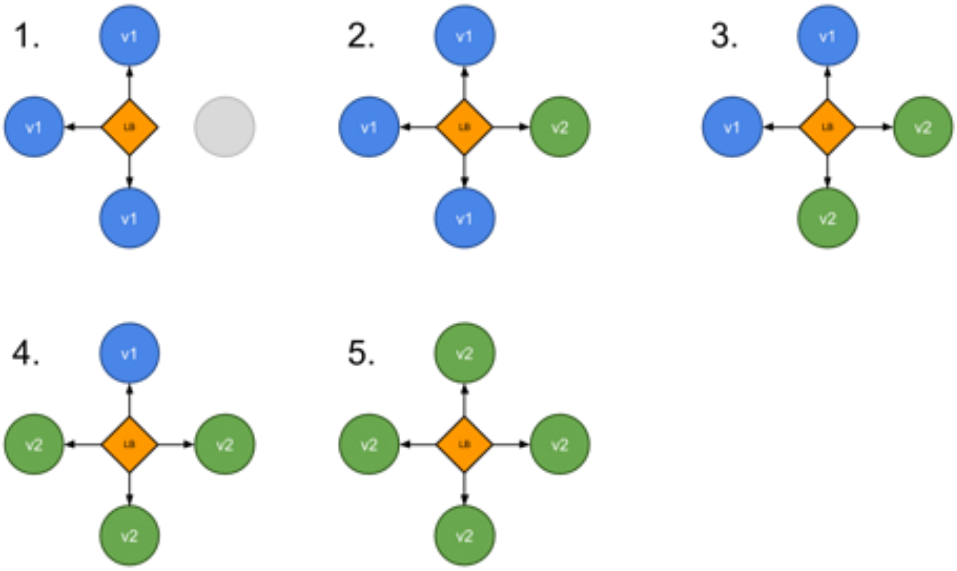

Рисунок 45 – Схема работы постепенной стратегии развертывания [13]

Спецификация для постепенной стратегии развертывания показана на рисунке 46.

```
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
       maxSurge: 25%
       maxUnavailable: 25%
  template:
```

Рисунок 46 – Описание постепенной стратегии развертывания

Здесь стоит отметить 2 важных параметра: maxSurge и maxUnavailable:

– maxSurge – определяет, скольким экземплярам пода вы позволяете существовать сверх нормы количества реплик, которое установлено в deployment’е. По умолчанию используется значение 25%. При преобразовании в абсолютное число, это значение округляется вверх. Также можно устанавливать сразу абсолютные значения [11];

– maxUnavailable – определяет, сколько экземпляров пода может быть недоступно, относительно указанного количества реплик во время обновления на новую версию приложения. По умолчанию также равно 25%. Но при преобразовании в число, это значения будет округлено вниз [11].

Экспериментируя с этими двумя значениями можно контролировать процесс развертывания в зависимости от мощностей кластера и необходимой скорости развертывания.

9.2 Recreate (повторное создание)

В этой стратегии развертывания старые поды разом уничтожаются и заменяются на поды с новой версией приложения. Минусом такого развертывания является наличия времени простоя, пока новые приложения запустятся, пройдут проверки готовности, и начнут принимать трафик от пользователей. Спецификация для стратегии повторного создания подов показана на рисунке 47.

```
spec:
  replicas: 5
  strategy:
    type: Recreate
  template:
```

Рисунок 47 – Описание стратегии повторного создания подов

9.3 Blue/Green (сине–зеленые развертывания)

Стратегия сине–зеленого развертывания подразумевает одновременное развертывание старой (зеленой) и новой (синей) версий приложения. После размещения обеих версий обычные пользователи получают доступ к зеленой, в то время как синяя доступна для QA–команды для автоматизации тестов. Схема работы сине–зеленого развертывания изображена на рисунке 48.

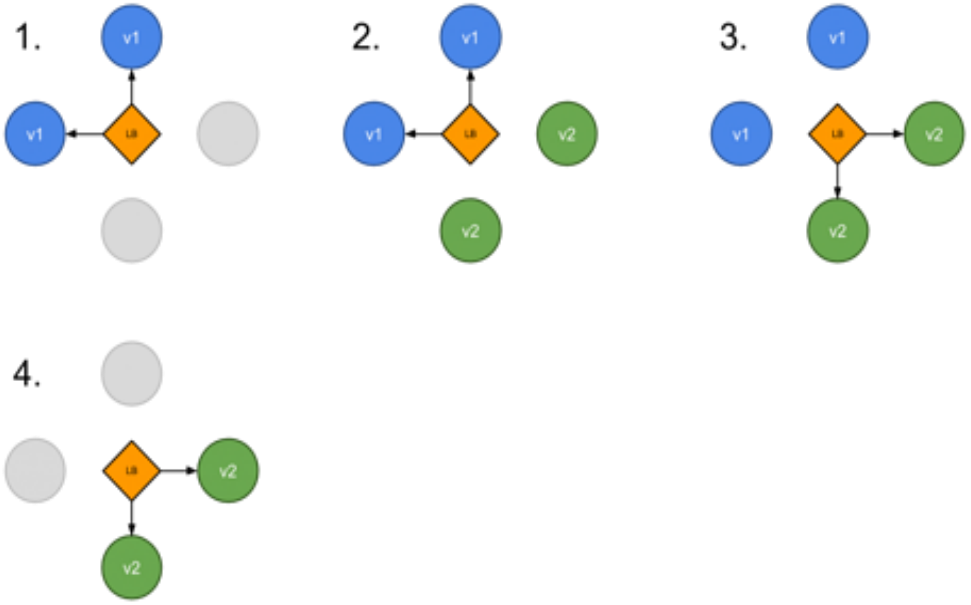

Рисунок 48 – Схема работы сине–зеленой стратегии развертывания [13]

Переключение на новую версию происходит при помощи изменения селектора меток, за которыми наблюдает сервис. То есть у подов с новой версией будет указан номер версии, и у сервиса также изменится только данный селектор.

Для каждого конкретного случая выбираются различные стратегии обновления подов приложения.

В следующем разделе будет рассмотрена тема настройки автомасштабирования подов для того, чтобы повысить отказоустойчивость системы в целом.

10 Масштабирование приложений в Kubernetes

Еще одним большим плюсом в использовании Kubernetes является довольно простая настройка масштабирования модулей. Горизонтальное автомасштабирование модулей – автоматический процесс изменения количества реплик модуля, управляемого контроллером.

Процесс автомасштабирования можно разделить на 3 стадии:

– получение метрик всех модулей, которые находятся в области действия ресурса контроллера репликации;

– расчёт нужного количества модулей для более эффективной работы системы;

– обновления поля replicas необходимого модуля.

Все метрики на каждом модуле считывает cAdvisor, которые после агрегируются при помощи кластерного компонента Heapster. Если проект будет слишком быстро усложняться, то можно внедрить другой инструмент для агрегации метрик и состояния системы под названием Prometeus. Очень хорошо работает связка с Grafana – решение с открытым исходным кодом для визуализации любых метрик системы из базы данных временных рядов.

Следует упомянуть о некоторых параметрах при настройке автомасштабирования: minReplicas, maxReplicas. Как понятно из названия – это ограничения по количеству реплик модуля. Также для того, чтобы Kubernetes понимал нужно ли масштабировать модуль, в определении автомасштабирования нужно указывать сколько памяти и CPU нужно для полноценной работы модуля [14].

Horizontal Pod Autoscaler – горизонтальное автомасштабирование подов – возможность Kubernetes, которая появилась в версии 1.1. Первая реализация могла совершать масштабирование на основе данных о потреблении ресурсов центрального процессора, и только позже на основе использования оперативной памяти.

В Kubernetes версии 1.6 добавлена поддержка для использования пользовательских метрик в горизонтальном автомасштабирования подов. Вы можете добавить пользовательские метрики для HPA, чтобы использовать их в API autoscaling/v2beta2 [15].

Horizontal Pod Autoscaler опрашивает с определенным периодом времени основные показатели, как использование памяти и процессора в Resource Metrics API, а чтобы получить пользовательские (кастомные) метрики опрашивает Custom Metrics API [15]. Схема взаимодействия HPA с Prometheus показана на рисунке 49.

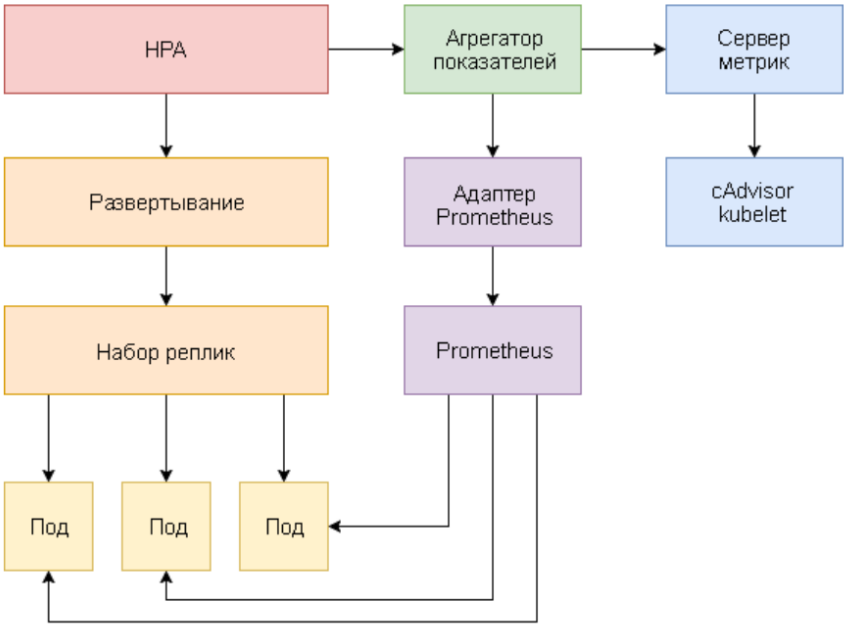

Рисунок 49 – Схема работы HPA совместно с Prometheus

На смену Heapster, что был раньше в Kubernetes, и собирал данные об использовании ресурсов кластера, пришел сервер метрик. Сервер метрик – агрегатор, собирающий данные об использовании памяти и процессора из специального API – Summary API. Он эффективно расходует память, что позволяет работать в фоне, и не ощутимо нагружать кластер, и служит для передачи собранных данных с Kubelet и cAdvisor. Схема взаимодействия горизонтального автомасштабирования подов с сервером метрик показана на рисунке 50.

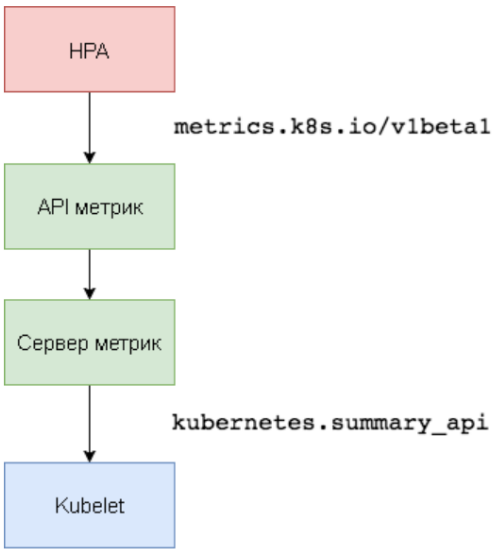

Рисунок 50 – Схема взаимодействия горизонтального автомасштабирования подов с сервером метрик

На рисунках 51 и 52 показаны ресурсы, которые нужно создать для работы объекта horizontal pod autoscaler.

```yaml
# файл auth-delegator.yaml
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: metrics-server:system:auth-delegator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:auth-delegator
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
---
# файл auth-reader.yaml
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: metrics-server-auth-reader
  namespace: kube-system
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: extension-apiserver-authentication-reader
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
```

```yaml
# файл resource-reader.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: system:metrics-server
rules:
- apiGroups:
  - ""
  resources:
  - pods
  - nodes
  - nodes/stats
  - namespaces
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - "extensions"
  resources:
  - deployments
  verbs:
  - get
  - list
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: system:metrics-server
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:metrics-server
subjects:
- kind: ServiceAccount
  name: metrics-server
  namespace: kube-system
```

Рисунок 51 – Роли и привязки ролей для Metrics Server

```yaml
---
# файл metrics-server-deployment.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: metrics-server
  namespace: kube-system
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: metrics-server
  namespace: kube-system
  labels:
    k8s-app: metrics-server
spec:
  selector:
    matchLabels:
      k8s-app: metrics-server
  template:
    metadata:
      name: metrics-server
      labels:
        k8s-app: metrics-server
    spec:
      serviceAccountName: metrics-server
      volumes:
      # mount in tmp so we can safely use from-scratch images and/or read-only containers
      - name: tmp-dir
        emptyDir: {}
      containers:
      - command:
        - /metrics-server
        - --kubelet-insecure-tls
        - --kubelet-preferred-address-types=InternalIP
        name: metrics-server
        image: k8s.gcr.io/metrics-server-amd64:v0.3.1
        imagePullPolicy: Always
        volumeMounts:
        - name: tmp-dir
          mountPath: /tmp
```

```yaml
---
# файл metrics-apiserver.yaml
apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: v1beta1.metrics.k8s.io
spec:
  service:
    name: metrics-server
    namespace: kube-system
  group: metrics.k8s.io
  version: v1beta1
  insecureSkipTLSVerify: true
  groupPriorityMinimum: 100
  versionPriority: 100
---

# файл metrics-server-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: metrics-server
  namespace: kube-system
  labels:
    kubernetes.io/name: "Metrics-server"
spec:
  selector:
    k8s-app: metrics-server
  ports:
  - port: 443
    protocol: TCP
    targetPort: 443
```

Рисунок 52 – Ресурсы Service Account, Deployment, APIService, Service для Metrics Server

Как только Kubernetes развернет все эти ресурсы для сервера метрик, начнется передача данных о том, как используются процессор и память на узлах и подах.

Для того, чтобы просмотреть показатели на подах, нужно выполнить команду, показанную на рисунке 53:

```
kubectl get --raw "/apis/metrics.k8s.io/v1beta1/pods" | jq .
```

Рисунок 53 – Команда для просмотра использования ресурсов на подах кластера

А чтобы посмотреть использование ресурсов на узлах кластера, нужно выполнить команду, показанную на рисунке 54:

```
kubectl get --raw "/apis/metrics.k8s.io/v1beta1/nodes" | jq .
```

Рисунок 54 – Команда для просмотра использования ресурсов на узлах кластера

Теперь, когда сервер метрик уже работает, можно приступить к развертыванию ресурса Horizontal Pod Autoscaler для приложения. Пример ресурса, который будет масштабировать число реплик от 2 до 10 при условии, если процессор будет занят на 80 процентов и память до 200Mi, представлен на рисунке 55.

```yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: asp-net-app
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: asp-net-app
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 80
  - type: Resource
    resource:
      name: memory
      targetAverageValue: 200Mi
```

Рисунок 55 – Ресурс HPA для автомасштабирования приложения asp–net–app

Теперь как всегда с помощью команды kubectl create –f <имя файла HPA>, нужно создать ресурс HPA.

Через некоторое время, обычно не больше 5 секунд, HPA получит от сервера метрик сведения об использовании оперативной памяти и процессора. Можно удостовериться в количестве реплик приложения, курируемого HPA с помощью команды: kubectl get hpa. Пример выполнения данной команды представлен на рисунке 56.

```
kubectl get hpa
NAME          REFERENCE                TARGETS                      MINPODS   MAXPODS   REPLICAS   AGE
asp-net-app   Deployment/asp-net-app   2826240 / 200Mi, 15% / 80%   2         10        2          1m
```

Рисунок 56 – Список всех HPA в кластере

Теперь можно протестировать работу горизонтального автомасштабирования с помощью нагрузочного теста, я буду использовать небольшую утилиту, которая опубликована на github.com в репозитории rakyll/hey. Пример выполнения нагрузочного тестирования представлен на рисунке 57.

```
go get -u github.com/rakyll/hey
hey -n 10000 -q 10 -c 5 http://<Ingress_Host>/asp-net-app
```

Рисунок 57 – Выполнение нагрузочного тестирования на pod

Описание параметров, используемых в тестировании:

– –n – Количество выполняемых запросов. Значение по умолчанию–200;

– –q – Ограничение скорости, в запросах в секунду (Query Per Second) на одного worker’a;

– –c – Количество worker’ов, работающих одновременно. Общее количество запросов не может быть меньше, чем уровень параллелизма. Значение по умолчанию – 50.

Здесь было сделано 10 000 запросов по 10 запросов в секунду для каждого из 5 worker’ов.

Таким образом теперь при изменении нагрузки на приложения, для которых созданы ресурсы горизонтального автомасштабирования будут увеличиваться и уменьшаться по горизонтали автоматически. Во многих IT проектах эта возможность сделала бы падения их приложений невозможными.

11 Создание Dockerfile для контейнеризации Angular приложений

Для того, чтобы приложение можно было разворачивать на платформе Kubernetes его нужно упаковать в контейнер Docker. В случае с Angular приложением в контейнер добавляется обратный прокси Nginx, который в итоге будет возвращать файлы приложения по запросу на 80 порт.

Далее на рисунке 58 приводится пример Docker файла приложения, написанного с использованием фреймворка Angular, 8 версии.

```
FROM node:latest as node
WORKDIR /app

# копирование необходимых файлов и установка зависимостей
COPY package.json package.json
RUN npm cache clean --force
RUN npm config set registry https://registry.npmjs.org/
RUN npm cache verify
RUN rm -rf  package-lock.json
RUN rm -rf node_modules
RUN npm install

# копирование всего проекта с установленными библиотеками
COPY . .
# компиляция проекта для production
RUN npm run build-prod

FROM nginx:alpine
# открытие 80 порта, чтобы по этому порту nginx отдавал приложение
EXPOSE 80

# копирование скомпилированных файлов проекта в рабочую папку Nginx
COPY --from=node /app/dist/ /usr/share/nginx/html/angular/
# копирование конфигурационного файла для Nginx
COPY ./config/nginx.conf /etc/nginx/conf.d/default.conf
```

Рисунок 58 – Dockerfile для проекта на Angular

Для того, чтобы собрать образ из Dockerfile, нужно выполнить следующую команду:

docker build –t angular–client:dev .

А чтобы запустить контейнер с только что созданным образом, нужно выполнить следующую команду:

docker run –d —name angular–client –p 4200:4200 angular–client:dev .

12 Создание Dockerfile для контейнеризации Asp.net приложений

Также, как и приложение, написанное на Angular, приложение backend’а тоже нужно упаковать в docker контейнер.

Далее на рисунке 59 приводится пример Docker файла приложения, написанного с использованием фреймворка Asp.net.

```
# escape=`
FROM microsoft/dotnet-framework:4.7.2-sdk-windowsservercore-ltsc2019 AS builder

WORKDIR C:\src\AspNetApp
COPY src\AspNetApp\packages.config .
RUN nuget restore packages.config -PackagesDirectory ..\packages

COPY src C:\src
RUN msbuild AspNetApp.csproj /p:OutputPath=c:\out /p:Configuration=Release


FROM mcr.microsoft.com/dotnet/framework/aspnet:4.7.2-windowsservercore-ltsc2019
SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop'; $ProgressPreference = 'SilentlyContinue';"]

ENV BING_MAPS_KEY bing_maps_key
WORKDIR C:\aspnetapp

RUN Remove-Website -Name 'Default Web Site'; `
    New-Website -Name 'aspnetapp' `
                -Port 80 -PhysicalPath 'c:\aspnetapp' `
                -ApplicationPool '.NET v4.5'

COPY --from=builder C:\out\_PublishedWebsites\AspNetApp C:\aspnetapp
```

Рисунок 59 – Dockerfile для проекта на Asp.net

Для того, чтобы собрать образ из Dockerfile, нужно выполнить следующую команду:

docker build –t asp–net–app:dev .

А чтобы запустить контейнер с только что созданным образом, нужно выполнить следующую команду:

docker run –d —name asp–net–app –p 80:80 asp–net–app:dev .

13 Настройка NGINX Ingress контроллера для Kubernetes

Для того, чтобы пользовательский трафик маршрутизировался в нужные сервисы, запущенные в кластере, нужно установить и настроить Ingress.

Ingress входит в состав Kubernetes и по большому счету представляет собой набор правил, которые позволяют направлять входящие соединения к сервисам кластера. Кроме того, некоторые контроллеры Ingress поддерживают следующие возможности:

– алгоритмы соединения;

– лимиты на запросы;

– перенаправление и переопределение URL–адресов;

– балансирование нагрузки на уровне TCP/UDP;
– разрыв SSL–соединений;
– контроль доступа и авторизацию [16].
Схема работы Ingress Controller’а представлена на рисунке 60.

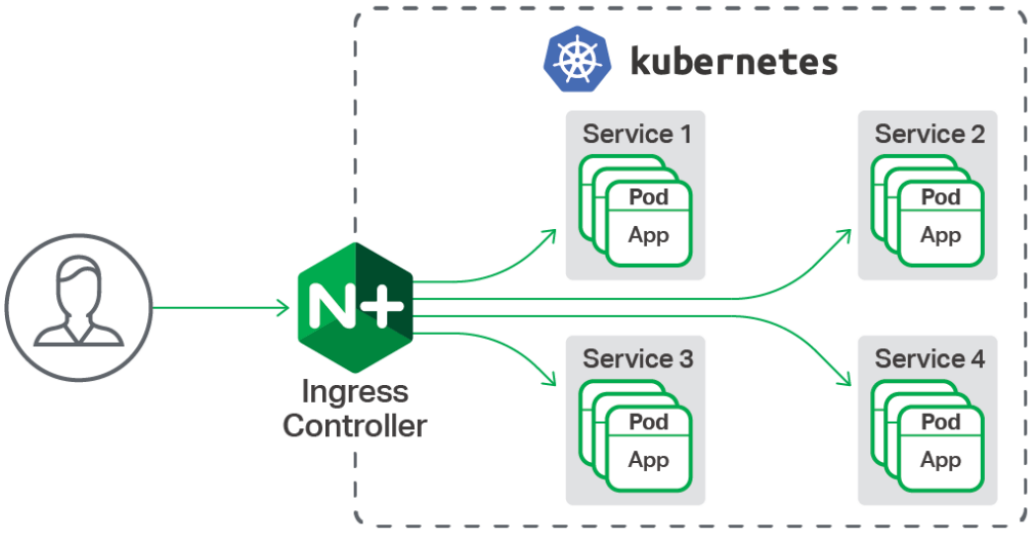

Рисунок 60 – Схема работы Ingress Controller’а

Для работы Ingress также требуется Ingress Controller.

Ingress Controller – под, который запускает Controller Ingress и nginx (в своем примере я использую Nginx, Kubernetes также поддерживает другие контроллеры). Nginx имеет файл конфигурации, в котором описывается способ балансировки и маршрутизации трафика. Ingress автоматически генерирует файл конфигурации Nginx [17].

Доступно использование 2 nginx ingress controller:

– Nginx ingress controller, разработанный сообществом Kubernetes;

– Nginx ingress controller, разработанный Nginx Inc.

В данной работе будет использоваться вариант от сообщества Kubernetes.

Как сказал Марко Лукша в книге «Kubernetes в действии»: «Одна из важных причин заключается в том, что для каждой службы LoadBalancer требуется собственный балансировщик нагрузки с собственным общедоступным IP–адресом, в то время как для Ingress’а требуется только один, даже когда предоставляется доступ к десяткам служб. Когда клиент отправляет HTTP–запрос к Ingress’у, хост и путь в запросе определяют, к какой службе этот запрос должен перенаправляться» [11].

Пример Ingress представлен на рисунке 61.

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-ingress
  namespace: stage
spec:
  rules:
    - host: mybackend1.example.com
      http:
        paths:
          - backend:
              serviceName: mybackend1-service
              servicePort: 80
```

Рисунок 61 – Пример ресурса Ingress

В данном случае все вызовы mybackend1.example.com будут маршрутизироваться в службу с именем mybackend1–service в пространстве имен stage.

Также рассмотрим еще один вариант настройки Nginx Ingress контроллера, который представлен на рисунке 62.

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-resource
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  rules:
  - host: dev.apps.idet.kz
    http:
      paths:
      - path: /lims-angular
        backend:
          serviceName: angular-service
          servicePort: 80
      - path: /lims-angular/api
        backend:
          serviceName: backend-for-angular-service
          servicePort: 80
```

Рисунок 62 – Ресурс Ingress

Так как здесь нет четко определенного пространства имен, Ingress работает как маршрутизатор для всего кластера. Чтобы не было жестких привязок к адресу, на котором доступно API, в приложениях, написанных с помощью фреймворка Angular очень удобно обращаться всегда просто по /api, именно поэтому в настройке ingress есть путь для фронтенд приложения, и такой же путь, но с дополнительной секцией /api. Такая настройка позволяет управлять бэкенд приложениями для фронтенд приложений только в одном

месте, а именно в Ingress. Как показано на рисунке 63, в данном случае в приложении Angular можно указать просто /api/ для базового url backend’а.

```
export const environment = {
  production: true,
  apiUrl: '/api/',
};
```

Рисунок 63 – Задание базового url для обращения к backend’у приложения

Реализация Ingress контроллера может быть создана на базе различных прокси серверов: nginx, envoy, haproxy, traefik, или самописного варианта.

Стоит упомянуть о схеме построение маршрутизации с помощью Ingress, когда создается один Ingress на весь кластер, но в путях мы описываем различные namespace’ы.

Например, у нас будет 3 приложения, в которых присутствует фронтенд и бэкенд. И также есть 3 namespace’а: dev, stage, production. В таком случае можно создать для каждого из них развертывается приложение с разными приставками к имени: dev–front–1, stage–front–1, prod–front–1. И в Ingress контроллере уже описываются именно эти имена для каждого хоста.

14 Проактивный мониторинг

При построении любой инфраструктуры для ИТ систем разной сложности нужно предоставить возможность отслеживания состояния системы, чтобы можно было реагировать на различные инциденты, связанные с отказом тех или иных сервисов. Хотя Kubernetes сам постоянно следит за разными метриками приложений, запущенных в кластере, чаще всего следует дополнить стандартную систему отслеживания метрик.

Существует 2 вида мониторинга систем:

– реактивный;

– проактивный.

В реализации реактивного мониторинга система мониторинга получает информацию о состоянии компонентов исследуемой системы в режиме реального времени. Это позволяет очень быстро реагировать на неполадки или некорректную работу компонентов системы. Минус данного вида в том, что реакция на инциденты возможна только после случившегося инцидента.

Более совершенным на данный момент является проактивный мониторинг. Данный вид мониторинга обеспечивает не только сбор метрик, отражающих состояния компонентов системы в реальном времени, но также позволяет предотвратить инцидент на ранних этапах его зарождения. С

помощью этого вида мониторинга можно анализировать работоспособность распределенных приложений и приложений, построенных при помощи микросервисной архитектуры.

14.1 Prometheus

На сегодняшний день существует решение, которое служит синонимом к словосочетанию проактивный мониторинг, и данное решение называется Prometheus.

Prometheus – целый набор утилит для мониторинга и настройки уведомлений, при наступлении определенных ситуаций. Это Open Source проект, который был создан в 2012 году и разрабатывался в компании Sound Cloud, а затем был передан в CNCF (Cloud Native Computing Foundation). Разработка началась из–за острой необходимости в системе, которая помогла бы проводить мониторинг приложений, написанных с помощью микросервисной архитектуры. Раньше было намного проще анализировать состояние монолита, проверяя лишь одно приложение, но теперь, когда темпы развития микросервисов только усиливаются и микросервисы показывают свои несомненные плюсы, понадобилась разработки совершенно нового инструмента.

На момент написания данной работы, Prometheus имеет 31400 звезд на github.com, а Prometheus–operator – 4400 звезд.

Проект Prometheus стал вторым за все время, ставшим «выпускником» CNCF, это произошло 9 августа 2018 года. Первым «выпускником» был Kubernetes. Также стоит сказать, что проект был разработан разработчиками, которые до работы в Sound Cloud работали в Google, и знали о проекте Google Borg Monitor.

Prometheus создан очень гибким, позволяющим проводить мониторинг совершенно разных систем: серверов, баз данных, облачных решений, запущенных в K8S, эта возможность достигается при помощи exporters.

Здесь следует отметить, что существует 2 модели мониторинга: push и pull.

При помощи обоих подходов создано немало отличных инструментов для мониторинга, перечислим основные различия:

– при push модели исследуемые сервисы сами должны отправлять значения метрик в инструмент мониторинга, и с одной стороны получается преимущество в том, что метрики для каждого сервиса могут быть особенными, и специфичными лишь для данного сервиса, но с другой стороны нет единого управляющего инструмента, который бы позволял собирать метрики централизованно;

– при pull модели, на которой построен мониторинг, реализующийся при помощи Prometheus, есть одно место в кластере, в котором происходит настройка сбора метрик, и Prometheus сам с нужной периодичностью

запрашивает данные у сервисов. Этот подход позволяет не копировать одинаковые части кода, а настроить все конфигурации лишь в одном месте, и когда нужно, например, поменять интервал – не придется еще раз развертывать все сервисы в кластере.

В Prometheus основной является модель pull, но также можно настроить мониторинг на основе модели push, с помощью компонента системы Pushgateway.

Для того, чтобы запустить мониторинг в кластере K8S разработчики специально создали Prometheus Operator. Операторы позволяют автоматизировать всю работу по созданию и конфигурированию всех ресурсов, относящихся к определенному процессу. Возможность создавать операторы появилась в 2016 году, и это было большим событием для всего сообщества, использовавшего и разрабатывающего для K8S.

В операторе есть следующие компоненты:

– prometheus – определяет кластер Prometheus;

– servicemonitor – определяет, как собирать метрики набора сервисов;

– alertmanager – определяет кластер Alertmanager'ов.

Архитектура Prometheus оператора представлена на рисунке 64.

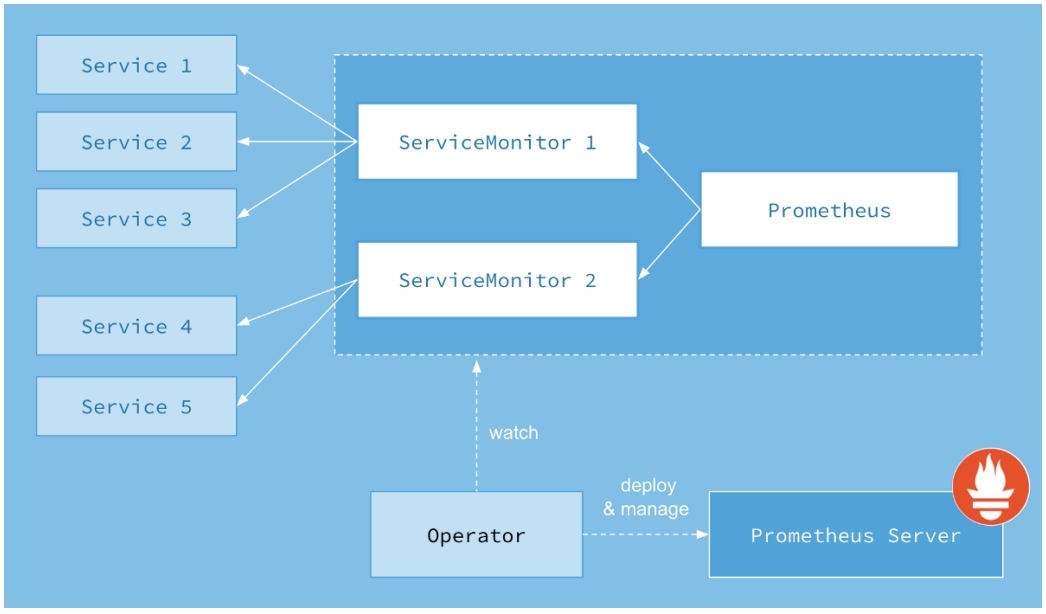

Рисунок 64 – Архитектура Prometheus Operator [18]

14.2 Архитектура Prometheus

В состав Prometheus входят следующие компоненты:

– сервер, считывающий и сохраняющий метрики в time series БД;

– клиентские библиотеки для языков программирования (Go, Java, Python, Ruby и т.д; также библиотеки для Node.js, .NET/C# были разработаны позже сообществом);

– Pushgateway – компонент, используемый для отправки метрик на основе push модели мониторинга;

– PROMDASH – dashboard для метрик;

– AlertManager – компонент для управления уведомлениями;

– командная строка для того, чтобы разработчик мог выполнять прямые запросы к данным.

Большинство из них написаны на Go, а совсем небольшая часть – на Ruby и Java. Все компоненты Prometheus взаимодействуют между собой по протоколу HTTP [19].

Между всеми компонентами архитектуры Prometheus взаимодействие происходит по HTTP протоколу. Управляющий элемент архитектуры – сервер Prometheus. Схема взаимодействия всех описанных компонентов представлена на рисунке 65.

Рисунок 65 – Архитектура Prometheus [19]

14.3 Установка Prometheus в кластере Kubernetes

Чтобы было удобнее работать с мониторингом в кластере, лучше создать пространство имен monitoring с помощью команды, показанной на рисунке 66.

```
kubectl create namespace monitoring
```

Рисунок 66 – Создание отдельного пространства имен для всех компонентов мониторинга

Далее нужно склонировать репозиторий, содержащий prometheus–operator, и перейти в директорию, как показано на рисунке 67.

```
git clone https://github.com/coreos/prometheus-operator.git
cd prometheus-operator
```

Рисунок 67 – Клонирование репозитория prometheus–operator

С помощью пакетного менеджера Helm установить оператор в только что созданное пространство имен monitoring, следующей командой, показанной на рисунке 68.

```
helm install --name prometheus-operator --set rbacEnable=true \
    --namespace=monitoring helm/prometheus-operator
```

Рисунок 68 – Команда для установки prometheus–operator

Далее нужно установить сам Prometheus, а также необходимые вспомогательные элементы, такие как Alert manager и Grafana. Команды для установки этих 3 компонентов показаны на рисунке 69.

```
helm install --name prometheus --set serviceMonitorsSelector.app=prometheus \
    --set ruleSelector.app=prometheus --namespace=monitoring helm/prometheus
helm install --name alertmanager --namespace=monitoring helm/alertmanager
helm install --name grafana --namespace=monitoring helm/grafana
```

Рисунок 69 – Команды для установки prometheus, alertmanager и grafana

Далее нужно установить сам оператор Prometheus, который автоматизирует основную настройку конфигурации и начальную установку. Он хранится в helm chart’е под названием helm/kube–prometheus. Команда для установки этого чарта показана на рисунке 70.

```
helm install --name kube-prometheus --namespace=monitoring helm/kube-prometheus
```

Рисунок 70 – Команда для установки chart’a helm/kube–prometheus

Далее для проверки установки можно выполнить команду kubectl get pods –n monitoring, и все поды должны иметь статус «Запущен» («Running»).

15 Пакетный менеджер Kubernetes – Helm

Helm изначально появился как Open Source – проект компании Deis. Перед Helm 1 задачей было дать возможность для пользователей быстро установить свои рабочие нагрузки в K8S. Helm был впервые официальный анонсирован на 1 конференции KubeCon San Francisco в 2015 году.

DevOps – автоматизация всего, что только возможно автоматизировать. Именно поэтому Helm привлек внимание сообщества, так как дает следующие возможности:

– переиспользование за счёт шаблонов;

– наличие официального репозитория chart'ов, из которых можно собирать собственные решения;

– инструменты для отладки (lint);

– использование секретов, рендер и использование других helm–плагинов;

– установка, обновление, удаление своих решений;

– версионирование, откат состояния до нужной версии, при неудавшемся развертывании.

Для того, чтобы автоматизировать развертывания нужно каждое приложение, запускаемое в кластере упаковать в так называемый chart. Это понятие сопоставимо, например, с package в npm. В таком chart'е есть все необходимые описания ресурсов K8S, которые составляют приложение в кластере, а также шаблонизируемые вставки, которые служат для переиспользования кода.

Так из чего же состоит созданный chart?

Каждый chart состоит из следующих файлов:

– Chart.yaml – .yaml файл, в котором указана информация о chart'е;

– LICENSE – текстовый файл, содержащий лицензию chart'а (если она есть);

– README.md – необязательный файл с документацией к использованию chart'а;

– requirements.yaml – .yaml файл со списком chart'ов – зависимостей (необязательный);

– values.yaml – .yaml файл, в котором хранятся значения по умолчанию для шаблонов;

– charts – папка с дочерними chart'ами (необязательная);

– templates – папка с необходимыми шаблонами манифестов ресурсов;

– templates/NOTES.txt – файл с текстом, который выводится пользователю при установке и обновлении (необязательный). Часто содержит инструкции для первоначальной настройки либо выставленные настройки.

Если не использовать Helm, то придется самому следить за всеми ресурсами, которые развертываются одним приложением. Команды могли бы выглядеть таким образом, как показано на рисунке 71.

```
kubectl create -f ./lims/web-api-deployment.yaml
kubectl create -f ./lims/web-api-service.yaml
kubectl create -f ./lims/angular-nginx-deployment.yaml
kubectl create -f ./lims/angular-nginx-service.yaml
```

Рисунок 71 – Необходимые команды для развертывания приложения без использования Helm

Также основной ценностью при работе с Helm является его способность откатывать систему к предыдущему состоянию. В Helm 3 версии этот механизм еще больше улучшился. Рассмотрим ситуацию:

– была развернута 1 версия приложения;

– разработчик изменил что–либо «руками», без использования Helm;

– нужно развернуть 2 версию приложения.

И в данной ситуации в Helm 2 было сложно предсказать, как поведет себя пакетный менеджер, и какое состояние получится после выкатки новой версии. Но в новой версии используется специальный механизм, позволяющий гарантировать приведение кластера к нужному состоянию.

Далее будет показано содержимое файлов для минимальной работоспособности Helm–чарта.

На рисунке 72 показано содержимое файла Chart.yaml.

```yaml
name: aspnet-app
version: 1.0.0
```

Рисунок 72 – Содержимое файла Chart.yaml.

На рисунке 73 показано содержимое файла values.yaml.

```yaml
environment: development

label:
  name: aspnet-app

container:
  name: aspnet-app
  pullPolicy: IfNotPresent
  image: localhost:5000/aspnet-app
  tag: 1
  port: 80
replicas: 3

service:
  port: 8888
  type: ClusterIP
```

Рисунок 73 – Содержимое файла values.yaml.

На рисунке 74 показано содержимое файла templates/deployment.yaml.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-deployment
  labels:
    app: {{ .Values.label.name }}
spec:
  replicas: {{ .Values.replicas }}
  selector:
    matchLabels:
      app: {{ .Values.label.name }}
  template:
    metadata:
      labels:
        app: {{ .Values.label.name }}
        environment: {{ .Values.environment }}
    spec:
      containers:
        - name: {{ .Values.container.name }}
          image: {{ .Values.container.image }}:{{ .Values.container.tag }}
          imagePullPolicy: {{ .Values.container.pullPolicy }}
          ports:
            - containerPort: {{ .Values.container.port }}
          env:
            - name: appenvironment
              value: {{ .Values.environment}}
```

Рисунок 74 – Содержимое файла templates/deployment.yaml.

На рисунке 75 показано содержимое файла templates/services.yaml.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}-service
  labels:
    app: {{ .Values.label.name }}
spec:
  ports:
  - port: {{ .Values.service.port}}
    protocol: TCP
    targetPort: {{ .Values.container.port }}
  selector:
    app: {{ .Values.label.name }}
  type: {{ .Values.service.type }}
```

Рисунок 75 – Содержимое файла templates/services.yaml.

ЗАКЛЮЧЕНИЕ

В данной работе были изучены 3 оркестратора служб, которые позволяют запускать контейнеризованные приложения. Лучшим из Docker Swarm, Nomad и Kubernetes оказался Kubernetes. В его составе очень много инструментов, позволяющих надежно управлять приложениями.

Kubernetes лучше всего подходит для разработки приложений, разработанных с помощью микросервисов, так как именно в этих системах есть риск, что любое место приложения может перестать работать в любой момент времени, но и при разработке с помощью монолитной архитектуры тоже требуется более совершенный контроль над приложениями. Проверки readiness и liveness помогают отследить работоспособность приложения, и вовремя пересоздать модули на другом узле. Даже если вдруг будет такая ситуация, что целый сервер выйдет из строя – контроллеры платформы через небольшой промежуток времени обнаружат данный факт, и перенесут все модули, которые работали на данном узле, на другие работоспособные узлы и администраторам инфраструктуры можно решать проблему уже после такого происшествия.

В ходе работы был создан кластер K8S при помощи утилиты kubeadm, и установлена утилита calico для настройки сети в кластере и установлена и настроена утилита Bare Metal load balancer, позволяющая настроить балансировщик нагрузки на сервере (облачные поставщики уже имеют настроенный балансировщик при создании проекта).

Был создан объект маршрутизации при помощи Nginx Kubernetes Ingress, который позволяет создавать понятную, простую, но в то же время надежную маршрутизацию на основе доменов. Маршрутизатор позволяет задать точки входа в кластер для пользователей приложений.

Были изучены инструменты и лучшие практики и паттерны для правильной разработки подов (модулей), которые будут разворачиваться в кластере K8S. Данные подходы к разработке позволяют создавать современные, быстрые и отказоустойчивые системы.

Также в ходе исследования платформы K8S выяснилось, что консольные приложения, которые запускаются в данный момент с помощью планировщика заданий в Windows, очень удобно запускать при помощи Job и CronJob в новой инфраструктуре, что позволит также контролировать успешное или неуспешное выполнение задач. Эти консольные приложения используются для выполнения различных синхронизаций в базах и хранилищах данных, которые получаются посредством интеграций с другими компаниями и их запуск должен быть всегда точно по расписанию.

Развертывания позволяют поставлять новые версии для пользователей совершенно без простоя, только с одним условием, что в приложениях существует обратная совместимость в API. Также в случае неуспешного обновления можно вернуться на любую версию – ревизию – и платформа

вернет состояние кластера к предыдущему. Эта возможность позволяет вывести разработку на новый уровень

Исследуя новые способы развертывания, стало известно, что существует специализированное программное обеспечение под названием Service Mesh (проект Istio), позволяющие совершать канареечные развертывания приложений, следить за авторизацией и аутентификацией вызовов между разными сервисами одной и той же системы, а также добавлять трассировку запросов для отслеживания пути прохождения запросов при разборе случившихся инцидентов. В скором времени данный инструмент также будет детально изучен, и добавлен в инфраструктурную платформу.

Канареечные развертывания похожи на сине-зеленые, но лучше управляются и используют прогрессивный поэтапный подход. К этому типу относятся несколько различных стратегий, включая «скрытые» запуски и A/B-тестирование [20].

Именно качественно настроенный deploy поможет в будущем настроить автоматизированный процесс обновления приложения при помощи инструментов continuous integration и continuous delivery. Для данной системы планируется использовать Azure DevOps Server, который стал следующим поколением Team Foundation Server. В нем есть все необходимое для настройки процессов CI/CD. Уже сейчас ведутся эксперименты для нахождения оптимальной схемы работы.

Таким образом, при проведении данной работы были изучены материалы и литература для понимания различных инструментов при построении отказоустойчивой и масштабируемой системы на базе Kubernetes.

Kubernetes – новая операционная система для разработчиков в облачной архитектуре, которая уже имеет свою, довольно зрелую, экосистему. В нем есть различные фоновые процессы, можно запускать свои приложения, предварительно упаковав их в контейнеры, у него есть хранилища (Persistent Volume и Volume), где приложения хранят свои данные во время работы и после нее, есть свой пакетный менеджер для удобной установки разных приложений, разработанной другими программистами. За такой полноценной платформой будущее, и чтобы в этом будущем не оставаться в прошлом, нужно изучать и применять в работе такие системы уже сейчас.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 DevOps. URL: https://ru.wikipedia.org/wiki/DevOps.

2 Kubernetes vs Docker – What Is the Difference? URL: https://www.nakivo.com/blog/docker-vs-kubernetes/.

3 Настройка docker swarm кластера в Unix/Linux. URL: https://linux-notes.org/nastrojka-docker-swarm-klastera-v-unix-linux/.

4 Официальная документация проекта Nomad. URL: https://www.nomadproject.io/.

5 Официальная документация проекта Consul. URL: https://www.consul.io/.

6 Пользователи и авторизация RBAC в Kubernetes. URL: https://habr.com/ru/company/flant/blog/470503/.

7 Building large clusters. URL: https://kubernetes.io/docs/setup/best–practices/cluster–large/.

8 Основы Kubernetes. URL: https://habr.com/ru/post/258443/.

9 Hideto Saito, Hui-Chuan Chloe Lee, Cheng-Yang Wu. DevOps with Kubernetes. – Packt Publishing, 2017. – 381 с.

10 Так что же такое pod в Kubernetes? URL: https://habr.com/ru/company/flant/blog/427819/.

11 Марко Лукша, Kubernetes в действии / пер. с англ. А. В. Логунов. – М.: ДМК Пресс, 2019. – 672с.

12 Билджин Ибрам, Роланд Хасс, Паттерны Kubernetes: Шаблоны разработки собственных облачных приложений. – СПб.: Питер, 2020.

13 Kubernetes deployment strategies. URL: https://blog.container-solutions.com/kubernetes-deployment-strategies.

14 Горизонтальное автомасштабирование подов Kubernetes и Prometheus для высокой доступности и работоспособности инфраструктуры, URL: https://habr.com/ru/company/otus/blog/457742/.

15 Horizontal Pod Autoscaler. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

16 Gigi Sayfan, Mastering Kubernetes. – Packt Publishing, 2017. – 400 с.

17 Конфигурация Ingress в Kubernetes с использованием Azure Container Service, URL: https://medium.com/southbridge/configure-ingress-on-kubernetes-using-azure-container-service-f72b5cee41f3.

18 The Prometheus Operator: Managed Prometheus setups for Kubernetes. URL: https://coreos.com/blog/the-prometheus-operator.html

19 Мониторинг сервисов с Prometheus. URL: https://habr.com/ru/company/selectel/blog/275803/

20. Стратегии деплоя в Kubernetes, URL: https://habr.com/ru/company/flant/blog/471620/