УТВЕРЖДЕНО

RU.86432418.00001-01 90 03-1 - ЛУ

Программное обеспечение «Deckhouse Platform Certified Security Edition»

Руководство администратора

RU.86432418.00001-01 90 03-1

Листов 171

2025

Содержание

Список используемых обозначений и сокращений

ОС	Операционная система
ПО	Программное обеспечение
ПО «Deckhouse Platform»	Программное обеспечение «Deckhouse Platform Certified Security Edition»

1 Действия по приемке поставленного средства

Приемка поставленного ПО «Deckhouse Platform» осуществляется в соответствии с указаниями, содержащимися в документе «Программное обеспечение «Deckhouse Platform». Технические условия. RU.86432418.00001-01 ТУ 03-1».

Перед началом эксплуатации, для обнаружения любого расхождения между оригиналом ПО «Deckhouse Platform» и версией, полученной Заказчиком, проводится процедура приемки.

Приемка поставленной ПО «Deckhouse Platform» включает в себя следующие процедуры:

- проверка упаковки;
- проверка маркировки;
- проверка комплектности;
- проверка целостности.

Проверка упаковки, маркировки и комплектности проводится методом визуального осмотра. При осмотре упаковки проверяется:

- наименование и адрес отправителя (изготовителя);
- отсутствие механических повреждений упаковки.

Если на упаковке имеются значительные повреждения, которые могут свидетельствовать о нарушении целостности упаковки и ее содержимого, а также о неправомерном вскрытии конверта третьими лицами, или отправителем не является Акционерное общество «Флант» (адрес: Акционерное общество «Флант», 115088, г. Москва, ул. Угрешская, д. 12, стр. 4, офис 47А) такой комплект поставки признается бракованным и отсылается обратно отправителю. Если по результатам проверки целостности упаковки нарушений выявлено не было – проводится проверка упаковки и маркировки.

Проверка маркировки подразумевает проверку наличия во вкладыше следующих данных:

- наименование изделия;
- децимальный номер;
- логотип предприятия-производителя;
- год выпуска;
- серийный номер;

- адрес, номер телефона, адрес электронной почты, ссылка на сайт предприятия-производителя;
- краткая информация об изделии и его основных функциональных возможностях.

На этапе приемки должна выполняться проверка заполнения и подписания ответственными разделов «Свидетельство о приемке» и «Свидетельство об упаковке и маркировке» в Формуляре, поставляемом на бумажном носителе.

Если по результатам проверки маркировки обнаружено отсутствие какого-либо элемента маркировки – такой комплект поставки признается бракованным и отсылается обратно отправителю. Если по результатам проверки маркировки нарушений выявлено не было – проводится проверка комплектности поставки.

Проверка комплектности поставки проводится методом сравнения состава полученной поставки требованиям, указанным в п. 4.1 Формуляра. Если по результатам проверки комплектности поставки расхождения с Формуляром отсутствуют – далее производится проверка целостности.

В рамках проверки целостности должны быть проведены:

- проверка целостности USB флэш-накопителя;
- проверка целостности дистрибутива ПО «Deckhouse Platform»;
- проверка целостности установленного ПО «Deckhouse Platform».

Проверка целостности USB флэш-накопителя выполняется путем визуального определения отсутствия каких-либо механических или иных повреждений. Если по результатам проверки на USB флэш-накопителе обнаружены повреждения – такой комплект поставки признается бракованным и отсылается обратно отправителю.

Проверка целостности ПО «Deckhouse Platform» на USB флэш-накопителе информации выполняется путем снятия контрольных сумм с помощью с использованием утилиты gostsum из состава сертифицированной ОС и сравнения их с контрольными суммами, приведенными в Электронном приложении к Формуляру (каталог «Электронные приложения» – «Deckhouse Platform. Формуляр» – «distr_cs.txt). Если по результатам проверки контрольная сумма дистрибутива, записанного на USB флэш-накопитель информации, соответствует значению контрольной суммы, приведенной в Формуляре – требуется выполнить установку ПО «Deckhouse Platform» в соответствии с разделом 3 настоящего документа. Если по результатам проверки контрольная сумма дистрибутива ПО «Deckhouse Platform», записанного на USB флэш-накопитель информации, не

соответствует значению контрольной суммы, приведенной в Формуляре – такой комплект поставки признается бракованным и отсылается обратно отправителю.

Проверка целостности, установленной ПО «Deckhouse Platform» выполняется путем снятия контрольных сумм с исполняемых файлов с помощью программного обеспечения утилиты gostsum из состава сертифицированной ОС и сравнения полученных контрольных сумм с контрольными суммами, приведенными в Электронном приложении к Формуляру (каталог «Электронные приложения» – «Deckhouse Platform. Формуляр» – «bins_cs.txt). Инструкция по расчету контрольных сумм приведена в Формуляре в разделе 4.4. Если по результатам проверки контрольные суммы исполняемых файлов, установленного ПО «Deckhouse Platform», соответствуют значениям контрольных сумм, приведенных в Формуляре – полученный экземпляр ПО «Deckhouse Platform» считается соответствующим оригиналу ПО «Deckhouse Platform». Если по результатам проверки контрольные суммы исполняемых файлов, установленного ПО «Deckhouse Platform», не соответствуют значениям контрольных сумм, приведенным в Формуляре – такой комплект поставки признается бракованным и отсылается обратно отправителю.

В случае, если по всем указанным процедурам установлено полное соответствие, ПО «Deckhouse Platform» признается годным к эксплуатации.

2 Действия по безопасной установке и настройке ПО «Deckhouse Platform»

При установке и настройке ПО «Deckhouse Platform» для обеспечения безопасности необходимо выполнение следующих условий:

- инсталляция ПО «Deckhouse Platform» должна осуществляться в защищенной инфраструктуре работником соответствующей квалификации, имеющим права администратора с присвоенными ему идентификационными данными для работы в среде функционирования ПО «Deckhouse Platform»;
- действия, проводимые при инсталляции ПО «Deckhouse Platform», а также при инициализации ПО «Deckhouse Platform», подлежат логированию;
- установка и конфигурирование ПО «Deckhouse Platform» должны осуществляться в соответствии с эксплуатационной документацией;
- должно обеспечиваться предотвращение несанкционированного доступа к идентификаторам и паролям пользователей сервиса и привилегированных пользователей (администраторов информационной безопасности) ПО «Deckhouse Platform»;
- должно обеспечиваться предотвращение несанкционированного доступа к идентификаторам и паролям администраторов среды виртуализации, которые необходимы для установки и настройки ПО «Deckhouse Platform».

3 Установка и настройка

Перед установкой необходимо убедиться, что выполнены минимальные Требования к аппаратному и программному обеспечению (п. 3.2.3.8 ТУ).

Таблица 1 Минимальные требования к аппаратным и программным средствам

Изделие	Требования к аппаратной части	Требования к программной части
ПО «Deckhouse Platform»	− архитектура процессора x86-64; − не менее 4 ядер CPU; − не менее 8 GB RAM; − не менее 60 ГБ дискового пространства	− ОС РЕД ОС (7.3), ОС Astra Linux Special Edition (1.7), Альт 8 СП.

Перед установкой необходимо также проверить следующие условия:

- ОС сервера находится в списке поддерживаемых ОС и до сервера есть SSH-доступ по ключу;
- на сервере обновлены пакеты ОС до последних версий;
- ОС сервера использует ядро Linux версии 5.7 или новее;
- каждый сервер должен иметь уникальное имя (hostname) в пределах серверов кластера;
- наличие сетевого доступа от персонального компьютера с которого производится установка по порту 22322/TCP;
- на сервере не должно быть установлено пакетов container runtime, например containerd или Docker;
- наличие доступа к хранилищу образов контейнеров (container registry) для загрузки образов ПО «Deckhouse Platform».

Справочник администратора, содержащий дополнительную информацию о ПО «Deckhouse Platform», приведен в электронном приложении к настоящему документу, каталог «Электронные приложения» - «Deckhouse Platform. Руководство администратора» - «cse-admin-reference.pdf».

3.1 Файл конфигурации установки

Для установки ПО «Deckhouse Platform» нужно подготовить YAML-файл конфигурации установки.

Если в кластере будет использоваться внутренний TLS-сертификат, то необходимо также добавить его в конфигурацию установки, поместив в манифест секрета в пространстве имен d8-system.

YAML-файл конфигурации установки содержит манифесты ресурсов и обычно называется config.yml. Файл конфигурации установки может содержать, в том числе следующие манифесты ресурсов:

- InitConfiguration – начальные параметры конфигурации ПО «Deckhouse Platform». С этой конфигурацией ПО «Deckhouse Platform» запустится после установки. См. примеры для установки с использованием локального и общедоступного registry общедоступного репозитория ПО «Deckhouse Platform» в п 3..5

- ClusterConfiguration – общие параметры кластера, такие как версия control plane, сетевые параметры, параметры CRI и т.д.

  Не изменяйте параметры serviceSubnetCIDR, podSubnetNodeCIDRPrefix, podSubnetCIDR в работающем кластере. Если изменение параметров необходимо — разверните новый кластер.

- StaticClusterConfiguration – ресурс для указания списка внутренних сетей узлов кластера, который используется для связи компонентов кластера между собой. Укажите, если узлы кластера имеют более одного сетевого интерфейса. Если на узлах кластера используется только один интерфейс, ресурс StaticClusterConfiguration можно не создавать.

- ModuleConfig – вид ресурсов, содержащих параметры глобальной конфигурации и конфигурации модулей Deckhouse.

Пример рабочей конфигурации установки (измените параметры, специфичные для вашей инфраструктуры). Приведенный пример конфигурации установки настраивает ПО «Deckhouse Platform» для выполнения заявленных функций безопасности:

```
apiVersion: deckhouse.io/v1
kind: InitConfiguration
deckhouse:
  imagesRepo: registry.company.my/deckhouse/cse
  # строку с данными аутентификации в хранилище можно получить командой:
  # echo '{"auths": {"<REGISTRY_HOST>": {"auth": "'$(echo -n
'<REGISTRY_USER>:<REGISTRY_PASSWORD>' | base64 -w0)'"}}}' | base64 -w0
  registryDockerCfg: <ДАННЫЕ_АУТЕНТИФИКАЦИИ_В_ХРАНИЛИЩЕ_ОБРАЗОВ>
  registryScheme: HTTPS
  registryCA: |
    -----BEGIN CERTIFICATE-----
```

```
    ...
    -----END CERTIFICATE-----
---
apiVersion: deckhouse.io/v1
kind: ClusterConfiguration
# Адресное пространство подов кластера
podSubnetCIDR: 10.111.0.0/16
# Адресное пространство сервисов кластера
serviceSubnetCIDR: 10.222.0.0/16
# Версия control plane кластера Kubernetes
kubernetesVersion: "Automatic"
# Домен кластера (используется для маршрутизации внутри кластера).
# Не должен совпадать с доменом, указанным
# в глобальном параметре publicDomainTemplate.
clusterDomain: cluster.local
clusterType: Static
# Глобальные настройки proxy-сервера.
#proxy:
#  httpProxy: https://user:password@proxy.company.my:8443
#  httpsProxy: https://user:password@proxy.company.my:8443
#  noProxy:
#  - company.my
---
apiVersion: deckhouse.io/v1
kind: StaticClusterConfiguration
internalNetworkCIDRs:
- 10.244.0.0/16
- 10.50.0.0/16
---
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: deckhouse
spec:
  version: 1
  enabled: true
  settings:
    bundle: Default
    logLevel: Info
---
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: global
spec:
  version: 2
  enabled: true
  settings:
    modules:
      https:
        mode: CertManager
        certManager:
          # Указывается имя ClusterIssuer,
          # который будет использоваться для выпуска сертификатов
          clusterIssuerName: selfsigned
      # Укажите шаблон DNS-имен кластера
      publicDomainTemplate: "%s.kube.company.my"
    defaultClusterStorageClass: local-path
---
apiVersion: deckhouse.io/v1alpha1
```

```
kind: ModuleConfig
metadata:
  name: cert-manager
spec:
  version: 1
  enabled: true
  settings:
    disableLetsencrypt: true
---
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: cni-cilium
spec:
  version: 1
  enabled: true
  settings:
    tunnelMode: VXLAN
---
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: cilium-hubble
spec:
  enabled: true
---
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: admission-policy-engine
spec:
  enabled: true
  settings:
    denyVulnerableImages:
      enabled: true
  version: 1
---
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: control-plane-manager
spec:
  enabled: true
  settings:
    apiserver:
      auditPolicyEnabled: true
  version: 1
---
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: gost-integrity-controller
spec:
  enabled: true
  version: 1
  # fix только для версии CSE 1.67
  settings:
    tlsSkipVerify: true
---
apiVersion: deckhouse.io/v1alpha1
```

```
kind: ModuleConfig
metadata:
  name: loki
spec:
  enabled: true
  settings:
    retentionPeriodHours: 24
    storeSystemLogs: true
    storageClass: local-path
  version: 1
---
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: node-manager
spec:
  enabled: true
  settings:
    earlyOomEnabled: false
  version: 2
---
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: operator-trivy
spec:
  enabled: true
  settings:
    linkCVEtoBDU: true
    severities:
      - UNKNOWN
      - LOW
      - MEDIUM
      - HIGH
      - CRITICAL
  version: 1
---
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: prometheus
spec:
  enabled: true
  settings:
    auth:
      allowedUserGroups:
      - admins
    vpa:
      longtermMaxMemory: 1500Mi
      maxMemory: 1500Mi
      updateMode: "Off"
  version: 2
---
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: runtime-audit-engine
spec:
  enabled: true
---
```

```
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  annotations:
  name: user-authn
spec:
  enabled: true
  settings:
    controlPlaneConfigurator:
      dexCAMode: FromIngressSecret
    publishAPI:
      enabled: true
  version: 2
---
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: user-authz
spec:
  enabled: true
  settings:
    enableMultiTenancy: true
  version: 1
---
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: console
spec:
  enabled: true
---
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: chrony
spec:
  enabled: true
  settings:
    ntpServers:
      - <YOUR_NTP_SERVER.1>
      - <YOUR_NTP_SERVER.2>
      - <YOUR_NTP_SERVER.3>
  version: 1
---
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: ingress-nginx
spec:
  enabled: true
---
apiVersion: v1
data:
  audit-policy.yaml:
YXBpVmVyc2lvbjogYXVkaXQuazhzLmlvL3YxICMgVGhpcyBpcyByZXF1aXJlZC4Ka2luZDogUG9sa
WN5CiMgRG9uJ3QgZ2VuZXJhdGUgYXVkaXQgZXZlbnRzIGZvciBhbGwgcmVxdWVzdHMgaW4gUmVxdW
VzdFJlY2VpdmVkIHN0YWdlLgpvbWl0U3RhZ2VzOgogIC0gIlJlcXVlc3RSZWNlaXZlZCIKcnVsZXM
6CiAgIyBBIGNhdGNoLWFsbCBydWxlIHRvIGxvZyBhbGwgb3RoZXIgcmVxdWVzdHMgYXQgdGhlIE1l
dGFkYXRhIGxldmVsLgogIC0gbGV2ZWw6IE1ldGFkYXRhCiAgICAjIExvbmctcnVubmluZyByZXF1Z
XN0cyBsaWtlIHdhdGNoZXMgdGhhdCBmYWxsIHVuZGVyIHRoaXMgcnVsZSB3aWxsIG5vdAogICAgIy
```

```
BnZW5lcmF0ZSBhbiBhdWRpdCBldmVudCBpbiBSZXF1ZXN0UmVjZWl2ZWQuCiAgICBvbWl0U3RhZ2V
zOgogICAgICAtICJSZXF1ZXN0UmVjZWl2ZWQiCgo=
kind: Secret
metadata:
  name: audit-policy
  namespace: kube-system
type: Opaque
---
apiVersion: deckhouse.io/v1alpha1
kind: NodeGroupConfiguration
metadata:
  name: sysctl-tune-fstec
spec:
  weight: 100
  bundles:
  - "*"
  nodeGroups:
  - "*"
  content: |
    sysctl -w kernel.dmesg_restrict=1
    sysctl -w kernel.kptr_restrict=2
    sysctl -w net.core.bpf_jit_harden=2
    sysctl -w kernel.perf_event_paranoid=3
    sysctl -w kernel.kexec_load_disabled=1
    sysctl -w user.max_user_namespaces=0
    sysctl -w kernel.unprivileged_bpf_disabled=1
    sysctl -w vm.unprivileged_userfaultfd=0
    sysctl -w dev.tty.ldisc_autoload=0
    sysctl -w vm.mmap_min_addr=4096
    sysctl -w kernel.randomize_va_space=2
    sysctl -w kernel.yama.ptrace_scope=3
    sysctl -w fs.protected_symlinks=1
    sysctl -w fs.protected_hardlinks=1
    sysctl -w fs.protected_fifos=2
    sysctl -w fs.protected_regular=2
    sysctl -w fs.suid_dumpable=0
---
apiVersion: deckhouse.io/v1alpha1
kind: LocalPathProvisioner
metadata:
  name: local-path
spec:
  path: "/opt/local-path-provisioner"
  reclaimPolicy: Delete
```

3.2 Установка ПО «Deckhouse Platform»

Подготовьте рабочую станцию (хост установки) и master-узел будущего кластера.

Для установки ПО «Deckhouse Platform» требуется наличие в локальной сети хранилища образов контейнеров (container registry). В случае отсутствия registry, выполните шаги по его установке, приведенные в п. 3.8 Развертывание registry.

Скопируйте файлы поставки с поставляемого USB-флеш накопителя на компьютер, с которого есть доступ до хранилища образа контейнеров.

С помощью утилиты gostsum (утилита из состава сертифицированной ОС по алгоритму ГОСТ Р 34.11-2012 (длина хэш-кода 256 бит), https://wiki.astralinux.ru/pages/viewpage.action?pageId=3277020) получите контрольную сумму файлов d8 и файла архива образов (например, platform.tar) и сравните полученные результаты соответственно с содержимым файлов с суффиксом .gostsum (d8.gostsum для d8 и, например, platform.tar.gostsum для platform.tar из состава поставки). Полученные результаты расчета контрольных сумм и контрольные суммы в указанных файлах поставки должны совпадать.

Загрузите данные в хранилище образов контейнеров, выполнив следующую команду (измените путь к файлу архива или папке образов):

```
d8 mirror push <PATH> <REGISTRY_ULR>/<REGISTRY_PATH> \
  --registry-login=<USERNAME> --registry-password=<PASSWORD>
```

, где

- `<PATH>` - директория поставки, содержащая архивы с образами поставки ПО «Deckhouse Platform»;
- `<REGISTRY_URL>` - адрес хранилища образов контейнеров в локальной сети;
- `<REGISTRY_PATH>` - путь в хранилище образов контейнеров, в который будут загружаться образы ПО «Deckhouse Platform». В примерах ниже будет использоваться путь /deckhouse/cse;
- `<USERNAME>` - имя пользователя для авторизации в хранилище образов контейнеров;
- `<PASSWORD>` - пароль пользователя для авторизации в хранилище образов контейнеров.

Если ваш container registry допускает анонимный доступ без авторизации, не указывайте параметры `--registry-login` и `--registry-password`.

Подробнее, загрузка образов в registry рассмотрена в п. 3.8 Загрузка образов в изолированный registry.

Создайте SSH-ключ для доступа к узлам кластера:

```
ssh-keygen -t rsa -f "$HOME/.ssh/id_rsa" -C "" -N ""
cat "$HOME/.ssh/id_rsa.pub"
```

Добавьте публичную часть ключа на все узлы кластера ПО «Deckhouse Platform», в данном примере на узлах создается пользователь caps:

```
# Укажите публичную часть SSH-ключа пользователя.
export KEY='<SSH-PUBLIC-KEY>'
useradd -m -s /bin/bash caps
usermod -aG sudo caps
```

```
echo 'caps ALL=(ALL) NOPASSWD: ALL' | sudo EDITOR='tee -a' visudo
mkdir /home/caps/.ssh
echo $KEY >> /home/caps/.ssh/authorized_keys
chown -R caps:caps /home/caps
chmod 700 /home/caps/.ssh
chmod 600 /home/caps/.ssh/authorized_keys

# для Astra Linux - максимальный уровень целостности для пользователя caps
pdpl-user -i 63 caps
```

Запустите контейнер установщика:

```
docker run --pull=always -it [<MOUNT_OPTIONS>] <REGISTRY_URL>/<REGISTRY_PATH>/install:<TAG> bash
```

, где:

- <REGISTRY_URL> – адрес container registry с образами ПО «Deckhouse Platform».
- <REGISTRY_PATH> - путь в хранилище образов контейнеров, в который были загружены образы ПО «Deckhouse Platform». В примерах ниже будет использоваться путь /deckhouse/cse;
- <TAG> - тег версии ПО «Deckhouse Platform». Например, v1.67.4
- <MOUNT_OPTIONS> – параметры монтирования файлов в контейнер инсталлятора, таких как:
  - SSH-ключи доступа
  - файл конфигурации
  - файл ресурсов и т.д.

Пример запуска контейнера инсталлятора:

```
docker run -it --pull=always \
-v "$PWD/config.yaml:/config.yaml" \
-v "$PWD/dhctl-tmp:/tmp/dhctl" \
-v "$HOME/.ssh/:/tmp/.ssh/" registry.company.int/deckhouse/cse/install:v1.67.4 bash
```

Установка ПО «Deckhouse Platform» запускается в контейнере инсталлятора с помощью команды dhctl:

- Для запуска установки ПО «Deckhouse Platform» используйте команду `dhctl bootstrap`.

Для получения справки по параметрам выполните **dhctl bootstrap -h**.

Пример запуска установки ПО «Deckhouse Platform»:

```
dhctl bootstrap --ssh-user=<SSH_USER> --ssh-host=<MASTER_HOST>
--ssh-agent-private-keys=/tmp/.ssh/id_rsa --config=/config.yml --ask-become-pas
s
```

, где:

- `/config.yml` — файл конфигурации установки;
- `<SSH_USER>` — пользователь на сервере для подключения по SSH;
- `<MASTER_HOST>` — адрес сервера для подключения по SSH;
- `--ssh-agent-private-keys` – файл приватного SSH-ключа, для подключения по SSH.
- `--ask-become-pass` - параметр указывает, что пользователю на узлах кластера ПО «Deckhouse Platform» необходимо вводить пароль при использовании sudo, если пользователю на узлах кластера sudo доступен без пароля, то данный параметр указывать не нужно.

Дождитесь успешного завершения установки

Пример успешного завершения установки:

```
...
└ ⛵ ~ Bootstrap: Create Resources (290.81 seconds)

┌ ⛵ ~ Bootstrap: Run post bootstrap actions
│ ┌ Create deckhouse release for version v1.67.4
│ │ 🎉 Succeeded!
│ └ Create deckhouse release for version v1.67.4 (0.01 seconds)
└ ⛵ ~ Bootstrap: Run post bootstrap actions (0.01 seconds)

┌ ⛵ ~ Bootstrap: Clear cache
│ ❗ ~ Next run of "dhctl bootstrap" will create a new Kubernetes cluster.
└ ⛵ ~ Bootstrap: Clear cache (0.00 seconds)

🎉 Deckhouse cluster was created successfully!
```

Проверьте состояние очереди, выполнив команду на master-узле:

```
d8 platform queue list
```

В очереди не должно быть задач на выполнение.

Пример вывода:

```
$ d8 platform queue list
Summary:
- 'main' queue: empty.
- 91 other queues (0 active, 91 empty): 0 tasks.
- no tasks to handle.
```

После успешной установки первого master-узла необходимо донастроить кластер.

Если планируется развернуть кластер для тестирования, состоящий из одного master-узла, то необходимо выполнить команду, которая снимает ограничение на размещение системных компонент Deckhouse на master-узлах.

Для этого на master-узле выполните следующую команду:

```
d8 k patch nodegroup master --type json -p '[{"op": "remove", "path": "/spec/nodeTemplate/taints"}]'
```

Если планируется развернуть полноценный отказоустойчивый кластер, состоящий из трех master-узлов, двух system-узлов, двух frontend-узлов и одного worker-узла, то используйте приведенную ниже команду. Если какие-либо группы узлов вам не нужны, то удалите описания соответствующих NodeGroup и StaticInstance.

Выполните на master-узле следующую команду, для подготовки добавления трех master-узлов, двух system-узлов, двух frontend-узлов и одного worker-узла (измените необходимые параметры):

```
d8 k apply -f - << EOF
# Данные подключения по SSH для доступа к хостам узлов кластера
apiVersion: deckhouse.io/v1alpha1
kind: SSHCredentials
metadata:
  name: ssh-credentials
spec:
  # измените имя пользователя, если у вас он другой
  user: caps
  privateSSHKey: '<SSH_PRIVATE_KEY>'
---
apiVersion: deckhouse.io/v1
kind: NodeGroup
metadata:
  name: master
spec:
  nodeTemplate:
    labels:
      node-role.kubernetes.io/control-plane: ""
      node-role.kubernetes.io/master: ""
    taints:
      - effect: NoSchedule
        key: node-role.kubernetes.io/control-plane
  nodeType: Static
  staticInstances:
    count: 2
    labelSelector:
      matchLabels:
        role: master
---
apiVersion: deckhouse.io/v1
kind: NodeGroup
metadata:
```

```
  name: system
spec:
  nodeTemplate:
    labels:
      node-role.deckhouse.io/system: ""
    taints:
      - effect: NoExecute
        key: dedicated.deckhouse.io
        value: system
  nodeType: Static
  staticInstances:
    count: 2
    labelSelector:
      matchLabels:
        role: system
---
apiVersion: deckhouse.io/v1
kind: NodeGroup
metadata:
  name: frontend
spec:
  nodeTemplate:
    labels:
      node-role.deckhouse.io/frontend: ""
    taints:
      - effect: NoExecute
        key: dedicated.deckhouse.io
        value: frontend
  nodeType: Static
  staticInstances:
    count: 2
    labelSelector:
      matchLabels:
        role: frontend
---
apiVersion: deckhouse.io/v1
kind: NodeGroup
metadata:
  name: worker
spec:
  nodeType: Static
  staticInstances:
    count: 1
    labelSelector:
      matchLabels:
        role: worker
---
apiVersion: deckhouse.io/v1alpha1
kind: StaticInstance
metadata:
  name: master-2
  labels:
    role: master
spec:
  address: '<MASTER_2_NODE_IP>'
  credentialsRef:
    kind: SSHCredentials
    name: ssh-credentials
---
apiVersion: deckhouse.io/v1alpha1
```

```
kind: StaticInstance
metadata:
  name: master-3
  labels:
    role: master
spec:
  address: '<MASTER_3_NODE_IP>'
  credentialsRef:
    kind: SSHCredentials
    name: ssh-credentials
---
apiVersion: deckhouse.io/v1alpha1
kind: StaticInstance
metadata:
  name: system-1
  labels:
    role: system
spec:
  address: '<SYSTEM_1_NODE_IP>'
  credentialsRef:
    kind: SSHCredentials
    name: ssh-credentials
---
apiVersion: deckhouse.io/v1alpha1
kind: StaticInstance
metadata:
  name: system-2
  labels:
    role: system
spec:
  address: '<SYSTEM_2_NODE_IP>'
  credentialsRef:
    kind: SSHCredentials
    name: ssh-credentials
---
apiVersion: deckhouse.io/v1alpha1
kind: StaticInstance
metadata:
  name: frontend-1
  labels:
    role: frontend
spec:
  address: '<FRONTEND_1_NODE_IP>'
  credentialsRef:
    kind: SSHCredentials
    name: ssh-credentials
---
apiVersion: deckhouse.io/v1alpha1
kind: StaticInstance
metadata:
  name: frontend-2
  labels:
    role: frontend
spec:
  address: '<FRONTEND_2_NODE_IP>'
  credentialsRef:
    kind: SSHCredentials
    name: ssh-credentials
---
apiVersion: deckhouse.io/v1alpha1
```

```
kind: StaticInstance
metadata:
  name: worker-1
  labels:
    role: worker
spec:
  address: '<WORKER_NODE_IP>'
  credentialsRef:
    kind: SSHCredentials
    name: ssh-credentials
EOF
```

, где:

- <SSH_PRIVATE_KEY> — приватная часть ключа пользователя `caps`, сгенерированная на хосте установки, в формате Base64. Для получения ключа в нужном формате выполните следующую команду: `base64 -w0 "$HOME/.ssh/id_rsa"`.
- <MASTER_2_NODE_IP> — IP-адрес узла master-2
- <MASTER_3_NODE_IP> — IP-адрес узла master-3
- <SYSTEM_1_NODE_IP> — IP-адрес узла system-1
- <SYSTEM_2_NODE_IP> — IP-адрес узла system-2
- <FRONTEND_1_NODE_IP> — IP-адрес узла frontend-1
- <FRONTEND_2_NODE_IP> — IP-адрес узла frontend-2
- <WORKER_NODE_IP> — IP-адрес узла worker

Дополнительные общие настройки кластера.

Создайте правило появления предупреждений о наличии необычного поведения в кластере на основе журнала аудита. Данная настройка включает срабатывание на все типы событий, в том числе на предупреждения (`warning`) и уведомления (`notice`).

```
d8 k apply -f - << EOF
apiVersion: deckhouse.io/v1
kind: CustomPrometheusRules
metadata:
  name: falco-critical-alerts
spec:
  groups:
  - name: falco-critical-alerts
    rules:
    - alert: FalcoCriticalAlertsAreFiring
      for: 1m
      annotations:
        description: |
          There is a suspicious activity on a node {{ $labels.node }}.
          Check you events journal for more details.
        summary: Falco detects a critical security incident
      expr: |
```

```
        sum by (node)
(rate(falco_events{priority=~"error|critical|warning|notice"}[5m]) > 0)
EOF
```

Добавьте локального пользователя и предоставьте ему права администратора системы:

```
d8 k apply -f - << EOF
apiVersion: deckhouse.io/v1
kind: User
metadata:
  name: admin
spec:
  email: admin@deckhouse.ru
  # Сгенерируйте пароль для пользователя и подставьте его в команду echo
ниже, чтобы получить hash и base64 от него, для указания в секции password:
  # echo "" | htpasswd -BinC 10 "" | cut -d: -f2 | base64 -w0
  password: 'base64-от-хеша-вашего-пароля'
---
apiVersion: deckhouse.io/v1alpha1
kind: Group
metadata:
  name: admins
spec:
  name: admins
  members:
    - kind: User
      name: admin
---
apiVersion: deckhouse.io/v1
kind: ClusterAuthorizationRule
metadata:
  name: admin
spec:
  subjects:
    - kind: Group
      name: admins
  accessLevel: SuperAdmin
  portForwarding: true
  namespaceSelector:
    matchAny: true
EOF
```

Настройте `IngressNginxController` для входящего трафика.

Если у вас кластер только из одного узла, то используйте следующую конфигурацию:

```
d8 k apply -f - << EOF
apiVersion: deckhouse.io/v1
kind: IngressNginxController
metadata:
  name: main
spec:
  ingressClass: nginx
  inlet: HostWithFailover
  nodeSelector:
    node-role.kubernetes.io/master: ""
  tolerations:
```

```
  - effect: NoSchedule
    operator: Exists
EOF
```

Если у вас отказоустойчивый кластер с выделенными frontend-узлами для приема входящего трафика, то используйте следующую конфигурацию:

```
d8 k apply -f - << EOF
apiVersion: deckhouse.io/v1
kind: IngressNginxController
metadata:
  name: main
spec:
  ingressClass: nginx
  inlet: HostWithFailover
  nodeSelector:
    node-role.deckhouse.io/frontend: ""
  tolerations:
    - effect: NoExecute
      key: dedicated.deckhouse.io
      value: frontend
EOF
```

Дополнительная настройка балансировки входящего трафика с помощью виртуальных IP-адресов.

Для отказоустойчивой инсталляции, когда необходимо использовать виртуальный IP-адрес или несколько IP-адресов для всего кластера ПО «Deckhouse Platform», чтобы балансировать входящий трафик между узлами кластера, где располагается `IngressNginxController`, необходимо включить и настроить модуль metallb.

При наличии двух frontend-узлов и равномерного распределения входящего трафика между ними, необходимо:

- выделить на кластер ПО «Deckhouse Platform» два виртуальных IP-адреса из подсети узлов кластера
- указать выделенные IP-адреса как A-записи на DNS-сервере у соответствующего доменного имени кластера (желательно использовать wildcard-запись, например *.kube.company.my)
- указать выделенные IP-адреса в настройках модуля metallb.

DNS-сервер будет разрешать запросы к доменному имени в IP-адреса поочередно при каждом DNS-запросе. Модуль metallb распределит два выделенных IP-адреса по двум frontend-узлам, по одному на каждый узел. В этом случае трафик равномерно распределится между узлами. В случае отказа одного из frontend-узлов, его IP-адрес «переедет» на другой работающий frontend-узел.

Команды, для настройки балансировки при отказоустойчивой установке:

```
# Включите модуль metallb:
d8 k apply -f - << EOF
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: metallb
spec:
  enabled: true
  version: 2
EOF

# Создайте ресурс MetalLoadBalancerClass:
d8 k apply -f - << EOF
apiVersion: network.deckhouse.io/v1alpha1
kind: MetalLoadBalancerClass
metadata:
  name: ingress
spec:
  addressPool:
    - 192.168.2.100-192.168.2.101
  isDefault: false
  nodeSelector:
    node-role.deckhouse.io/frontend: ""
  type: L2
EOF

# Пересоздайте ресурс IngressNginxController:
d8 k delete IngressNginxController main
d8 k apply -f - << EOF
apiVersion: deckhouse.io/v1
kind: IngressNginxController
metadata:
  name: main
spec:
  ingressClass: nginx
  inlet: LoadBalancer
  loadBalancer:
    loadBalancerClass: ingress
    annotations:
      # Количество адресов, которые будут выделены из пула, описанного в
_MetalLoadBalancerClass_.
      network.deckhouse.io/l2-load-balancer-external-ips-count: "2"
EOF

#Платформа создаст сервис с типом LoadBalancer, которому будет присвоено
заданное количество адресов, например:
d8 k -n d8-ingress-nginx get svc
NAME                    TYPE           CLUSTER-IP      EXTERNAL-IP
PORT(S)                      AGE
main-load-balancer      LoadBalancer   10.222.130.11
192.168.2.100,192.168.2.101   80:30689/TCP,443:30668/TCP   11s
```

Проверьте состояние очереди, выполнив команду:

```
d8 platform queue list
```

В очереди не должно быть задач на выполнение. Пример вывода:

```
$ d8 platform queue list
Summary:
- 'main' queue: empty.
- 91 other queues (0 active, 91 empty): 0 tasks.
- no tasks to handle.
```

Дождитесь запуска подов ПО «Deckhouse Platform». Например, можно использовать следующую команду (вывод должен быть пуст):

```
d8 k get po -A | grep -v Running
```

Убедитесь, что у вас есть доступ к веб-интерфейсам ПО «Deckhouse Platform» через браузер. Например, можно проверить доступность следующих адресов:

- https://grafana.<PUBLIC_DOMAIN_NAME>
- https://kubeconfig.<PUBLIC_DOMAIN_NAME>

Проверьте, что на главной странице системы мониторинга (Grafana) корректно отображаются все данные.

3.3 Откат установки и удаление ПО «Deckhouse Platform»

При установке в случае прерывания установки или возникновения проблем во время установки, могут остаться созданные ресурсы. Для удаления созданных используйте команду: dhctl bootstrap-phase abort. Обратите внимание, что файл конфигурации (передаваемый через параметр --config) должен быть тот же, с которым производилась установка.

Чтобы удалить кластер развернутого ПО «Deckhouse Platform», используйте команду: dhctl destroy. В этом случае dhctl подключится к master-узлу, получит от него terraform state и корректно удалит созданные ресурсы.

3.4 Ресурс ClusterConfiguration

Ресурс ClusterConfiguration описывает общие параметры кластера.

Определяет, например, сетевые параметры, параметры CRI, версию control plane и т.д. Некоторые параметры можно изменять после развертывания кластера, во время его работы.

Чтобы изменить содержимое ресурса ClusterConfiguration в работающем кластере, выполните следующую команду (должна быть настроена конфигурация kubectl на работу с кластером, либо выполняйте команды на master-узле):

d8 k -n d8-system exec -ti deploy/deckhouse -- deckhouse-controller edit cluster-configuration

Пример:

```
apiVersion: deckhouse.io/v1
kind: ClusterConfiguration
podSubnetNodeCIDRPrefix: '24'
podSubnetCIDR: 10.244.0.0/16
serviceSubnetCIDR: 192.168.0.0/16
kubernetesVersion: "Automatic"
clusterDomain: "cluster.local"
clusterType: Static
#proxy:
#  httpProxy: https://user:password@proxy.company.my:8443
#  httpsProxy: https://user:password@proxy.company.my:8443
#  noProxy:
#  - company.my
```

- apiVersion *строка*

  Обязательный параметр

  Используемая версия API Deckhouse.

  Допустимые значения: deckhouse.io/v1, deckhouse.io/v1alpha1

- clusterDomain *строка*

  Обязательный параметр

  Домен кластера (используется для маршрутизации внутри кластера).

  По умолчанию: "cluster.local"

- clusterType *строка*

  Обязательный параметр

  Тип инфраструктуры кластера. Всегда - Static

- defaultCRI *строка*

Тип container runtime, используемый на узлах кластера (в NodeGroup’ах) по умолчанию.

Если используется значение NotManaged, то ПО «Deckhouse Platform» не будет управлять (устанавливать и настраивать) container runtime. В этом случае образы, используемые в NodeGroup’ах, должны содержать уже установленный container runtime.

По умолчанию: "Containerd"

Допустимые значения: Containerd, NotManaged

- kind *строка*

  Обязательный параметр

  Допустимые значения: ClusterConfiguration

- kubernetesVersion *строка*

  Обязательный параметр

Версия control plane кластера Kubernetes.

Допустимые значения: 1.27, 1.29, Automatic

- podSubnetCIDR *строка*

Обязательный параметр

Адресное пространство подов кластера.

- podSubnetNodeCIDRPrefix *строка*

Префикс сети подов на узле.

Внимание! Не меняйте параметр в уже развернутом кластере.

По умолчанию: "24"

- proxy *объект*

Глобальная настройка proxy-сервера.

Внимание! *Для того, чтобы избежать использования прокси в запросах между компонентами кластера, важно заполнить параметр noProxy списком подсетей, которые используются на узлах.*

  - proxy.httpProxy *строка*

    URL proxy-сервера для HTTP-запросов.

    При необходимости укажите имя пользователя, пароль и порт proxy-сервера.

    Шаблон: ^https?://[0-9a-zA-Z\.\-:@]+$

    Примеры:

```
httpProxy: http://proxy.company.my
httpProxy: https://user:password@proxy.company.my:8443
```

  - proxy.httpsProxy *строка*

    URL proxy-сервера для HTTPS-запросов.

    При необходимости укажите имя пользователя, пароль и порт proxy-сервера.

    Шаблон: ^https?://[0-9a-zA-Z\.\-:@]+$

    Примеры:

    httpsProxy: http://proxy.company.my

    httpsProxy: https://user:password@proxy.company.my:8443

  - proxy.noProxy *массив строк*

    Список IP и доменных имен, для которых проксирование не применяется.

    Для настройки wildcard-доменов используйте написание вида “.example.com”.

    Шаблон: ^[a-z0-9\-\./]+$

- serviceSubnetCIDR *строка*

Обязательный параметр

Адресное пространство для service’ов кластера.

3.5 Ресурс InitConfiguration

Version: deckhouse.io/v1

Конфигурация ПО «Deckhouse Platform», с которой он запустится после установки.

Пример конфигурации при использовании локального репозитория:

```
apiVersion: deckhouse.io/v1
kind: InitConfiguration
deckhouse:
  imagesRepo: registry.company.my/deckhouse/cse
  registryDockerCfg:
eyJhdXRocyI6IHsgIm5leHVzLmNvbXBhbnkubXkiOiB7InVzZXJuYW1lIjoibmV4dXMtdXNlciI
sInBhc3N3b3JkIjoibmV4dXMtcEBzc3cwcmQiLCJhdXRoIjoiYm1WNGRYTXRkWE5sY2pwdVpYaD
FjeTF3UUhOemR6QnlaQW89In19fQo=
  registryScheme: HTTPS
  registryCA: |
    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
```

Пример конфигурации при использовании общедоступного репозитория ПО «Deckhouse Platform» (замените <DKP_CSE_LICENSE_TOKEN> на лицензионный ключ от CSE редакции):

```
apiVersion: deckhouse.io/v1
kind: InitConfiguration
deckhouse:
  imagesRepo: registry-cse.deckhouse.ru/deckhouse/cse
  # строку с данными аутентификации в хранилище можно получить командой:
  # echo '{"auths": {"registry-cse.deckhouse.ru": {"auth": "'$(echo -n
'license-token:<DKP_CSE_LICENSE_TOKEN>' | base64 -w0)'"}}}' | base64 -w0
  registryDockerCfg:
eyJhdXRocyI6IHsicmVnaXN0cnktY3NlLmRlY2tob3VzZS5ydSI6IHsiYXV0aCI6ICJiR2xqWlc
1elpTMTBiMnRsYmpvOFJFdFFYME5UUlY5TVNVTkZUbE5GWDFSUFMwVk9QZz09In19fQo=
```

- apiVersion *строка*

  Обязательный параметр

  Используемая версия API Deckhouse.

  Допустимые значения: deckhouse.io/v1, deckhouse.io/v1alpha1

- deckhouse *объект*

  Обязательный параметр

  Параметры, необходимые для установки ПО «Deckhouse Platform».

- devBranch

  Укажите версию ПО «Deckhouse Platform», загруженную в хранилище образов контейнеров.

- deckhouse.imagesRepo строка

  Адрес container registry с образами ПО «Deckhouse Platform».

  Обязательное поле.

- deckhouse.registryCA *строка*

  Корневой сертификат, которым можно проверить сертификат container registry при работе по HTTPS (если registry использует самоподписанные SSL-сертификаты).

- deckhouse.registryDockerCfg *строка*

  Строка с правами доступа к container registry, зашифрованная в Base64.

  Обязательное поле.

- deckhouse.registryScheme *строка*

  Протокол доступа к container registry (HTTP или HTTPS).

  По умолчанию: "HTTPS"

  Допустимые значения: HTTP, HTTPS

- kind *строка*

  Обязательный параметр

  Допустимые значения: InitConfiguration

3.6 Ресурс StaticClusterConfiguration

Version: deckhouse.io/v1

Дополнительные параметры кластера.

Пример:

```
apiVersion: deckhouse.io/v1
kind: StaticClusterConfiguration
internalNetworkCIDRs:
- 10.244.0.0/16
- 10.50.0.0/16
```

- apiVersion *строка*

  Обязательный параметр.

  Используемая версия API Deckhouse.

  Допустимые значения: deckhouse.io/v1, deckhouse.io/v1alpha1

- internalNetworkCIDRs *массив строк*

  Список внутренних сетей узлов кластера.

Внутренние сети используются для связи компонентов Kubernetes (kube-apiserver, kubelet и т.д.) между собой.

Если каждый узел в кластере имеет только один сетевой интерфейс, то параметр можно не указывать и ресурс StaticClusterConfiguration можно не создавать.

- Элемент массива *строка*

  Шаблон:

  ^(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])(\/(3[0-2]|[1-2][0-9]|[0-9]))$

  Пример:

  192.168.42.0/24

- Kind *строка*

  Обязательный параметр

  Допустимые значения: StaticClusterConfiguration

3.7 Создание образов

Создание образов происходит с помощью утилиты docker из среды функционирования ПО «Deckhouse Platform».

Описания создания образов приведено ниже:

- Astra Linux Special Edition
  `https://wiki.astralinux.ru/pages/viewpage.action?pageId=158601444&ysclid=lwsxujpv4r224070482`
- ОС РЕД ОС
  `https://redos.red-soft.ru/base/arm/arm-other/docker-install/?ysclid=lwsy07s1ux387721779`
- Альт 8 СП
  `https://www.altlinux.org/Docker`

При включенном и настроенном operator-trivy п. 4.1, развертывание контейнеров из уязвимых образов будет запрещено.

3.8 Развертывание registry

Для развертывания registry (хранилища образов контейнеров) выполните следующие шаги:

Установите на сервер пакеты ОС docker-registry apache2-utils:

```
apt install docker-registry apache2-utils
```

Сгенерируйте пользователя для доступа в registry (укажите имя пользователя и пароль, в примере используется пользователь deckhouse):

```
htpasswd -bnB deckhouse <PASSWORD> > /etc/docker/registry/htpasswd
```

Отредактируйте конфигурацию registry в файле /etc/docker/registry/config.yml и измените параметр auth.htpasswd.path с /etc/docker/registry на /etc/docker/registry/htpasswd.

Поместите ключ и TLS-сертификат в следующие файлы:

/etc/docker/registry/private.key # Ключ

/etc/docker/registry/public.crt # Сертификат

При отсутствии TLS-сертификатов, можно сгенерировать самоподписанные сертификаты (см п.3.15.)

Обновите конфигурацию локального registry:

```
cat <<EOF > /etc/docker/registry/config.yml
version: 0.1
log:
  fields:
    service: registry
storage:
  cache:
    blobdescriptor: inmemory
  filesystem:
    rootdirectory: /var/lib/docker-registry
  delete:
    enabled: true
http:
  addr: :443
  tls:
    certificate: /etc/docker/registry/public.crt
    key: /etc/docker/registry/private.key
  secret: deckhouse
  headers:
    X-Content-Type-Options: [nosniff]
auth:
  htpasswd:
    realm: basic-realm
    path: /etc/docker/registry/htpasswd
health:
  storagedriver:
    enabled: true
    interval: 10s
    threshold: 3
EOF
```

Добавьте права на чтение приватного ключа сертификата пользователю docker-registry.

```
setfacl -m u:docker-registry:r /etc/docker/registry/private.key
```

Добавьте права на исполняемый файл docker-registry, чтобы он смог использовать системный порт 443

```
setcap 'cap_net_bind_service=+ep' /usr/bin/docker-registry
```

Перезапустите сервис docker-registry:

```
systemctl restart docker-registry
```

Проверьте работоспособность локального репозитория, например, с помощью следующей команды (укажите пароль пользователя deckhouse, который задавали на предыдущих шагах, и адрес registry):

```
curl -u deckhouse:<PASSWORD> -v https://<REGISTRY_HOST>/v2/
```

Используйте адрес registry при загрузке образов перед установкой ПО «Deckhouse Platform» (п. 3.2). Пример:

```
d8 mirror push /root/d8-bundle https://10.128.0.37:5000/deckhouse
--registry-login=deckhouse --registry-password=deckhouse
```

Если registry работает по протоколу HTTP, то добавьте параметр --insecure.

В registry ПО «Deckhouse Platform» загружается с поставляемого USB-флеш накопителя, входящего в комплект поставки, согласно п.3.3 Технических условий RU.86432418.00001-01 ТУ 03-1.

3.9 Настройка ПО «Deckhouse Platform»

ПО «Deckhouse Platform» состоит из оператора Deckhouse и модулей. Модуль – это набор из Helm-чарта, хуков, правил сборки компонентов модуля (компонентов Deckhouse) и других файлов.

ПО «Deckhouse Platform» настраивается с помощью:

- **Глобальных настроек.** Глобальные настройки хранятся в custom resource ModuleConfig/global. Глобальные настройки можно рассматривать как специальный модуль global, который нельзя отключить.
- **Настроек модулей.** Настройки каждого модуля хранятся в custom resource ModuleConfig, имя которого совпадает с именем модуля (в kebab-case).
- **Custom resource'ов.** Некоторые модули настраиваются с помощью дополнительных custom resource'ов.

Пример набора custom resource'ов конфигурации Deckhouse:

```
# Глобальные настройки.
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: global
spec:
  version: 1
  settings:
    modules:
      publicDomainTemplate: "%s.kube.company.my"
---
# Настройки модуля monitoring-ping.
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: monitoring-ping
spec:
  version: 1
  settings:
    externalTargets:
    - host: 8.8.8.8
---
# Отключить модуль dashboard.
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: dashboard
spec:
  enabled: false
```

Посмотреть список custom resource’ов ModuleConfig, состояние модуля (включен/выключен) и его статус можно с помощью команды d8 k get moduleconfigs.

Список и состояние модулей можно также получить с помощью команды d8 k get modules.

Чтобы изменить глобальную конфигурацию Deckhouse или конфигурацию модуля, нужно создать или отредактировать соответствующий ресурс ModuleConfig.

Например, чтобы отредактировать конфигурацию модуля istio выполните следующую команду:

```
d8 k edit moduleconfig/istio
```

После завершения редактирования изменения применяются автоматически.

3.10 Настройка модуля

Модуль настраивается с помощью custom resource ModuleConfig, имя которого совпадает с именем модуля (в kebab-case). Custom resource ModuleConfig имеет следующие поля:

- metadata.name – название модуля ПО «Deckhouse Platform» в kebab-case (например prometheus, node-manager).

- spec.version – версия схемы настроек модуля (целое число, больше нуля). Обязательное поле, если spec.settings не пустое. Номер актуальной версии можно увидеть в документации модуля в разделе *“Настройки”*.
    - ПО «Deckhouse Platform» поддерживает обратную совместимость версий схемы настроек модуля. Если используется схема настроек устаревшей версии, при редактировании или просмотре custom resource’а будет выведено предупреждение о необходимости обновить схему настроек модуля.
- spec.settings – настройки модуля. Необязательное поле, если используется поле spec.enabled.
- spec.enabled – необязательное поле для явного включения или отключения модуля. Если не задано, модуль может быть включен по умолчанию в одном из наборов модулей.

ПО «Deckhouse Platform» не изменяет custom resource’ы ModuleConfig. Это позволяет применять подход Infrastructure as Code (IaC) при хранении конфигурации. Другими словами, можно воспользоваться всеми преимуществами системы контроля версий для хранения настроек ПО «Deckhouse Platform», использовать Helm, kubectl и другие привычные инструменты. В данном руководстве большая часть команд подразумевает использование консольной утилиты d8, которая идет в составе поставки ПО «Deckhouse Platform», а также доступна для скачивания по адресу https://deckhouse.ru/products/kubernetes-platform/documentation/v1/deckhouse-cli/. Утилита d8 позволяет выполнять все необходимые для управления кластером операции.

Пример custom resource для настройки модуля Kube DNS:

```
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: kube-dns
spec:
  version: 1
  settings:
    stubZones:
    - upstreamNameservers:
      - 192.168.121.55
      - 10.2.7.80
      zone: directory.company.my
    upstreamNameservers:
    - 10.2.100.55
    - 10.2.200.55
```

Некоторые модули настраиваются с помощью дополнительных custom resource’ов. Воспользуйтесь поиском (вверху страницы) или выберите модуль в меню слева, чтобы просмотреть документацию по его настройкам и используемым custom resource’ам.

3.11 Включение и отключение модуля

Некоторые модули могут быть включены по умолчанию в зависимости от используемого набора модулей.

Для явного включения или отключения модуля необходимо установить `true` или `false` в поле `.spec.enabled` в соответствующем custom resource `ModuleConfig`. Если для модуля нет такого custom resource `ModuleConfig`, его нужно создать.

Пример явного выключения модуля `user-authn` (модуль будет выключен независимо от используемого набора модулей):

```
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: user-authn
spec:
  enabled: false
```

Проверить состояние модуля можно с помощью команды `d8 k get moduleconfig <ИМЯ_МОДУЛЯ>`.

3.12 Подключение провайдера аутентификации

В ПО «Deckhouse Platform» реализован ролевой метод управления доступом с помощью внешнего провайдера аутентификации, реализующего взаимодействие по протоколу LDAP или OIDC. Подключение внешнего провайдера аутентификации выполняется при помощи ресурса DexProvider (см. спецификацию в п. 3.12.1 или Справочник администратора, приведенный в электронном приложении к настоящему документу, каталог «Электронные приложения» - «Deckhouse Platform. Руководство администратора» - «cse-admin-reference.pdf»).

Перед подключением провайдера аутентификации его необходимо настроить с учетом ролевой модели доступа, используемой в ПО «Deckhouse Platform». В каталоге, доступ к которому обеспечивает провайдер аутентификации, пользователям, которые должны иметь доступ в ПО «Deckhouse Platform», должны быть присвоены группы, в соответствии с необходимым уровнем прав в рамках ролевой модели доступа ПО «Deckhouse Platform» (Технические условия, Приложение 1. Список ролей ПО «Deckhouse Platform Certified Security Edition»).

Пример ресурса DexProvider для подключения провайдера аутентификации LDAP:

```
kind: DexProvider
metadata:
  name: ldap
spec:
  displayName: LDAP
  type: LDAP
  ldap:
    bindDN: cn=admin,dc=novalocal
    bindPW: passw0rd
    groupSearch:
      baseDN: ou=groups,dc=novalocal
      filter: (objectClass=groupOfNames)
      nameAttr: cn
      userMatchers:
      - groupAttr: member
        userAttr: DN
    host: 192.168.10.10:389
    insecureNoSSL: true
    insecureSkipVerify: true
    startTLS: false
    userSearch:
      baseDN: ou=users,dc=novalocal
      emailAttr: mail
      filter: (objectClass=person)
      idAttr: DN
      nameAttr: cn
      username: cn
    usernamePrompt: Email Address
```

3.12.1 Ресурс DexProvider

- spec.displayName *строка*

  Обязательный параметр.

  Имя провайдера, которое будет отображено на странице выбора провайдера для аутентификации.

  Если настроен всего один провайдер, страница выбора провайдера показываться не будет.

- spec.ldap *объект*

  Параметры провайдера LDAP.

  - spec.ldap.bindDN *строка*

    Путь до сервис-аккаунта приложения в LDAP.

    Пример:

bindDN: uid=serviceaccount,cn=users,dc=example,dc=com

- spec.ldap.bindPW *строка*

  Пароль для сервис-аккаунта приложения в LDAP.

  Пример:
  bindPW: password

- spec.ldap.groupSearch *объект*

  Настройки фильтра для поиска групп для указанного пользователя.

  - spec.ldap.groupSearch.baseDN *строка*

    Обязательный параметр.
    Откуда будет начат поиск групп

    Пример:
    baseDN: cn=users,dc=example,dc=com

  - spec.ldap.groupSearch.filter *строка*

    Фильтр для директории с группами.

    Пример:
    filter: "(objectClass=person)"

  - spec.ldap.groupSearch.nameAttr *строка*

    Обязательный параметр.
    Имя атрибута, в котором хранится уникальное имя группы.

    Пример:
    nameAttr: name

  - spec.ldap.groupSearch.userMatchers *массив объектов*

    Обязательный параметр.
    Список сопоставлений атрибута имени юзера с именем группы.

    - spec.ldap.groupSearch.userMatchers.groupAttr *строка*

      Обязательный параметр.

Имя атрибута, в котором хранятся имена пользователей, состоящих в группе.

Пример:
groupAttr: member

- spec.ldap.groupSearch.userMatchers.userAttr *строка*

  Обязательный параметр.
  Имя атрибута, в котором хранится имя пользователя.

  Пример:
  userAttr: uid

- spec.ldap.host *строка*

  Обязательный параметр.

  Адрес и порт (опционально) LDAP-сервера.

  Пример:
  host: ldap.example.com:636

- spec.ldap.insecureNoSSL *булевый*

  Подключаться к каталогу LDAP не по защищенному порту.

  По умолчанию: false

- spec.ldap.insecureSkipVerify *булевый*

  Не производить проверку подлинности провайдера с помощью TLS.
  Небезопасно, не рекомендуется использовать в production-окружениях.

  По умолчанию: false

- spec.ldap.rootCAData *строка*

  Цепочка CA в формате PEM, используемая для валидации TLS.

  Пример:
  rootCAData: |
  -----BEGIN CERTIFICATE-----
  MIIFaDC...

-----END CERTIFICATE-----

- spec.ldap.startTLS *булевый*

  Использовать STARTTLS для шифрования.

  По умолчанию: false

- spec.ldap.userSearch *объект*

  Обязательный параметр.
  Настройки фильтров пользователей, которые помогают сначала отфильтровать директории, в которых будет производиться поиск пользователей, а затем найти пользователя по полям (его имени, адресу электронной почты или отображаемому имени).

  - spec.ldap.userSearch.baseDN *строка*

    Обязательный параметр.
    Откуда будет начат поиск пользователей.

    Пример:
    baseDN: cn=users,dc=example,dc=com

  - spec.ldap.userSearch.emailAttr *строка*

    Обязательный параметр.
    Имя атрибута, из которого будет получен email пользователя.

    Пример:
    emailAttr: mail

  - spec.ldap.userSearch.filter *строка*

    Позволяет добавить фильтр для директории с пользователями.

    Пример:
    filter: "(objectClass=person)"

  - spec.ldap.userSearch.idAttr *строка*

    Обязательный параметр.
    Имя атрибута, из которого будет получен ID пользователя.

Пример:
idAttr: uid

- spec.ldap.userSearch.nameAttr *строка*

  Атрибут отображаемого имени пользователя.

  Пример:
  nameAttr: name

- spec.ldap.userSearch.username *строка*

  Обязательный параметр.
  Имя атрибута, из которого будет получен username пользователя.

  Пример:
  username: uid

- spec.ldap.usernamePrompt *строка*

  Строка, которая будет отображаться возле поля для имени пользователя в форме ввода логина и пароля.

  По умолчанию: "LDAP username"

  Пример:
  usernamePrompt: SSO Username

- spec.oidc *объект*

  Параметры провайдера OIDC (можно указывать, только если type: OIDC).

  - spec.oidc.basicAuthUnsupported *булевый*

    Использовать POST-запросы для общения с провайдером вместо добавления токена в Basic Authorization header.
    В большинстве случаев Dex сам определяет, какой запрос ему нужно сделать, но иногда включение этого параметра может помочь.

    По умолчанию: false

  - spec.oidc.claimMapping *объект*

Некоторые провайдеры возвращают нестандартные claim’ы (например, mail). Claim mappings помогают Dex преобразовать их в стандартные claim’ы OIDC.
Dex может преобразовать нестандартный claim в стандартный, только если id_token, полученный от OIDC-провайдера, не содержит аналогичный стандартный claim.

- spec.oidc.claimMapping.email *строка*

  Claim, который будет использован для получения email пользователя.

  По умолчанию: "email"

- spec.oidc.claimMapping.groups *строка*

  Claim, который будет использован для получения групп пользователя.

  По умолчанию: "groups"

- spec.oidc.claimMapping.preferred_username *строка*

  Claim, который будет использован для получения предпочтительного имени пользователя.

  По умолчанию: "preferred_username"

o spec.oidc.clientID *строка*

Обязательный параметр.
ID приложения, созданного в OIDC-провайдере.

o spec.oidc.clientSecret *строка*

Обязательный параметр.
Пароль приложения, созданного в OIDC-провайдере.

o spec.oidc.getUserInfo *булевый*

Запрашивать дополнительные данные об успешно подключенном пользователе.

По умолчанию: false

- spec.oidc.insecureSkipEmailVerified *булевый*

  Игнорировать информацию о статусе подтверждения email пользователя. Как именно подтверждается email, решает сам провайдер. В ответе от провайдера приходит лишь информация — подтвержден email или нет.

  По умолчанию: false

- spec.oidc.insecureSkipVerify *булевый*

  Не производить проверку подлинности провайдера с помощью TLS. Небезопасно, не рекомендуется использовать в production-окружениях.

  По умолчанию: false

- spec.oidc.issuer *строка*

  Обязательный параметр.
  Адрес OIDC-провайдера.

  Пример:
  issuer: https://accounts.google.com

- spec.oidc.promptType *строка*

  Определяет — должен ли Issuer запрашивать подтверждение и давать подсказки при аутентификации.
  По умолчанию будет запрошено подтверждение при первой аутентификации. Допустимые значения могут изменяться в зависимости от Issuer.

  По умолчанию: "consent"

- spec.oidc.rootCAData *строка*

  Цепочка CA в формате PEM, используемая для валидации TLS.

  Пример:
  rootCAData: |
  -----BEGIN CERTIFICATE-----
  MIIFaDC...

-----END CERTIFICATE-----

- spec.oidc.scopes *массив строк*

  Список полей для включения в ответ при запросе токена.

  По умолчанию: ["openid","profile","email","groups","offline_access"]

- spec.oidc.userIDKey *строка*

  Claim, который будет использован для получения ID пользователя.

  По умолчанию: "sub"

- spec.oidc.userNameKey *строка*

  Claim, который будет использован для получения имени пользователя.

  По умолчанию: "name"

- spec.type *строка*

  Тип внешнего провайдера.

  Допустимые значения: OIDC, LDAP

3.13 Экспорт данных

ПО «Deckhouse Platform» предоставляет возможность администратору кластера управлять экспортом данных при помощи объектов `DataExport`.

Для того чтобы создать объект `DataExport`:

1. Выведите имя `VolumeSnapshotClass` (данный ресурс создаётся автоматически для каждого типа CSI и нужен для экспорта данных). Например, `sds-local-volume-snapshot-class`:

```
d8 k get volumesnapshotclass

Sds-local-volume-snapshot-class
local.csi.storage.deckhouse.io
Delete
22h
```

2. Создайте объект `VolumeSnapshot`:

```
d8 k apply -f - << EOF
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: my-snapshot
  namespace: <имя пространства имён, где находится PVC>
```

```
spec:
  volumeSnapshotClassName: <имя VolumeSnapshotClass>
  source:
    persistentVolumeClaimName: <имя PVC для экспорта данных>
EOF
```

3. Убедитесь, что `VolumeSnapshot` создан и готов к использованию (это занимает несколько минут):

```
d8 k -n <имя пространства имён> get volumesnapshot my-snapshot

NAMESPACE
<имя пространства имён>

NAME
My-snapshot

READYTOUSE
true

SOURCEPVC
test-pvc-for-snapshot

SOURCESNAPSHOTCONTENT

RESTORESIZE
2Gi

SNAPSHOTCLASS
sds-local-volume-snapshot-class

SNAPSHOTCONTENT
snapcontent-faf2ab1f-891d-4e5e-972c-334a490c99d8
```

4. Экспортируйте данные при помощи команды:

   ```
   d8 data create export-name snapshot/my-snapshot
   ```

5. После выполнения команды объект `DataExport` будет создан в том же пространстве имён, что и исходный ресурс.

3.14 Обновление

3.14.1 Обновление с версии v1.67.0 на версию v1.67.4

Процесс обновления ПО «Deckhouse Platform» с версии v1.67.0 на версию v1.67.4.

Загрузите образы ПО «Deckhouse Platform» версии v1.67.4 из поставки в хранилище образов контейнеров, выполнив следующую команду (измените путь к файлу архива или папке образов):

```
d8 mirror push <PATH> <REGISTRY_ULR>/<REGISTRY_PATH> \
  --registry-login=<USERNAME> --registry-password=<PASSWORD>
```

Проверьте состояние очереди, выполнив команду:

```
d8 k -ti -n d8-system exec svc/deckhouse-leader -c deckhouse --
deckhouse-controller queue list
```

В очереди не должно быть задач на выполнение. Пример вывода:

```
$ d8 k -ti -n d8-system exec svc/deckhouse-leader -c deckhouse --
deckhouse-controller queue list
Summary:
- 'main' queue: empty.
- 91 other queues (0 active, 91 empty): 0 tasks.
- no tasks to handle.
```

Проверьте, что в кластере нет критичных алертов (с SEVERITY меньше 4), выполнив команду:

```
d8 k get clusteralerts
```

Пример вывода:

```
$ d8 k get clusteralerts
NAME ALERT SEVERITY AGE LAST RECEIVED STATUS
1d2c2f7d69f69f4b D8DeckhouseIsNotOnReleaseChannel 9 3m46s 16s firing
4de1153b89a8f329 ModuleConfigObsoleteVersion 4 7m46s 16s firing
caf5186a4bb62385 ModuleConfigObsoleteVersion 4 7m46s 16s firing
```

Измените образ ПО «Deckhouse Platform», выполнив команду на master-узле (укажите адрес хранилища образов и путь, который использовался при загрузке образов):

```
d8 k --as=system:serviceaccount:d8-system:deckhouse set image deployment -n
d8-system deckhouse deckhouse=<REGISTRY_ULR>/<REGISTRY_PATH>:v1.67.4
```

Пример вывода:

```
$ d8 k --as=system:serviceaccount:d8-system:deckhouse set image deployment
-n d8-system deckhouse deckhouse=registry.local/deckhouse/cse:v1.67.4
deployment.apps/deckhouse image update
```

Дождитесь когда все поды ПО «Deckhouse Platform» перейдут в состояние готовности (Ready). Для проверки используйте команду:

```
d8 k get po -n d8-system -l app=deckhouse
```

Пример вывода:

```
$ d8 k get po -n d8-system -l app=deckhouse
NAME READY STATUS RESTARTS AGE
deckhouse-785cd848c8-4shrv 2/2 Running 0 2m31s
deckhouse-785cd848c8-kgx2j 2/2 Running 0 2m31s
deckhouse-785cd848c8-rd79j 2/2 Running 0 2m31s
```

Проверьте состояние очереди, выполнив команду:

```
d8 k -ti -n d8-system exec svc/deckhouse-leader -c deckhouse --
deckhouse-controller queue list
```

После смены образа, выполнение всех заданий в очереди может занять до 30 минут. В очереди не должно быть задач на выполнение.

Пример вывода:

```
$ d8 k -ti -n d8-system exec svc/deckhouse-leader -c deckhouse --
deckhouse-controller queue list
Summary:
- 'main' queue: empty.
- 91 other queues (0 active, 91 empty): 0 tasks.
- no tasks to handle.
```

Проверьте, что в кластере нет критичных алертов (с SEVERITY меньше 4), выполнив команду:

```
d8 k get clusteralerts
```

Пример вывода:

```
$ d8 k get clusteralerts
NAME ALERT SEVERITY AGE LAST RECEIVED STATUS
1d2c2f7d69f69f4b D8DeckhouseIsNotOnReleaseChannel 9 3m46s 16s firing
4de1153b89a8f329 ModuleConfigObsoleteVersion 4 7m46s 16s firing
caf5186a4bb62385 ModuleConfigObsoleteVersion 4 7m46s 16s firing
```

Если в списке алертов есть алерт D8DeckhouseIsNotOnReleaseChannel, то его можно игнорировать.

Если в списке алертов есть алерты ModuleConfigObsoleteVersion, то для каждого алерта выполните следующие действия:

- Выведите информацию об алерте, выполнив команду (укажите уникальное имя алерта из колонки NAME в списке алертов):

  ```
  d8 k describe clusteralert <NAME>
  ```

  Пример вывода:

  ```
  $ d8 k describe clusteralert 4de1153b89a8f329
  Name: 4de1153b89a8f329
  Namespace:
  Labels: app=prometheus
   heritage=deckhouse
  Annotations: <none>
  Alert:
   Description: ModuleConfig node-manager is outdated. Update ModuleConfig
  node-manager to the latest version.
   Labels:
   Name: node-manager
   Prometheus: deckhouse
   Name: ModuleConfigObsoleteVersion
  ```

```
 Severity Level: 4
 Summary: ModuleConfig node-manager is outdated.
API Version: deckhouse.io/v1alpha1
Kind: ClusterAlert
Metadata:
 Creation Timestamp: 2025-03-05T10:03:56Z
 Generation: 1
 Resource Version: 671301
 UID: 41cbd7dd-36cf-4ebd-a7c6-b3e0f02985d0
Status:
 Alert Status: firing
 Last Update Time: 2025-03-05T12:00:44Z
 Starts At: 2025-03-05T10:03:44Z
Events: <none>
```

- Алерт в примере сообщает о том, что необходимо обновить версию ModuleConfig для модуля node-manager.
- Отредактируйте версию соответствующего объекта ModulConfig, увеличив параметр `spec.versions`. Выполните команду:

```
d8 k edit mc node-manager
```

3.15 Создание самоподписанного сертификата

TLS-сертификат необходим для организации работы по протоколу HTTPS. Если нет сертификата, выпущенного доверенным центром сертификации, то можно сгенерировать самоподписанный сертификат, ориентируясь на шаги, приведенные в данном разделе. Для некоторых версий операционных систем шаги могут отличаться.

Сгенерируйте самоподписанный сертификат, указав свой домен от кластера и репозитория в переменных CLUSTER_DNS_NAME и REGISTRY_DNS_NAME.

Укажите актуальные данные в переменных окружения (они потребуются для выполнения команд далее):

```
CLUSTER_DNS_NAME="cluster-domain.test"
REGISTRY_DNS_NAME="registry.cluster-domain.test"
```

Создайте директорию для сертификатов:

```
mkdir ~/ca && cd ~/ca
```

Сгенерируйте сертификат корневого ЦС (Root CA)

```
openssl req -x509 -newkey rsa:2048 -nodes -days 3650 -keyout rootCA.key
-out rootCA.crt -subj "/CN=Root CA" -extensions v3_ca -config <(echo -e
"[req]\ndistinguished_name=req_distinguished_name\n[ v3_ca
]\nbasicConstraints=critical,CA:TRUE\nkeyUsage=critical,keyCertSign,cRLSi
gn\n[req_distinguished_name]")
```

Сгенерируйте сертификат промежуточного ЦС (Intermediate CA):

```
openssl req -newkey rsa:2048 -nodes -keyout intermediateCA.key -out
intermediateCA.csr -subj "/CN=Intermediate CA"
openssl x509 -req -in intermediateCA.csr -CA rootCA.crt -CAkey rootCA.key
-CAcreateserial -out intermediateCA.crt -days 3648 -extensions v3_ca
-extfile <(echo -e
"[v3_ca]\nbasicConstraints=critical,CA:TRUE,pathlen:0\nkeyUsage=critical,
keyCertSign,cRLSign\n")
```

Сгенерируйте wildcard-сертификат:

```
openssl req -newkey rsa:2048 -nodes -keyout wildcard.key -out
wildcard.csr -subj "/CN=*.$CLUSTER_DNS_NAME"
openssl x509 -req -in wildcard.csr -CA intermediateCA.crt -CAkey
intermediateCA.key -CAcreateserial -out wildcard.crt -days 3646
-extensions v3_req -extfile <(echo -e "[ v3_req
]\nkeyUsage=critical,digitalSignature,keyEncipherment\nextendedKeyUsage=s
erverAuth,clientAuth\nsubjectAltName=DNS:*.$CLUSTER_DNS_NAME,DNS:$CLUSTER
_DNS_NAME,DNS:$REGISTRY_DNS_NAME")
```

Проверьте корректность выпущенного сертификата:

```
openssl x509 -noout -text -in ~/ca/wildcard.crt
```

Добавьте сертификат в хранилище доверенных центров сертификации ОС:

```
apt install -y ca-certificates
cp ~/ca/rootCA.crt
/usr/local/share/ca-certificates/self_signed_rootCA.crt
update-ca-certificates
```

4 Доставка приложений

4.1 Описание функциональных характеристик

ПО «Deckhouse Platform» организовывает:

- Сборку образов;
- Дистрибуцию образов в реестр контейнеров;
- Очистку реестра контейнеров от собранных образов;
- Дистрибуцию Helm-чартов;
- Дистрибуция бандлов — Helm-чартов и связанных с ними образов как единого целого;
- Развертывание Helm-чартов и бандлов в кластеры Kubernetes.

Основные функциональные характеристики ПО «Deckhouse Platform» включают:

Название функции	**Результат**
Сборка образов контейнеров и их дистрибуция: с распределенным кешированием слоев в реестре контейнеров, безопасными параллельными сборками, автоматическим тегированием образов на основе их содержимого и ранее собранных слоев и, как следствие, высокой воспроизводимостью сборки	Быстрые, безопасные, воспроизводимые сборки образов и хранение образов в реестре контейнеров
Очистка реестра контейнеров от собранных образов: умная очистка более ненужных образов на основе Git-политик и информации об используемых образах из Kubernetes-кластеров	Эффективное использование реестра контейнеров
Дистрибуция Helm-чартов: загрузка и скачивание Helm-чартов в OCI и HTTP репозитории Helm-чартов	Helm-чарты, готовые к развертыванию в Kubernetes
Дистрибуция бандлов: загрузка, скачивание, перенос бандлов между OCI-репозиториями, с возможностью доставлять бандлы в изолированные окружения	Бандлы, готовые к развертыванию в Kubernetes
Развертывание Helm-чартов и бандлов в Kubernetes-кластеры: установка, обновление и удаление Helm-чартов/бандлов в Kubernetes-кластерах, с	Helm-чарты/бандлы, развернутые в Kubernetes

Название функции	Результат
тщательным отслеживанием состояния развертывания и гибко настраиваемым порядком развертывания ресурсов	

4.2 Конфигурация проекта

4.2.1 Основы

ПО «Deckhouse Platform» следует принципам подхода IaC (Infrastructure as Code) и стимулирует пользователя хранить конфигурацию доставки проекта вместе с кодом приложения в Git и осознанно использовать внешние зависимости. Отвечает за это механизм под названием гитерминизм.

Типовая конфигурация проекта состоит из нескольких файлов:

- `werf.yaml`;
- одного или нескольких Dockerfile-файлов;
- Helm-чарта.

`werf.yaml` – главный конфигурационный файл проекта в ПО «Deckhouse Platform». Его основное предназначение — связывание инструкций для сборки и развёртывания.

Инструкции для сборки – определяются для каждого компонента приложения. Представлены в формате Dockerfile-файлов.

Инструкции развёртывания – определяются для всего приложения (и всех окружений развёртывания) и должны быть представлены в виде Helm-чарта.

Пример типовой конфигурации проекта:

```
# werf.yaml
project: app
configVersion: 1
---
image: backend
context: backend
dockerfile: Dockerfile
---
image: frontend
context: frontend
dockerfile: Dockerfile
```

```
$ tree -a
.
├── .helm
│   ├── templates
│   │   ├── NOTES.txt
│   │   ├── _helpers.tpl
│   │   ├── deployment.yaml
```

```
│   │   ├── hpa.yaml
│   │   ├── ingress.yaml
│   │   ├── service.yaml
│   │   └── serviceaccount.yaml
│   └── values.yaml
├── backend
│   ├── Dockerfile
│   └── ...
├── frontend
│   ├── Dockerfile
│   └── ...
└── werf.yaml
```

4.2.2 Гитерминизм

4.2.2.1 О гитерминизме

Для обеспечения согласованности и гарантии воспроизводимости ПО «Deckhouse Platform» вводит механизм гитерминизма. Его название происходит от совмещения слов `git` и `determinism`, что можно понимать как режим «детерминированный Git». Конфигурацию и сборочный контекст ПО «Deckhouse Platform» читает из текущего коммита репозитория проекта, а также по умолчанию не позволяет использовать внешние зависимости.

Любое отступление пользователя от гитерминизма должно фиксироваться в специальном файле `werf-giterminism.yaml`, чтобы процесс управления конфигурацией был осмысленным, а использование прозрачным для всех участников доставки.

4.2.2.2 Исключение неиспользуемых файлов

ПО «Deckhouse Platform» не позволяет работать с незакоммиченными и неотслеживаемыми файлами в Git. Если файлы не требуются, то их следует явно исключать с помощью файлов `.gitignore` и `.helmignore`.

4.2.2.3 Использование незакоммиченных и неотслеживаемых файлов при отладке и разработке

При отладке и разработке изменение файлов проекта может доставлять неудобства за счёт необходимости создания промежуточных коммитов. Мы работаем над режимом разработки, чтобы упростить этот процесс и в то же время оставить всю логику работы неизменной.

В текущих версиях режим разработки (активируется опцией `--dev`) работает с состоянием worktree Git-репозитория проекта, с отслеживаемыми (tracked) и

неотслеживаемыми (untracked) файлами. ПО «Deckhouse Platform» игнорирует изменения с учётом правил, описанных в `.gitignore`, а также правил, заданных пользователем опцией `--dev-ignore=<glob>` (может использоваться несколько раз).

4.2.2.4 Выборочное разрешение недетерминированной функциональности

Использование функции Go-шаблонизатора `env` при шаблонизации werf.yaml усложняет совместное использование и воспроизводимость конфигурации в заданиях CI и среди разработчиков, поскольку значение переменной среды влияет на окончательный дайджест собираемых образов. Значение должно быть идентичным на всех этапах CI-пайплайна и во время локальной разработки при воспроизведении.

Для активации функции `env` необходимо использовать `werf-giterminism.yaml`, но мы рекомендуем еще раз подумать о возможных последствиях.

Использование директивы `contextAddFiles` при сборке Dockerfile-образа усложняет совместное использование и воспроизводимость конфигурации в заданиях CI и среди разработчиков, поскольку данные файла влияют на окончательный дайджест собираемых образов и должны быть идентичными на всех этапах CI-пайплайна и во время локальной разработки при воспроизведении.

Для активации директивы `contextAddFiles` необходимо использовать werf-giterminism.yaml, но мы рекомендуем еще раз подумать о возможных последствиях.

Использование алиасов тегов с опцией `--use-custom-tag` при развёртывании с неизменяемыми значениями (например, `%image%-master`) делает предыдущие выкаты невоспроизводимыми и требует указания политики `imagePullPolicy: Always` для каждого образа при конфигурации контейнеров приложения в Helm-чарте.

Для активации опции `--use-custom-tag` необходимо использовать `werf-giterminism.yaml`, но мы рекомендуем еще раз подумать о возможных последствиях.

4.3 Сборка

4.3.1 Основы

Доставка приложения в Kubernetes предполагает его контейнеризацию (сборку одного или нескольких образов) для последующего развёртывания в кластере.

Для сборки пользователю необходимо описать сборочные инструкции в виде Dockerfile’a, а остальное ПО «Deckhouse Platform» возьмёт на себя:

- оркестрация одновременной/параллельной сборки образов приложения;

- кроссплатформенная и мультиплатформенная сборка образов;
- общий кеш промежуточных слоёв и образов в container registry, доступный с любых раннеров;
- оптимальная схема тегирования, основанная на содержимом образа, предотвращающая лишние пересборки и время простоя приложения при выкате;
- система обеспечения воспроизводимости и неизменности образов для коммита: однажды собранные образы для коммита более не будут пересобраны.

Парадигма сборки и публикации образов в ПО «Deckhouse Platform» отличается от парадигмы, предлагаемой сборщиком Docker, в котором есть несколько команд: `build`, `tag` и `push`. ПО «Deckhouse Platform» собирает, тегирует и публикует образы в один шаг. Данная особенность связана с тем, что ПО «Deckhouse Platform» по сути не собирает образы, а синхронизирует текущее состояние приложения (для текущего коммита) с container registry, дособирая недостающие слои образов и синхронизируя работу параллельных сборщиков.

В общем случае сборка образов с ПО «Deckhouse Platform» предполагает наличие container registry (`--repo`), поскольку образ не только собирается, но и сразу публикуется. При этом ручной вызов команды `d8 dk build` не требуется, так как сборка выполняется автоматически при запуске всех верхнеуровневых команд ПО «Deckhouse Platform», в которых требуются образы (например, `d8 dk converge`).

Однако ручной запуск команды `d8 dk build` может быть полезным во время локальной разработки. В этом случае сборку можно запускать отдельно и без участия container registry.

4.3.2 Образы и зависимости

4.3.2.1 Добавление образов

Для сборки с ПО «Deckhouse Platform» необходимо добавить описание образов в `werf.yaml` проекта. Каждый образ добавляется директивой `image` с указанием имени образа:

```
project: example
configVersion: 1
---
image: frontend
# ...
---
image: backend
# ...
```

```
---
image: database
# ...
```

Имя образа — это уникальный внутренний идентификатор образа, который позволяет ссылаться на него при конфигурации и при вызове команд ПО «Deckhouse Platform».

Далее для каждого образа в `werf.yaml` необходимо определить сборочные инструкции с помощью Dockerfile.

4.3.2.1.1 Использование Dockerfile

Конфигурация сборки Dockerfile может выглядеть следующим образом:

```
# Dockerfile
FROM node
WORKDIR /app
COPY package*.json /app/
RUN npm ci
COPY . .
CMD ["node", "server.js"]
```

```
# werf.yaml
project: example
configVersion: 1
---
image: backend
dockerfile: Dockerfile
```

4.3.2.1.2 Использование определённой Dockerfile-стадии

Также вы можете описывать несколько целевых образов из разных стадий одного и того же Dockerfile:

```
# Dockerfile

FROM node as backend
WORKDIR /app
COPY package*.json /app/
RUN npm ci
COPY . .
CMD ["node", "server.js"]

FROM python as frontend
WORKDIR /app
COPY requirements.txt /app/
RUN pip install -r requirements.txt
COPY . .
CMD ["gunicorn", "app:app", "-b", "0.0.0.0:80", "--log-file", "-"]
```

```
# werf.yaml
project: example
```

```
configVersion: 1
---
image: backend
dockerfile: Dockerfile
target: backend
---
image: frontend
dockerfile: Dockerfile
target: frontend
```

И конечно вы можете описывать образы, основанные на разных Dockerfile:

```
# werf.yaml
project: example
configVersion: 1
---
image: backend
dockerfile: dockerfiles/Dockerfile.backend
---
image: frontend
dockerfile: dockerfiles/Dockerfile.frontend
```

4.3.2.1.3 Выбор директории сборочного контекста

Чтобы указать сборочный контекст используется директива `context`. **Важно:** в этом случае путь до Dockerfile указывается относительно директории контекста:

```
project: example
configVersion: 1
---
image: docs
context: docs
dockerfile: Dockerfile
---
image: service
context: service
dockerfile: Dockerfile
```

Для образа `docs` будет использоваться Dockerfile по пути `docs/Dockerfile`, а для `service` — `service/Dockerfile`.

4.3.2.1.4 Добавление произвольных файлов в сборочный контекст

По умолчанию контекст сборки Dockerfile-образа включает только файлы из текущего коммита репозитория проекта. Файлы, не добавленные в Git, или некоммитнутые изменения не попадают в сборочный контекст. Такая логика действует в соответствии) по умолчанию.

Чтобы добавить в сборочный контекст файлы, которые не хранятся в Git, нужна директива `contextAddFiles` в `werf.yaml`, а также нужно разрешить использование директивы `contextAddFiles` в `werf-giterminism.yaml`:

```
# werf.yaml
project: example
configVersion: 1
---
image: app
context: app
contextAddFiles:
- file1
- dir1/
- dir2/file2.out
```

```
# werf-giterminism.yaml
giterminismConfigVersion: 1
config:
  dockerfile:
    allowContextAddFiles:
    - app/file1
    - app/dir1/
    - app/dir2/file2.out
```

В данной конфигурации контекст сборки будет состоять из следующих файлов:

- `app/**/*` из текущего коммита репозитория проекта;
- файлы `app/file1`, `app/dir2/file2.out` и директория `dir1`, которые находятся в директории проекта.

4.3.2.2 Взаимодействие между образами

4.3.2.2.1 Наследование и импортирование файлов

При написании одного Dockerfile в нашем распоряжении имеется механизм multi-stage. Он позволяет объявить в Dockerfile отдельный образ-стадию и использовать её в качестве базового для другого образа, либо скопировать из неё отдельные файлы.

ПО «Deckhouse Platform» позволяет реализовать это не только в рамках одного Dockerfile, но и между произвольными образами, определяемыми в `werf.yaml`, в том числе собираемыми из разных Dockerfile'ов. Всю оркестрацию и выстраивание зависимостей ПО «Deckhouse Platform» возьмёт на себя и произведёт сборку за один шаг (вызов `d8 dk build`).

Пример использования образа собранного из `base.Dockerfile` в качестве базового для образа из Dockerfile:

```
# base.Dockerfile
FROM ubuntu:22.04
RUN apt update -q && apt install -y gcc g++ build-essential make curl
python3
```

```
# Dockerfile
```

```
ARG BASE_IMAGE
FROM ${BASE_IMAGE}
WORKDIR /app
COPY . .
CMD [ "/app/server", "start" ]
```

```
# werf.yaml
project: example
configVersion: 1
---
image: base
dockerfile: base.Dockerfile
---
image: app
dockerfile: Dockerfile
dependencies:
- image: base
  imports:
  - type: ImageName
    targetBuildArg: BASE_IMAGE
```

4.3.2.2.2 Передача информации о собранном образе в другой образ

ПО «Deckhouse Platform» позволяет получить информацию о собранном образе при сборке другого образа. Например, если в сборочных инструкциях образа `app` требуются имена и digest'ы образов `auth` и `controlplane`, опубликованных в container registry, то конфигурация могла бы выглядеть так:

```
# modules/auth/Dockerfile
FROM alpine
WORKDIR /app
COPY . .
RUN ./build.sh
```

```
# modules/controlplane/Dockerfile
FROM alpine
WORKDIR /app
COPY . .
RUN ./build.sh
```

```
# Dockerfile
FROM alpine
WORKDIR /app
COPY . .

ARG AUTH_IMAGE_NAME
ARG AUTH_IMAGE_DIGEST
ARG CONTROLPLANE_IMAGE_NAME
ARG CONTROLPLANE_IMAGE_DIGEST

RUN echo AUTH_IMAGE_NAME=${AUTH_IMAGE_NAME} >>
modules_images.env
```

```
RUN echo AUTH_IMAGE_DIGEST=${AUTH_IMAGE_DIGEST} >>
modules_images.env
RUN echo CONTROLPLANE_IMAGE_NAME=${CONTROLPLANE_IMAGE_NAME} >>
modules_images.env
RUN echo CONTROLPLANE_IMAGE_DIGEST=${CONTROLPLANE_IMAGE_DIGEST} >>
modules_images.env
```

```
# werf.yaml
project: example
configVersion: 1
---
image: auth
dockerfile: Dockerfile
context: modules/auth/
---
image: controlplane
dockerfile: Dockerfile
context: modules/controlplane/
---
image: app
dockerfile: Dockerfile
dependencies:
- image: auth
  imports:
  - type: ImageName
    targetBuildArg: AUTH_IMAGE_NAME
  - type: ImageDigest
    targetBuildArg: AUTH_IMAGE_DIGEST
- image: controlplane
  imports:
  - type: ImageName
    targetBuildArg: CONTROLPLANE_IMAGE_NAME
  - type: ImageDigest
    targetBuildArg: CONTROLPLANE_IMAGE_DIGEST
```

В процессе сборки ПО «Deckhouse Platform» автоматически подставит в указанные build-arguments соответствующие имена и идентификаторы. Всю оркестрацию и выстраивание зависимостей ПО «Deckhouse Platform» возьмёт на себя и произведёт сборку за один шаг (вызов `d8 dk build`).

4.3.2.3 Мультиплатформенная и кроссплатформенная сборка

ПО «Deckhouse Platform» позволяет собирать образы как для родной архитектуры хоста, где запущен ПО «Deckhouse Platform», так и в кроссплатформенном режиме с помощью эмуляции целевой архитектуры, которая может быть отлична от архитектуры хоста. Также ПО «Deckhouse Platform» позволяет собрать образ сразу для множества целевых платформ.

4.3.2.3.1 Сборка образов под одну целевую платформу

По умолчанию в качестве целевой используется платформа хоста, где запущен ПО «Deckhouse Platform». Выбор другой целевой платформы для собираемых образов осуществляется с помощью параметра `--platform`:

```
d8 dk build --platform linux/arm64
```

— все конечные образы, указанные в `werf.yaml`, будут собраны для указанной платформы с использованием эмуляции.

Целевую платформу можно также указать директивой конфигурации `build.platform`:

```
# werf.yaml
project: example
configVersion: 1
build:
  platform:
  - linux/arm64
---
image: frontend
dockerfile: frontend/Dockerfile
---
image: backend
dockerfile: backend/Dockerfile
```

В этом случае запуск `d8 dk build` без параметров вызовет сборку образов для указанной платформы (при этом явно указанный параметр `--platform` переопределяет значение из `werf.yaml`).

4.3.2.3.2 Сборка образов под множество целевых платформ

Поддерживается и сборка образов сразу для набора архитектур. В этом случае в container registry публикуется манифест включающий в себя собранные образы под каждую из указанных платформ (во время скачивания такого образа автоматически будет выбираться образ под требуемую архитектуру).

Можно определить общий список платформ для всех образов в `werf.yaml` с помощью конфигурации:

```
# werf.yaml
project: example
configVersion: 1
build:
  platform:
  - linux/arm64
  - linux/amd64
  - linux/arm/v7
```

Можно определить список целевых платформ отдельно для каждого собираемого образа (такая настройка будет иметь приоритет над общим списком определённым в `werf.yaml`):

```
# werf.yaml
project: example
configVersion: 1
---
image: mysql
dockerfile: ./Dockerfile.mysql
platform:
- linux/amd64
---
image: backend
dockerfile: ./Dockerfile.backend
platform:
- linux/amd64
- linux/arm64
```

Общий список можно также переопределить параметром `--platform` непосредственно в момент вызова сборки:

```
d8 dk build --platform=linux/amd64,linux/i386
```

— такой параметр переопределяет список целевых платформ указанных в `werf.yaml` (как общих, так и для отдельных образов).

4.3.3 Сборочный процесс

4.3.3.1 Аутентификация в container registry

Перед работой с образами необходимо аутентифицироваться в container registry:

```
d8 dk cr login <registry url>
```

Например:

```
# Login with username and password from command line
d8 dk cr login -u username -p password registry.example.com

# Login with token from command line
d8 dk cr login -p token registry.example.com

# Login into insecure registry (over http)
d8 dk cr login --insecure-registry registry.example.com
```

В случае использования команды `d8 dk ci-env` с поддерживаемыми CI/CD-системами аутентификация во встроенные container registry выполняется в рамках команды, поэтому использование команды `d8 dk cr login` в этом случае не требуется.

4.3.3.2 Тегирование образов

Тегирование образов с ПО «Deckhouse Platform» выполняется автоматически в рамках сборочного процесса. Используется оптимальная схема тегирования, основанная на содержимом образа, которая предотвращает лишние пересборки и время простоя приложения при выкате.

4.3.3.2.1 Получение тегов

Для получения тегов образов может использоваться опция `--save-build-report` для команд `d8 dk build`, `d8 dk converge` и пр.:

```
# По умолчанию формат JSON.
d8 dk build --save-build-report --repo REPO

# Поддерживается формат envfile.
d8 dk converge --save-build-report --build-report-path
.werf-build-report.env --repo REPO

# В команде рендера финальные теги будут доступны только с параметром
--repo.
d8 dk render --save-build-report --repo REPO
```

Получить теги заранее, не вызывая сборочный процесс, на данный момент невозможно, можно получить лишь теги уже собранных ранее образов.

4.3.3.2.2 Добавление произвольных тегов

Пользователь может добавить произвольное количество дополнительных тегов с опцией `--add-custom-tag`:

```
d8 dk build --repo REPO --add-custom-tag main

# Можно добавить несколько тегов-алиасов.
d8 dk build --repo REPO --add-custom-tag main --add-custom-tag latest
--add-custom-tag prerelease
```

Шаблон тега может включать следующие параметры:

- `%image%`, `%image_slug%` или `%image_safe_slug%` для использования имени образа из `werf.yaml` (обязательно при сборке нескольких образов);
- `%image_content_based_tag%` для использования content-based тега.

```
d8 dk build --repo REPO --add-custom-tag "%image%-latest"
```

При использовании опций создаются **дополнительные теги-алиасы**, ссылающиеся на автоматические теги-хэши. Полное отключение создания автоматических тегов не предусматривается.

4.3.3.3 Послойное кэширование образов

Послойное кэширование образов является неотъемлемой частью сборочного процесса ПО «Deckhouse Platform». ПО «Deckhouse Platform» сохраняет и переиспользует сборочный кэш в container registry, а также синхронизирует работу параллельных сборщиков.

Предполагается, что репозиторий образов для проекта не будет удален или очищен сторонними средствами без негативных последствий для пользователей CI/CD, построенного на основе `d8 dk cleanup`.

4.3.3.3.1 Dockerfile

По умолчанию Dockerfile-образы кешируются одним образом в container registry.

Для включения послойного кеширования Dockerfile-инструкций в container registry необходимо использовать директиву `staged` в `werf.yaml`:

```
# werf.yaml
image: example
dockerfile: ./Dockerfile
staged: true
```

4.3.3.3.2 Параллельность и порядок сборки образов

Все образы, описанные в `werf.yaml`, собираются параллельно на одном сборочном хосте. При наличии зависимостей между образами сборка разбивается на этапы, где каждый этап содержит набор независимых образов и может собираться параллельно.

При использовании Dockerfile-стадий параллельность их сборки также определяется на основе дерева зависимостей. Также, если Dockerfile-стадия используется разными образами, объявленными в `werf.yaml`, ПО «Deckhouse Platform» обеспечит однократную сборку этой общей стадии без лишних пересборок

Параллельная сборка в ПО «Deckhouse Platform» регулируется двумя параметрами `--parallel` и `--parallel-tasks-limit`. По умолчанию параллельная сборка включена и собирается не более 5 образов одновременно.

Рассмотрим следующий пример:

```
# backend/Dockerfile
FROM node as backend
WORKDIR /app
COPY package*.json /app/
RUN npm ci
COPY . .
CMD ["node", "server.js"]
```

```
# frontend/Dockerfile

FROM ruby as application
WORKDIR /app
COPY Gemfile* /app
RUN bundle install
COPY . .
RUN bundle exec rake assets:precompile
CMD ["rails", "server", "-b", "0.0.0.0"]

FROM nginx as assets
WORKDIR /usr/share/nginx/html
COPY configs/nginx.conf /etc/nginx/conf.d/default.conf
COPY --from=application /app/public/assets .
COPY --from=application /app/vendor .
ENTRYPOINT ["nginx", "-g", "daemon off;"]
```

```
image: backend
dockerfile: Dockerfile
context: backend
---
image: frontend
dockerfile: Dockerfile
context: frontend
target: application
---
image: frontend-assets
dockerfile: Dockerfile
context: frontend
target: assets
```

Имеется 3 образа `backend`, `frontend` и `frontend-assets`. Образ `frontend-assets` зависит от `frontend`, потому что он импортирует скомпилированные ассеты из `frontend`.

Формируются следующие наборы для сборки:

```
┌ Concurrent builds plan (no more than 5 images at the same time)
│ Set #0:
│ - ⛵ image backend
│ - ⛵ image frontend
│
│ Set #1:
│ - ⛵ frontend-assets
└ Concurrent builds plan (no more than 5 images at the same time)
```

4.3.3.3.3 Использование container registry

При использовании ПО «Deckhouse Platform» container registry используется не только для хранения конечных образов, но также для сборочного кэша и служебных данных, необходимых для работы ПО «Deckhouse Platform» (например, метаданные для очистки container registry на основе истории Git). Репозиторий container registry задаётся параметром `--repo`:

```
d8 dk converge --repo registry.mycompany.org/project
```

В дополнение к основному репозиторию существует ряд дополнительных:

- `--final-repo` для сохранения конечных образов в отдельном репозитории;
- `--secondary-repo` для использования репозитория в режиме `read-only` (например, для использования container registry CI, в который нельзя пушить, но можно переиспользовать сборочный кэш);
- `--cache-repo` для поднятия репозитория со сборочным кэшом рядом со сборщиками.

Важно: Для корректной работы ПО «Deckhouse Platform» container registry должен быть надёжным (persistent), а очистка должна выполняться только с помощью специальной команды `d8 dk cleanup`.

4.3.3.3.4 Дополнительный репозиторий для конечных образов

С помощью параметра `--cache-repo` можно указать один или несколько т.н. **кеширующих** репозиториев.

```
# Дополнительный кэширующий репозиторий в локальной сети.
d8 dk build --repo registry.mycompany.org/project --cache-repo
localhost:5000/project
```

Кеширующий репозиторий может помочь сократить время загрузки сборочного кэша, но для этого скорость загрузки из него должна быть значительно выше по сравнению с основным репозиторием — как правило, это достигается за счёт поднятия container registry в локальной сети, но это необязательно.

При загрузке сборочного кэша кеширующие репозитории имеют больший приоритет, чем основной репозиторий. При использовании кеширующих репозиториев сборочный кэш продолжает сохраняться и в основном репозитории.

Очистка кеширующего репозитория может осуществляться путём его полного удаления без каких-либо рисков.

4.3.3.3.5 Синхронизация сборщиков

Для обеспечения согласованности в работе параллельных сборщиков, а также гарантии воспроизводимости образов и промежуточных слоёв, ПО «Deckhouse Platform» берёт на себя ответственность за синхронизацию сборщиков.

Сервис синхронизации — это компонент ПО «Deckhouse Platform», который предназначен для координации нескольких процессов ПО «Deckhouse Platform» и

выполняет роль менеджера блокировок. Блокировки требуются для корректной публикации новых образов в container registry и реализации алгоритма сборки.

На сервис синхронизации отправляются только обезличенные данные в виде хэш-сумм тегов, публикуемых в container registry.

В качестве сервиса синхронизации может выступать:

- HTTP-сервер синхронизации, реализованный в команде `d8 dk synchronization`.
- Ресурс ConfigMap в кластере Kubernetes. В качестве механизма используется библиотека `lockgate`, реализующая распределённые блокировки через хранение аннотаций в выбранном ресурсе.
- Локальные файловые блокировки, предоставляемые операционной системой.

4.3.3.3.5.1 HTTP-сервер

Сервер синхронизации можно запустить командой d8 dk synchronization, например для использования порта 55581 (по умолчанию):

```
d8 dk synchronization --host 0.0.0.0 --port 55581
```

— данный сервер поддерживает только работу в режиме HTTP, для использования HTTPS необходима настройка дополнительной SSL-терминации сторонними средствами (например через Ingress в Kubernetes).

Далее во всех командах ПО «Deckhouse Platform», которые используют параметр `--repo` дополнительно указывается параметр `--synchronization=http[s]://DOMAIN`, например:

```
d8 dk build --repo registry.mydomain.org/repo --synchronization https://synchronization.domain.org
d8 dk converge --repo registry.mydomain.org/repo --synchronization https://synchronization.domain.org
```

4.3.3.3.5.2 Специальный ресурс в Kubernetes

Требуется лишь предоставить рабочий кластер Kubernetes, и выбрать namespace, в котором будет хранится сервисный `ConfigMap/werf`, через аннотации которого будет происходить распределённая блокировка.

Далее во всех командах ПО «Deckhouse Platform», которые используют параметр `--repo` дополнительно указывается параметр `--synchronization=kubernetes://NAMESPACE[:CONTEXT][@(base64:CONFIG_DATA)|CONFIG_PATH]`, например:

```
# Используем стандартный ~/.kube/config или KUBECONFIG.
d8 dk build --repo registry.mydomain.org/repo --synchronization
kubernetes://mynamespace
d8 dk converge --repo registry.mydomain.org/repo --synchronization
kubernetes://mynamespace

# Явно указываем содержимое kubeconfig через base64.
d8 dk build --repo registry.mydomain.org/repo --synchronization
kubernetes://mynamespace@base64:YXBpVmVyc2lvbjogdjEKa2luZDogQ29uZmlnCnByZ
WZlcmVuY2VzOiB7fQoKY2x1c3RlcnM6Ci0gY2x1c3RlcjoKICBuYW1lOiBkZXZlbG9wbWVudA
otIGNsdXN0ZXI6CiAgbmFtZTogc2NyYXRjaAoKdXNlcnM6Ci0gbmFtZTogZGV2ZWxvcGVyCi0
gbmFtZTogZXhwZXJpbWVudGVyCgpjb250ZXh0czoKLSBjb250ZXh0OgogIG5hbWU6IGRldi1m
cm9udGVuZAotIGNvbnRleHQ6CiAgbmFtZTogZGV2LXN0b3JhZ2UKLSBjb250ZXh0OgogIG5hb
WU6IGV4cC1zY3JhdGNoCg==

# Используем контекст mycontext в конфиге /etc/kubeconfig.
d8 dk build --repo registry.mydomain.org/repo --synchronization
kubernetes://mynamespace:mycontext@/etc/kubeconfig
```

Данный способ неудобен при доставке проекта в разные кластера Kubernetes из одного Git-репозитория из-за сложности корректной настройки. В этом случае для всех команд ПО «Deckhouse Platform» требуется указывать один и тот же адрес кластера и ресурс, даже если деплой происходит в разные контура, чтобы обеспечить консистивность данных в container registry. Поэтому для такого случая рекомендуется запустить отдельный общий сервис синхронизации, чтобы исключить вероятность некорректной конфигурации.

4.3.3.3.5.3 Локальная синхронизация

Включается опцией `--synchronization=:local`. Локальный менеджер блокировок использует файловые блокировки, предоставляемые операционной системой.

```
d8 dk build --repo registry.mydomain.org/repo --synchronization :local
d8 dk converge --repo registry.mydomain.org/repo --synchronization :local
```

Данный способ подходит лишь в том случае, если в вашей CI/CD системе все запуски ПО «Deckhouse Platform» происходят с одного и того же раннера.

4.4 Развёртывание

4.4.1 Основы

При организации доставки приложения в Kubernetes необходимо определиться с тем, какой формат выбрать для управления конфигурацией развёртывания (параметризации, управления зависимостями, конфигурации под различные окружения и т.д.), а также способом применения этой конфигурации – непосредственно механизмом развёртывания.

В ПО «Deckhouse Platform» встроен Helm, и именно он используется для решения перечисленных задач. Разработка и сопровождение конфигурации реализуется с помощью

Helm-чарта, а для процесса развёртывания предлагается Helm с дополнительными возможностями:

- отслеживание состояния выкатываемых ресурсов (с возможностью изменения поведения для каждого ресурса):
  - умное ожидание готовности ресурсов;
  - мгновенное завершение проблемного развертывания без необходимости ожидания таймаута;
  - прогресс развёртывания, логи, системные события и ошибки приложения.
- использование произвольного порядка развертывания для любых ресурсов, а не только для хуков;
- ожидание создания и готовности ресурсов, не принадлежащих релизу;
- интеграция сборки и развертывания и многое другое.

ПО «Deckhouse Platform» стремится сделать работу с Helm более простой, удобной и гибкой, при этом не ломая обратную совместимость с Helm-чартами, Helm-шаблонами и Helm-релизами.

4.4.1.1 Простой пример развертывания

Для развертывания простого приложения достаточно двух файлов и команды `d8 dk converge`, запущенной в Git-репозитории приложения:

```
# .helm/templates/hello.yaml:
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello
spec:
  selector:
    matchLabels:
      app: hello
  template:
    metadata:
      labels:
        app: hello
    spec:
      containers:
      - image: nginxdemos/hello:plain-text
```

```
# werf.yaml:
configVersion: 1
project: hello
```

```
d8 dk converge --repo registry.example.org/repo --env production
```

Результат: Deployment `hello` развёрнут в Namespace'е `hello-production`.

4.4.1.2 Расширенный пример развертывания

Более сложный пример развертывания со сборкой образов и внешними Helm-чартами:

```
# Dockerfile:
FROM node
WORKDIR /app
COPY . .
RUN npm ci
CMD ["node", "server.js"]
```

```
# Dockerfile:
FROM node
WORKDIR /app
COPY . .
RUN npm ci
CMD ["node", "server.js"]
```

```
# .helm/Chart.yaml:
dependencies:
- name: postgresql
  version: "~12.1.9"
  repository: https://charts.bitnami.com/bitnami
```

```
# .helm/values.yaml:
backend:
  replicas: 1
```

```
# .helm/templates/backend.yaml:
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: {{ $.Values.backend.replicas }}
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - image: {{ $.Values.werf.image.backend }}
```

```
d8 dk converge --repo registry.example.org/repo --env production
```

Результат: собран образ `backend`, а затем Deployment `backend` и ресурсы чарта `postgresql` развёрнуты в Namespace'е `myapp-production`.

4.4.2 Чарты и зависимости

4.4.2.1 О чартах

Чарты в ПО «Deckhouse Platform» – это Helm-чарты с некоторыми дополнительными возможностями. А Helm-чарты – это распространяемые пакеты с Helm-шаблонами, values-файлами и некоторыми метаданными. Из чартов формируются конечные Kubernetes-манифесты для дальнейшего развертывания.

4.4.2.2 Создание нового чарта

При развертывании с `d8 dk converge` или публикации с `d8 dk bundle publish` используется чарт, лежащий в директории `.helm` в корне Git-репозитория. Этот чарт называется основным. Директорию основного чарта можно изменить директивой `deploy.helmChartDir` файла `werf.yaml`.

Для обычного чарта требуется создание файла `Chart.yaml` и указание в нём имени и версии чарта:

```
# Chart.yaml:
apiVersion: v2
name: mychart
version: 1.2.3
```

А вот для основного чарта это не обязательно, т. к. при отсутствии файла `Chart.yaml` или отсутствии в нём имени или версии чарта будет использована следующая конфигурация:

```
# .helm/Chart.yaml:
apiVersion: v2
name: <имя проекта ПО «Deckhouse Platform»>
version: 1.0.0
```

Если такая конфигурация основного чарта вас не устраивает, то создайте файл `.helm/Chart.yaml` самостоятельно и переопределите вышеупомянутые директивы.

В случае, если ваш чарт будет содержать только именованные шаблоны для использования в других чартах, добавьте в `Chart.yaml` директиву `type: library`:

```
# Chart.yaml:
type: library
```

Если ваш чарт совместим только с частью версий Kubernetes, то ограничьте версии кластера Kubernetes, в которые чарт можно развернуть, директивой `kubeVersion`, например:

```
# Chart.yaml:
kubeVersion: "~1.20.3"
```

При желании можно добавить следующие информационные директивы:

```
# Chart.yaml:
appVersion: "1.0"
deprecated: false
icon: https://example.org/mychart-icon.svg
description: This is My Chart
home: https://example.org
sources:
  - https://github.com/my/chart
keywords:
  - apps
annotations:
  anyAdditionalInfo: here
maintainters:
  - name: John Doe
    email: john@example.org
    url: https://john.example.org
```

Полученный чарт уже можно развернуть или опубликовать, хотя в таком виде от него мало пользы. Теперь вам понадобится, по меньшей мере, либо добавить шаблоны в директорию `templates`, либо подключить зависимые чарты.

4.4.2.3 Добавление файлов в чарт

По мере необходимости добавьте в чарт шаблоны, параметры, зависимые чарты и прочее. Содержимое основного чарта может выглядеть так:

```
.helm/
  charts/
    dependent-chart/
      # ...
  templates/
    deployment.yaml
    _helpers.tpl
    NOTES.txt
  crds/
    crd.yaml
  secret/
    some-secret-file
```

```
values.yaml
values.schema.json
secret-values.yaml
Chart.yaml
Chart.lock
README.md
LICENSE
.helmignore
```

Подробнее:

- `charts/*` — зависимые чарты, чьи Helm-шаблоны/values-файлы используются для формирования манифестов наравне с Helm-шаблонами/values-файлами родительского чарта;
- `templates/*.yaml` — Helm-шаблоны, из которых формируются Kubernetes-манифесты;
- `templates/*.tpl` — файлы с Helm-шаблонами для использования в других Helm-шаблонах. Результат шаблонизации этих файлов игнорируется;
- `templates/NOTES.txt` — ПО «Deckhouse Platform» отображает содержимое этого файла в терминале в конце каждого удачного развертывания;
- `crds/*.yaml` — Custom Resource Definitions, которые развертываются до развертывания манифестов в `templates`;
- `secret/*` — зашифрованные файлы, расшифрованное содержимое которых можно подставлять в Helm-шаблоны;
- `values.yaml` — файлы с декларативной конфигурацией для использования в Helm-шаблонах. Конфигурация в них может переопределяться переменными окружения, аргументами командной строки или другими values-файлами;
- `values.schema.json` — JSON-схема для валидации `values.yaml`;
- `secret-values.yaml` — зашифрованный файл с декларативной конфигурацией, аналогичный `values.yaml`. Его расшифрованное содержимое объединяется с values.yaml во время формирования манифестов;
- `Chart.yaml` — основная конфигурация и метаданные чарта;
- `Chart.lock` — lock-файл, защищающий от нежелательного изменения/обновления зависимых чартов;

- `README.md` — документация чарта;
- `LICENSE` — лицензия чарта;
- `.helmignore` — список файлов в директории чарта, которые не нужно включать в чарт при его публикации.

4.4.2.4 Подключение дополнительных чартов

4.4.2.4.1 Подключение зависимых локальных чартов

Подключить зависимые локальные чарты можно, положив их в директорию `charts` родительского чарта. В таком случае манифесты сформируются и для родительского, и для зависимых чартов, после чего объединятся вместе для дальнейшего развертывания. Дополнительная конфигурации не обязательна.

Пример содержимого основного чарта, имеющего локальные зависимые чарты:

```
.helm/
  charts/
    postgresql/
      templates/
        postgresql.yaml
      Chart.yaml
    redis/
      templates/
        redis.yaml
      Chart.yaml
  templates/
    backend.yaml
```

Обратите внимание, что у локальных зависимых чартов имя должно обязательно совпадать с именем их директории.

Если локальному зависимому чарту требуется дополнительная конфигурация, то в файле `Chart.yaml` родительского чарта укажите имя зависимого чарта без указания `dependencies[].repository` и добавьте интересующие директивы таким образом:

```
# .helm/Chart.yaml:
apiVersion: v2
dependencies:
- name: redis
  condition: redis.enabled
```

При необходимости подключить локальный чарт не из директории `charts`, а из другого места, используйте директиву `dependencies[].repository` так:

```
# .helm/Chart.yaml:
apiVersion: v2
dependencies:
- name: redis
  repository: file://../redis
```

4.4.2.4.2 Подключение зависимых чартов из репозитория

Подключить дополнительные чарты из OCI/HTTP-репозитория можно, указав их как зависимые в директиве `dependencies` файла `Chart.yaml` родительского чарта. В таком случае манифесты сформируются и для родительского, и для зависимых чартов, после чего объединятся вместе для дальнейшего развертывания.

Пример конфигурации чарта из репозитория, зависимого от основного чарта:

```
# .helm/Chart.yaml:
apiVersion: v2
dependencies:
- name: database
  version: "~1.2.3"
  repository: https://example.com/charts
```

После каждого добавления/обновления удалённых зависимых чартов или изменения их конфигурации требуется:

1. (Если используется приватный OCI или HTTP-репозиторий с зависимым чартом) Добавить OCI или HTTP-репозиторий вручную с `d8 dk helm repo add`, указав нужные опции для доступа к репозиторию.
2. Вызвать `d8 dk helm dependency update`, который обновит `Chart.lock`.
3. Закоммитить обновлённые `Chart.yaml` и `Chart.lock` в Git.

Также при использовании чартов из репозитория рекомендуется добавить `.helm/charts/**.tgz` в `.gitignore`.

4.4.2.4.3 Указание имени подключаемого чарта

В директиве `dependencies[].name` родительского чарта указывается оригинальное имя зависимого чарта, установленное его разработчиком, например:

```
# .helm/Chart.yaml:
apiVersion: v2
dependencies:
- name: backend
```

Если нужно подключить несколько зависимых чартов с одинаковым именем или подключить один и тот же зависимый чарт несколько раз, то используйте директиву

`dependencies[].alias` родительского чарта, чтобы поменять имена подключаемых чартов, например:

```
# .helm/Chart.yaml:
apiVersion: v2
dependencies:
- name: backend
  alias: main-backend
- name: backend
  alias: secondary-backend
```

4.4.2.4.4 Указание версии подключаемого чарта

Ограничить подходящие версии зависимого чарта, из которых будет выбрана наиболее свежая, можно директивой `dependencies[].version` родительского чарта, например:

```
# .helm/Chart.yaml:
apiVersion: v2
dependencies:
- name: backend
  version: "~1.2.3"
```

Результат: будет использована самая свежая версия 1.2.x, но как минимум 1.2.3.

4.4.2.4.5 Указание источника подключаемого чарта

Указать путь к источнику чартов, в котором можно найти указанный зависимый чарт, можно директивой `dependencies[].repository` родительского чарта, например:

```
# .helm/Chart.yaml:
apiVersion: v2
dependencies:
- name: mychart
  repository: oci://example.org/myrepo
```

Результат: будет использован чарт `mychart` из OCI-репозитория `example.org/myrepo`.

4.4.2.4.6 Включение/отключение зависимых чартов

По умолчанию все зависимые чарты включены. Для произвольного включения/отключения зависимых чартов можно использовать директиву `dependencies[].condition` родительского чарта, например:

```
# .helm/Chart.yaml:
apiVersion: v2
dependencies:
- name: backend
  condition: backend.enabled
```

Результат: зависимый чарт `backend` будет включен, только если параметр `$.Values.backend.enabled` имеет значение `true` (по умолчанию — `true`).

Также можно использовать директиву `dependencies[].tags` родительского чарта для включения/отключения целых групп зависимых чартов сразу, например:

```
# .helm/Chart.yaml:
dependencies:
- name: backend
  tags: ["app"]
- name: frontend
  tags: ["app"]
```

Результат: зависимые чарты `backend` и `frontend` будут включены, только если параметр `$.Values.tags.app` имеет значение `true` (по умолчанию — `true`).

4.4.3 Шаблоны

4.4.3.1 Шаблонизация

Механизм шаблонизации в ПО «Deckhouse Platform» ничем не отличается от Helm. Используется движок шаблонов `Go text/template`, расширенный готовым набором функций Sprig и Helm.

4.4.3.2 Файлы шаблонов

В директории templates чарта находятся файлы шаблонов.

Файлы шаблонов `templates/*.yaml` формируют конечные Kubernetes-манифесты для развертывания. Каждый из этих файлов может формировать сразу несколько манифестов Kubernetes-ресурсов. Для этого манифесты должны быть разделены строкой `---`.

Файлы шаблонов `templates/_*.tpl` содержат только именованные шаблоны для использования в других файлах. Файлы `*.tpl` не формируют Kubernetes-манифесты сами по себе.

4.4.3.3 Действия

Главный элемент шаблонизации — действие. Действие может возвращать только строки. Действие заключается в двойные фигурные скобки:

```
{{ print "hello" }}
```

Результат:

```
hello
```

4.4.3.4 Переменные

Переменные используются для хранения или указания на данные любого типа.

Объявление и присваивание переменной:

```
{{ $myvar := "hello" }}
```

Присваивание нового значения существующей переменной:

```
{{ $myvar = "helloworld" }}
```

Использование переменной:

```
{{ $myvar }}
```

Результат:

```
helloworld
```

Использование предопределенных переменных:

```
{{ $.Values.werf.env }}
```

Данные можно подставлять и без объявления переменных:

```
labels:
  app: {{ "myapp" }}
```

Результат:

```
labels:
  app: myapp
```

Также в переменные можно сохранять результат выполнения функций или конвейеров:

```
{{ $myvar := 1 | add 1 1 }}
{{ $myvar }}
```

Результат:

```
3
```

4.4.3.5 Области видимости переменных

Область видимости ограничивает видимость переменных. По умолчанию область видимости переменных ограничена файлом-шаблоном.

Область видимости может меняться при использовании некоторых блоков и функций. К примеру, блок `if` создаёт новую область видимости, а переменные, объявленные в блоке `if`, будут недоступны снаружи:

```
{{ if true }}
  {{ $myvar := "hello" }}
{{ end }}
{{ $myvar }}
```

Результат:

```
Error: ... undefined variable "$myvar"
```

Чтобы обойти это ограничение, объявите переменную за пределами блока, а значение присвойте ей внутри блока:

```
{{ $myvar := "" }}
{{ if true }}
  {{ $myvar = "hello" }}
{{ end }}
{{ $myvar }}
```

Результат:

```
hello
```

4.4.3.6 Типы данных

Доступные типы данных:

Тип данных	Пример
Логический	`{{ true }}`
Строка	`{{ "hello" }}`
Целое число	`{{ 1 }}`
Число с плавающей точкой	`{{ 1.1 }}`
Список с элементами любого типа, упорядоченный	`{{ list 1 2 3 }}`
Словарь с ключами-строками и значениями любого типа, неупорядоченный	`{{ dict "key1" 1 "key2" 2 }}`
Специальные объекты	`{{ $.Files }}`
Нуль	`{{ nil }}`

4.4.3.7 Функции

В ПО «Deckhouse Platform» встроена обширная библиотека функций для использования в шаблонах. Основная их часть — функции Helm.

Функции можно использовать только в действиях. Функции могут иметь аргументы и могут возвращать данные любого типа. Например, приведенная ниже функция сложения принимает три аргумента-числа и возвращает число:

```
{{ add 3 2 1 }}
```

Результат:

```
6
```

Обратите внимание, что результат выполнения действия всегда конвертируется в строку независимо от возвращаемого функцией типа данных.

Аргументами функций могут быть:

- простые значения: `1`;
- вызовы других функций: `add 1 1`;
- конвейеры: `1 | add 1`;
- комбинации вышеперечисленных типов: 1 | add (add 1 1).

Если аргумент — не простое значение, а вызов другой функции или конвейер, заключите его в круглые скобки `()`:

```
{{ add 3 (add 1 1) (1 | add 1) }}
```

Чтобы игнорировать возвращаемый функцией результат, просто сохраните его в переменную `$_`:

```
{{ $_ := set $myDict "mykey" "myvalue"}}
```

4.4.3.8 Конвейеры

Конвейеры позволяют передать результат выполнения первой функции как последний аргумент во вторую функцию, а результат второй функции — как последний аргумент в третью и так далее:

```
{{ now | unixEpoch | quote }}
```

Здесь результат выполнения функции `now` (получить текущую дату) передаётся как аргумент в функцию `unixEpoch` (преобразует дату в Unix time), после чего полученное значение передаётся в функцию `quote` (оборачивает в кавычки).

Итоговый результат:

```
"1671466310"
```

Использование конвейеров не обязательно, и при желании их можно переписать следующим образом:

```
{{ quote (unixEpoch (now)) }}
```

… однако рекомендуется использовать именно конвейеры.

4.4.3.9 Логические операции и сравнения

Логические операции реализуются следующими функциями:

Операция	Функция	Пример
НЕ	`not <arg>`	`{{ not false }}`
И	`and <arg> <arg> [<arg>, ...]`	`{{ and true true }}`
ИЛИ	`or <arg> <arg> [<arg>, ...]`	`{{ or false true }}`

Сравнения реализуются следующими функциями:

Сравнение	Функция	Пример
Эквивалентно	`eq <arg> <arg> [<arg>, ...]`	`{{ eq "hello" "hello" }}`
Не эквивалентно	`neq <arg> <arg> [<arg>, ...]`	`{{ neq "hello" "world" }}`
Меньше	`lt <arg> <arg>`	`{{ lt 1 2 }}`
Больше	`gt <arg> <arg>`	`{{ gt 2 1 }}`
Меньше или эквивалентно	`le <arg> <arg>`	`{{ le 1 2 }}`
Больше или эквивалентно	`ge <arg> <arg>`	`{{ ge 2 1 }}`

Пример комбинирования:

```
{{ and (eq true true) (neq true false) (not (empty "hello")) }}
```

4.4.3.10 Ветвления

Ветвления `if/else` позволяют выполнять шаблонизацию только при выполнении/невыполнении определенных условий. Пример:

```
{{ if $.Values.app.enabled }}
# ...
{{ end }}
```

Условие считается невыполненным, если результатом его вычисления является:

- логическое `false`;
- число `0`;
- пустая строка `""`;
- пустой список `[]`;
- пустой словарь `{}`;
- нуль: `nil`.

В остальных случаях условие считается выполненным. Условием могут быть данные, переменная, функция или конвейер.

Полный пример:

```
{{ if eq $appName "backend" }}
app: mybackend
{{ else if eq $appName "frontend" }}
app: myfrontend
{{ else }}
app: {{ $appName }}
{{ end }}
```

Простые ветвления можно реализовывать не только с `if/else`, но и с функцией `ternary`. Например, следующее выражение с `ternary`:

```
{{ ternary "mybackend" $appName (eq $appName "backend") }}
… аналогично приведенной ниже конструкции if/else:
{{ if eq $appName "backend" }}
app: mybackend
{{ else }}
app: {{ $appName }}
{{ end }}
```

4.4.3.11 Циклы

4.4.3.11.1 Циклы по спискам

Циклы `range` позволяют перебирать элементы списка и выполнять нужную шаблонизацию на каждой итерации:

```
{{ range $urls }}
{{ . }}
{{ end }}
```

Результат:

```
https://example.org
https://sub.example.org
```

Относительный контекст `.` всегда указывает на элемент списка, соответствующий текущей итерации, хотя указатель можно сохранить и в произвольную переменную:

```
{{ range $elem := $urls }}
{{ $elem }}
{{ end }}
```

Результат будет таким же:

```
https://example.org
```

```
https://sub.example.org
Получить индекс элемента в списке можно следующим образом:
{{ range $i, $elem := $urls }}
{{ $elem }} имеет индекс {{ $i }}
{{ end }}
```

Результат:

```
https://example.org имеет индекс 0
https://sub.example.org имеет индекс 1
```

4.4.3.11.2 Циклы по словарям

Циклы `range` позволяют перебирать ключи и значения словарей и выполнять нужную шаблонизацию на каждой итерации:

```
# values.yaml:
apps:
  backend:
    image: openjdk
  frontend:
    image: node
```

```
# templates/app.yaml:
{{ range $.Values.apps }}
{{ .image }}
{{ end }}
```

Результат:

```
openjdk
node
```

Относительный контекст `.` всегда указывает на значение элемента словаря, соответствующего текущей итерации, при этом указатель можно сохранить и в произвольную переменную:

```
{{ range $app := $.Values.apps }}
{{ $app.image }}
{{ end }}
```

Результат будет таким же:

```
openjdk
node
```

Получить ключ элемента словаря можно так:

```
{{ range $appName, $app := $.Values.apps }}
{{ $appName }}: {{ $app.image }}
{{ end }}
```

Результат:

```
backend: openjdk
frontend: node
```

4.4.3.11.3 Контроль выполнения цикла

Специальное действие `continue` позволяет пропустить текущую итерацию цикла. В качестве примера пропустим итерацию для элемента `https://example.org`:

```
{{ range $url := $urls }}
{{ if eq $url "https://example.org" }}{{ continue }}{{ end }}
{{ $url }}
{{ end }}
```

Специальное действие `break` позволяет не только пропустить текущую итерацию, но и прервать весь цикл:

```
{{ range $url := $urls }}
{{ if eq $url "https://example.org" }}{{ break }}{{ end }}
{{ $url }}
{{ end }}
```

4.4.3.12 Контекст

4.4.3.12.1 Корневой контекст ($)

Корневой контекст — словарь, на который ссылается переменная `$`. Через него доступны values и некоторые специальные объекты. Корневой контекст имеет глобальную видимость в пределах файла-шаблона (исключение — блок `define` и некоторые функции).

Пример использования:

```
{{ $.Values.mykey }}
```

Результат:

```
myvalue
```

К корневому контексту можно добавлять произвольные ключи/значения, которые также станут доступны из любого места файла-шаблона:

```
{{ $_ := set $ "mykey" "myvalue"}}
{{ $.mykey }}
```

Результат:

```
myvalue
```

Корневой контекст остаётся неизменным даже в блоках, изменяющих относительный контекст (исключение — `define`):

```
{{ with $.Values.backend }}
- command: {{ .command }}
  image: {{ $.Values.werf.image.backend }}
{{ end }}
```

Некоторые функции вроде `tpl` или `include` могут терять корневой контекст. Для сохранения доступа к корневому контексту многим из них можно передать корневой контекст аргументом:

```
{{ tpl "{{ .Values.mykey }}" $ }}
```

Результат:

```
myvalue
```

4.4.3.12.2 Относительный контекст (.)

Относительный контекст — данные любого типа, на которые ссылается переменная `.`. По умолчанию относительный контекст указывает на корневой контекст.

Некоторые блоки и функции могут менять относительный контекст. В примере ниже в первой строке относительный контекст указывает на корневой контекст `$`, а во второй строке — уже на `$.Values.containers`:

```
{{ range .Values.containers }}
{{ . }}
{{ end }}
```

Для смены относительного контекста можно использовать блок `with`:

```
{{ with $.Values.app }}
image: {{ .image }}
{{ end }}
```

4.4.3.13 Переиспользование шаблонов

4.4.3.13.1 Именованные шаблоны

Для переиспользования шаблонизации объявите именованные шаблоны в блоках `define` в файлах `templates/_*.tpl`:

```
# templates/_helpers.tpl:
{{ define "labels" }}
app: myapp
team: alpha
{{ end }}
```

Далее подставляйте именованные шаблоны в файлы `templates/*.(yaml|tpl)` функцией `include`:

```
# templates/deployment.yaml:
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
```

```
spec:
  selector:
    matchLabels: {{ include "labels" nil | nindent 6 }}
  template:
    metadata:
      labels: {{ include "labels" nil | nindent 8 }}
```

Результат:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  selector:
    matchLabels:
      app: myapp
      team: alpha
  template:
    metadata:
      labels:
        app: myapp
        team: alpha
```

Имя именованного шаблона для функции `include` может быть динамическим:

```
{{ include (printf "%s.labels" $prefix) nil }}
```

Именованные шаблоны обладают глобальной видимостью — единожды объявленный в родительском или любом дочернем чарте именованный шаблон становится доступен сразу во всех чартах — и в родительском, и в дочерних. Убедитесь, что в подключенных родительском и дочерних чартах нет именованных шаблонов с одинаковыми именами.

4.4.3.13.2 Параметризация именованных шаблонов

Функция `include`, подставляющая именованные шаблоны, принимает один произвольный аргумент. Этот аргумент можно использовать для параметризации именованного шаблона, где этот аргумент станет относительным контекстом `.`:

```
{{ include "labels" "myapp" }}
{{ define "labels" }}
app: {{ . }}
{{ end }}
```

Результат:

```
app: myapp
```

Для передачи сразу нескольких аргументов используйте список с несколькими аргументами:

```
{{ include "labels" (list "myapp" "alpha") }}
{{ define "labels" }}
app: {{ index . 0 }}
team: {{ index . 1 }}
{{ end }}
… или словарь:
{{ include "labels" (dict "app" "myapp" "team" "alpha") }}
{{ define "labels" }}
app: {{ .app }}
team: {{ .team }}
{{ end }}
```

Необязательные позиционные аргументы можно реализовать так:

```
{{ include "labels" (list "myapp") }}
{{ include "labels" (list "myapp" "alpha") }}
{{ define "labels" }}
app: {{ index . 0 }}
{{ if gt (len .) 1 }}
team: {{ index . 1 }}
{{ end }}
{{ end }}
```

А необязательные непозиционные аргументы — так:

```
{{ include "labels" (dict "app" "myapp") }}
{{ include "labels" (dict "team" "alpha" "app" "myapp") }}
{{ define "labels" }}
app: {{ .app }}
{{ if hasKey . "team" }}
team: {{ .team }}
{{ end }}
{{ end }}
```

Именованному шаблону, не требующему параметризации, просто передайте `nil`:

```
{{ include "labels" nil }}
```

4.4.3.13.3 Результат выполнения include

Функция `include`, подставляющая именованный шаблон, **всегда возвращает только текст**. Для возврата структурированных данных нужно десериализовать результат выполнения include с помощью функции `fromYaml`:

```
{{ define "commonLabels" }}
```

```
app: myapp
{{ end }}
{{ $labels := include "commonLabels" nil | fromYaml }}
{{ $labels.app }}
```

Результат:

```
myapp
```

Обратите внимание, что `fromYaml` не работает для списков. Специально для них (и только для них) предназначена функция `fromYamlArray`.

Для явной сериализации данных можно воспользоваться функциями `toYaml` и `toJson`, для десериализации — функциями `fromYaml/fromYamlArray` и `fromJson/fromJsonArray`.

4.4.3.13.4 Контекст именованных шаблонов

Объявленные в `templates/_*.tpl` именованные шаблоны теряют доступ к корневому и относительному контекстам файла, в который они включаются функцией `include`. Исправить это можно, передав корневой и/или относительный контекст в виде аргументов `include`:

```
{{ include "labels" $ }}
{{ include "labels" . }}
{{ include "labels" (list $ .) }}
{{ include "labels" (list $ . "myapp") }}
```

4.4.3.13.5 include в include

В блоках `define` тоже можно использовать функцию `include` для включения именованных шаблонов:

```
{{ define "doSomething" }}
{{ include "doSomethingElse" . }}
{{ end }}
```

Через `include` можно вызвать даже тот именованный шаблон, из которого и происходит вызов, т. е. вызвать его рекурсивно:

```
{{ define "doRecursively" }}
{{ if ... }}
{{ include "doRecursively" . }}
{{ end }}
{{ end }}
```

4.4.3.14 Шаблонизация с tpl

Функция `tpl` позволяет выполнить шаблонизацию любой строки и тут же получить результат. Она принимает один аргумент, который должен быть корневым контекстом.

Пример шаблонизации `values`:

```
# values.yaml:
appName: "myapp"
deploymentName: "{{ .Values.appName }}-deployment"
```

```
# templates/app.yaml:
{{ tpl $.Values.deploymentName $ }}
```

Результат:

```
myapp-deployment
```

Пример шаблонизации произвольных файлов, которые сами по себе не поддерживают Helm-шаблонизацию:

```
{{ tpl ($.Files.Get "nginx.conf") $ }}
```

Для передачи дополнительных аргументов в функцию `tpl` можно добавить аргументы как новые ключи корневого контекста:

```
{{ $_ := set $ "myarg" "myvalue"}}
{{ tpl "{{ $.myarg }}" $ }}
```

4.4.3.15 Контроль отступов

Используйте функцию `nindent` для выставления отступов:

```
containers: {{ .Values.app.containers | nindent 6 }}
```

Результат:

```
containers:
- name: backend
  image: openjdk
```

Пример комбинации с другими данными:

```
containers:
{{ .Values.app.containers | nindent 6 }}
- name: frontend
  image: node
```

Результат:

```
containers:
- name: backend
  image: openjdk
- name: frontend
  image: node
```

Используйте - после {{ и/или до }} для удаления лишних пробелов до и/или после результата выполнения действия, например:

```
 {{- "hello" -}} {{ "world" }}
```

Результат:

```
helloworld
```

4.4.3.16 Комментарии

Поддерживаются два типа комментариев — комментарии шаблонизации {{ /* */ }} и комментарии манифестов #.

4.4.3.16.1 Комментарии шаблонизации

Комментарии шаблонизации скрываются при формировании манифестов:

```
{{ /* Этот комментарий пропадёт */ }}
app: myApp
```

Комментарии могут быть многострочными:

```
{{ /*
Hello
World
/* }}
```

Шаблоны в них игнорируются:

```
{{ /*
{{ print "Эта шаблонизация игнорируется" }}
/* }}
```

4.4.3.16.2 Комментарии манифестов

Комментарии манифестов сохраняются при формировании манифестов:

```
# Этот комментарий сохранится
app: myApp
```

Комментарии могут быть только однострочнными:

```
# Для многострочных комментариев используйте
# несколько однострочных комментариев подряд
```

Шаблоны в них выполняются:

```
# {{ print "Эта шаблонизация выполняется" }}
```

4.4.3.17 Отладка

Используйте `d8 dk render`, чтобы полностью сформировать и отобразить конечные Kubernetes-манифесты. Укажите опцию `--debug`, чтобы увидеть манифесты, даже если они не являются корректным YAML.

Отобразить содержимое переменной:

```
output: {{ $appName | toYaml }}
```

Отобразить содержимое переменной-списка или словаря:

```
output: {{ $dictOrList | toYaml | nindent 2 }}
```

Отобразить тип данных у переменной:

```
output: {{ kindOf $myvar }}
```

Отобразить произвольную строку, остановив дальнейшее формирование шаблонов:

```
{{ fail (printf "Тип данных: %s" (kindOf $myvar)) }}
```

4.4.4 Параметризация шаблонов

4.4.4.1 Основы параметризации

Содержимое словаря `$.Values` можно использовать для параметризации шаблонов. Каждый чарт имеет свой словарь `$.Values`. Словарь формируется слиянием параметров, полученных из файлов параметров, опций командной строки и других источников.

Простой пример параметризации через `values.yaml`:

```
# values.yaml:
myparam: myvalue
```

```
# templates/example.yaml:
{{ $.Values.myparam }}
```

Результат:

```
myvalue
```

Более сложный пример:

```
# values.yaml:
myparams:
- value: original
```

```
# templates/example.yaml:
{{ (index $.Values.myparams 0).value }}
```

```
d8 dk render --set myparams[0].value=overriden
```

Результат:

```
overriden
```

4.4.4.2 Источники параметров и их приоритет

Словарь `$.Values` формируется объединением параметров из источников параметров в указанном порядке:

1. `values.yaml` текущего чарта.
2. `secret-values.yaml` текущего чарта.
3. Словарь в `values.yaml` родительского чарта, у которого ключ — алиас или имя текущего чарта.
4. Словарь в `secret-values.yaml` родительского чарта, у которого ключ — алиас или имя текущего чарта.
5. Файлы параметров из переменной `WERF_VALUES_*`.
6. Файлы параметров из опции `--values`.
7. Файлы секретных параметров из переменной `WERF_SECRET_VALUES_*`.
8. Файлы секретных параметров из опции `--secret-values`.
9. Параметры в set-файлах из переменной `WERF_SET_FILE_*`.
10. Параметры в set-файлах из опции `--set-file`.
11. Параметры из переменной `WERF_SET_STRING_*`.
12. Параметры из опции `--set-string`.
13. Параметры из переменной `WERF_SET_*`.
14. Параметры из опции `--set`.
15. Служебные параметры ПО «Deckhouse Platform».
16. Параметры из директивы `export-values` родительского чарта.
17. Параметры из директивы `import-values` дочерних чартов.

Правила объединения параметров:

- простые типы данных перезаписываются;
- списки перезаписываются;
- словари объединяются;
- при конфликтах параметры из источников выше по списку перезаписываются параметрами из источников ниже по списку.

4.4.4.3 Параметризация чарта

Чарт можно параметризовать через его файл параметров:

```
# values.yaml:
myparam: myvalue
```

```
# templates/example.yaml:
{{ $.Values.myparam }}
```

Результат:

```
myvalue
```

Также добавить/переопределить параметры чарта можно и аргументами командной строки:

```
d8 dk render --set myparam=overriden
# или WERF_SET_MYPARAM=myparam=overriden d8 dk render
```

```
# .helm/values-production.yaml:
myparam: overriden
```

… или дополнительными файлами параметров:

```
# .helm/values-production.yaml:
myparam: overriden
```

```
d8 dk render --values .helm/values-production.yaml
# или WERF_VALUES_PROD=.helm/values-production.yaml d8 dk render
```

… или файлом секретных параметров основного чарта:

```
# .helm/secret-values.yaml:
myparam: <encrypted>
```

```
d8 dk render
```

… или дополнительными файлами секретных параметров основного чарта:

```
# .helm/secret-values-production.yaml:
myparam: <encrypted>
```

```
d8 dk render --secret-values .helm/secret-values-production.yaml
# или WERF_SECRET_VALUES_PROD=.helm/secret-values-production.yaml d8 dk
render
```

… или set-файлами:

```
# myparam.txt:
```

```
overriden
```

```
d8 dk render --set-file myparam=myparam.txt
# или WERF_SET_FILE_PROD=myparam=myparam.txt d8 dk render
```

Результат везде тот же:

```
overriden
```

4.4.4.4 Параметризация зависимых чартов

Зависимый чарт можно параметризовать как через его собственный файл параметров, так и через файл параметров родительского чарта.

К примеру, здесь параметры из словаря `mychild` в файле `values.yaml` чарта `myparent` перезаписывают параметры в файле `values.yaml` чарта `mychild`:

```
# Chart.yaml:
name: myparent
dependencies:
- name: mychild
```

```
# values.yaml:
mychild:
  myparam: overriden
```

```
# charts/mychild/values.yaml:
myparam: original
```

```
# charts/mychild/templates/example.yaml:
{{ $.Values.myparam }}
```

Результат:

```
overriden
```

Обратите внимание, что словарь, находящийся в `values.yaml` родительского чарта и содержащий параметры для зависимого чарта, должен иметь в качестве имени `alias` (если есть) или `name` зависимого чарта.

Также добавить/переопределить параметры зависимого чарта можно и аргументами командной строки:

```
# Chart.yaml:
name: myparent
dependencies:
- name: mychild
```

```
# values.yaml:
mychild:
  myparam: overriden
```

… или дополнительными файлами параметров:

```
# .helm/values-production.yaml:
mychild:
  myparam: overriden
```

```
d8 dk render --values .helm/values-production.yaml
# или WERF_VALUES_PROD=.helm/values-production.yaml d8 dk render
```

… или файлом секретных параметров основного чарта:

```
# .helm/secret-values.yaml:
mychild:
  myparam: <encrypted>
```

```
d8 dk render
```

… или дополнительными файлами секретных параметров основного чарта:

```
# .helm/secret-values-production.yaml:
mychild:
  myparam: <encrypted>
```

```
d8 dk render --secret-values .helm/secret-values-production.yaml
# или WERF_SECRET_VALUES_PROD=.helm/secret-values-production.yaml d8 dk
render
```

… или set-файлами:

```
# mychild-myparam.txt:
overriden
```

… или директивой `export-values`:

```
# Chart.yaml:
name: myparent
dependencies:
- name: mychild
  export-values:
  - parent: myparam
    child: myparam
```

```
# values.yaml:
myparam: overriden
```

```
d8 dk render
```

Результат везде тот же:

```
overriden
```

4.4.4.5 Использование параметров зависимого чарта в родительском

Для передачи параметров зависимого чарта в родительский можно использовать директиву `import-values` в родительском чарте:

```
# Chart.yaml:
name: myparent
dependencies:
- name: mychild
  import-values:
  - child: myparam
    parent: myparam
```

```
# values.yaml:
myparam: original
```

```
# charts/mychild/values.yaml:
myparam: overriden
```

```
# templates/example.yaml:
{{ $.Values.myparam }}
```

Результат:

```
overriden
```

4.4.4.6 Глобальные параметры

Параметры чарта доступны только в этом же чарте (и ограниченно доступны в зависимых от него). Один из простых способов получить доступ к параметрам одного чарта в других подключенных чартах — использование глобальных параметров.

Глобальный параметр имеет глобальную область видимости — параметр, объявленный в родительском, дочернем или другом подключенном чарте становится доступен во всех подключенных чартах по одному и тому же пути:

```
# Chart.yaml:
name: myparent
dependencies:
- name: mychild1
- name: mychild2
```

```
# charts/mychild1/values.yaml:
global:
  myparam: myvalue
```

```
# templates/example.yaml:
```

```
myparent: {{ $.Values.global.myparam }}
# charts/mychild1/templates/example.yaml:
mychild1: {{ $.Values.global.myparam }}
# charts/mychild2/templates/example.yaml:
mychild2: {{ $.Values.global.myparam }}
```

Результат:

```
myparent: myvalue
---
mychild1: myvalue
---
mychild2: myvalue
```

4.4.4.7 Секретные параметры

Для хранения секретных параметров можно использовать файлы секретных параметров, хранящиеся в зашифрованном виде в Git-репозитории.

По умолчанию ПО «Deckhouse Platform» пытается найти файл .helm/secret-values.yaml, содержащий зашифрованные параметры, и при нахождении файла расшифровывает его и объединяет расшифрованные параметры с остальными:

```
# .helm/values.yaml:
plainParam: plainValue
```

```
# .helm/secret-values.yaml:
secretParam:
1000625c4f1d874f0ab853bf1db4e438ad6f054526e5dcf4fc8c10e551174904e6d0
```

```
{{ $.Values.plainParam }}
{{ $.Values.secretParam }}
```

Результат:

```
plainValue
secretValue
```

4.4.4.7.1 Работа с файлами секретных параметров

Порядок работы с файлами секретных параметров:

1. Возьмите существующий секретный ключ или создайте новый командой `d8 dk helm secret generate-secret-key`.
2. Сохраните секретный ключ в переменную окружения `WERF_SECRET_KEY`, либо в файлы `<корень Git-репозитория>/.werf_secret_key` или `<домашняя директория>/.werf/global_secret_key`.

3. Командой `d8 dk helm secret values edit .helm/secret-values.yaml` откройте файл секретных параметров и добавьте/измените в нём расшифрованные параметры.
4. Сохраните файл — файл зашифруется и сохранится в зашифрованном виде.
5. Закоммитите в Git добавленный/изменённый файл `.helm/secret-values.yaml`;
6. При дальнейших вызовах ПО «Deckhouse Platform» секретный ключ должен быть установлен в вышеупомянутых переменной окружения или файлах, иначе файл секретных параметров не сможет быть расшифрован.

Имеющий доступ к секретному ключу может расшифровать содержимое файла секретных параметров, поэтому держите секретный ключ в безопасном месте!

При использовании файла `<корень Git-репозитория>/.werf_secret_key` обязательно добавьте его в `.gitignore`, чтобы случайно не сохранить его в Git-репозитории.

Многие команды ПО «Deckhouse Platform» можно запускать и без указания секретного ключа благодаря опции `--ignore-secret-key`, но в таком случае параметры будут доступны для использования не в расшифрованной форме, а в зашифрованной.

4.4.4.7.2 Дополнительные файлы секретных параметров

В дополнение к файлу `.helm/secret-values.yaml` можно создавать и использовать дополнительные секретные файлы:

```
# .helm/secret-values-production.yaml:
secret:
1000625c4f1d874f0ab853bf1db4e438ad6f054526e5dcf4fc8c10e551174904e6d0
```

```
d8 dk --secret-values .helm/secret-values-production.yaml
```

4.4.4.8 Информация о собранных образах

ПО «Deckhouse Platform» хранит информацию о собранных образах в параметрах `$.Values.werf` основного чарта:

```
werf:
  image:
        # Полный путь к собранному Docker-образу для ПО «Deckhouse
Platform»-образа "backend":
                                                                backend:
example.org/apps/myapp:a243949601ddc3d4133c4d5269ba23ed58cb8b18bf2b64047f
35abd2-1598024377816
  # Адрес container registry для собранных образов:
  repo: example.org/apps/myapp
  tag:
```

```
      # Тег собранного Docker-образа для ПО «Deckhouse Platform»-образа
"backend":
  backend:
a243949601ddc3d4133c4d5269ba23ed58cb8b18bf2b64047f35abd2-1598024377816
```

Пример использования:

```
image: {{ $.Values.werf.image.backend }}
```

Результат:

```
image:
example.org/apps/myapp:a243949601ddc3d4133c4d5269ba23ed58cb8b18bf2b64047f
35abd2-1598024377816
```

Для использования `$.Values.werf` в зависимых чартах воспользуйтесь директивой `export-values`:

```
# .helm/Chart.yaml:
dependencies:
- name: backend
  export-values:
  - parent: werf
    child: werf
```

```
# .helm/charts/backend/templates/example.yaml:
image: {{ $.Values.werf.image.backend }}
```

Результат:

```
image:
example.org/apps/myapp:a243949601ddc3d4133c4d5269ba23ed58cb8b18bf2b64047f
35abd2-1598024377816
```

4.4.4.9 Информация о релизе

ПО «Deckhouse Platform» хранит информацию о релизе в свойствах объекта `$.Release`:

```
# Устанавливается ли релиз в первый раз:
IsInstall: true
# Обновляется ли уже существующий релиз:
IsUpgrade: false
# Имя релиза:
Name: myapp-production
# Имя Kubernetes Namespace:
Namespace: myapp-production
# Номер ревизии релиза:
Revision: 1
```

… и в параметрах `$.Values.werf` основного чарта:

```
werf:
```

```
  # Имя ПО «Deckhouse Platform»-проекта:
  name: myapp
  # Окружение:
  env: production
```

Пример использования:

```
{{ $.Release.Namespace }}
{{ $.Values.werf.env }}
```

Результат:

```
myapp-production
production
```

Для использования `$.Values.werf` в зависимых чартах воспользуйтесь директивой `export-values:`

```
# .helm/Chart.yaml:
dependencies:
- name: backend
  export-values:
  - parent: werf
    child: werf
```

```
# .helm/charts/backend/templates/example.yaml:
{{ $.Values.werf.env }}
```

Результат:

```
production
```

4.4.4.10 Информация о чарте

ПО «Deckhouse Platform» хранит информацию о текущем чарте в объекте `$.Chart`:

```
# Является ли чарт основным:
IsRoot: true

# Содержимое Chart.yaml:
Name: mychart
Version: 1.0.0
Type: library
KubeVersion: "~1.20.3"
AppVersion: "1.0"
Deprecated: false
Icon: https://example.org/mychart-icon.svg
Description: This is My Chart
Home: https://example.org
Sources:
  - https://github.com/my/chart
Keywords:
  - apps
Annotations:
  anyAdditionalInfo: here
Dependencies:
```

```
- Name: redis
  Condition: redis.enabled
```

Пример использования:

```
{{ $.Chart.Name }}
```

Результат:

```
mychart
```

4.4.4.11 Информация о шаблоне

ПО «Deckhouse Platform» хранит информацию о текущем шаблоне в свойствах объекта `$.Template`:

```
# Относительный путь к директории templates чарта:
BasePath: mychart/templates
# Относительный путь к текущему файлу шаблона:
Name: mychart/templates/example.yaml
```

Пример использования:

```
{{ $.Template.Name }}
```

Результат:

```
mychart/templates/example.yaml
```

4.4.4.12 Информация о Git-коммите

ПО «Deckhouse Platform» хранит информацию о Git-коммите, на котором он был запущен, в параметрах `$.Values.werf.commit` основного чарта:

```
werf:
  commit:
    date:
       # Дата Git-коммита, на котором был запущен ПО «Deckhouse Platform»
(человекочитаемая форма):
      human: 2022-01-21 18:51:39 +0300 +0300
       # Дата Git-коммита, на котором был запущен ПО «Deckhouse Platform»
(Unix time):
      unix: 1642780299
    # Хэш Git-коммита, на котором был запущен ПО «Deckhouse Platform»:
    hash: 1b28e6843a963c5bdb3579f6fc93317cc028051c
```

Пример использования:

```
{{ $.Values.werf.commit.hash }}
```

Результат:

```
1b28e6843a963c5bdb3579f6fc93317cc028051c
```

Для использования `$.Values.werf.commit` в зависимых чартах воспользуйтесь директивой `export-values`:

```
# .helm/Chart.yaml:
dependencies:
- name: backend
  export-values:
  - parent: werf
    child: werf
```

```
# .helm/charts/backend/templates/example.yaml:
{{ $.Values.werf.commit.hash }}
```

Результат:

```
1b28e6843a963c5bdb3579f6fc93317cc028051c
```

4.4.4.13 Информация о возможностях кластера Kubernetes

ПО «Deckhouse Platform» предоставляет информацию о возможностях кластера Kubernetes, в который ПО «Deckhouse Platform» стал бы применять Kubernetes-манифесты, через свойства объекта `$.Capabilities`:

```
KubeVersion:
  # Полная версия кластера Kubernetes:
  Version: v1.20.0
  # Мажорная версия кластера Kubernetes:
  Major: "1"
  # Минорная версия кластера Kubernetes:
  Minor: "20"
# API, поддерживаемые кластером Kubernetes:
APIVersions:
- apps/v1
- batch/v1
- # ...
```

… и методы объекта `$.Capabilities`:

- `APIVersions.Has <arg>` — поддерживается ли кластером Kubernetes указанное аргументом API (например, `apps/v1`) или ресурс (например, `apps/v1/Deployment`).

Пример использования:

```
{{ $.Capabilities.KubeVersion.Version }}
{{ $.Capabilities.APIVersions.Has "apps/v1" }}
```

Результат:

```
v1.20.0
true
```

4.4.5 Разные окружения

4.4.5.1 Параметризация шаблонов в зависимости от окружения

Окружение ПО «Deckhouse Platform» указывается опцией `--env ($WERF_ENV)`, либо автоматически выставляется командой `d8 dk ci-env`. Текущее окружение доступно в параметре `$.Values.werf.env` основного чарта.

Окружение ПО «Deckhouse Platform» используется при формировании имени релиза и имени Namespace'а, а также может использоваться для параметризации шаблонов:

```
# .helm/values.yaml:
memory:
  staging: 1G
  production: 2G
```

```
# .helm/templates/example.yaml:
memory: {{ index $.Values.memory $.Values.werf.env }}
```

```
d8 dk render --env production
```

Результат:

```
memory: 2G
```

Для использования `$.Values.werf.env` в зависимых чартах воспользуйтесь директивой `export-values`:

```
# .helm/Chart.yaml:
dependencies:
- name: child
  export-values:
  - parent: werf
    child: werf
```

```
# .helm/charts/child/templates/example.yaml:
{{ $.Values.werf.env }}
```

Результат:

```
production
```

4.4.5.2 Развертывание в разные Kubernetes Namespace

Имя Kubernetes Namespace для развертываемых ресурсов формируется автоматически по специальному шаблону `[[ project ]]-[[ env ]]`, где `[[ project ]]` — имя проекта ПО «Deckhouse Platform», а `[[ env ]]` — имя окружения.

Достаточно изменить окружение ПО «Deckhouse Platform» и вместе с ним изменится и Namespace:

```
# werf.yaml:
project: myapp
```

```
d8 dk converge --env staging
d8 dk converge --env production
```

Результат: один экземпляр приложения развёрнут в Namespace `myapp-staging`, а второй — в `myapp-production`.

Обратите внимание, что если в манифесте Kubernetes-ресурса явно указан Namespace, то для этого ресурса будет использован именно указанный в нём Namespace.

4.4.5.2.1 Изменение шаблона имени

Если вас не устраивает специальный шаблон, из которого формируется имя Namespace, вы можете его изменить:

```
# werf.yaml:
project: myapp
deploy:
  namespace: "backend-[[ env ]]"
```

```
d8 dk converge --env production
```

Результат: приложение развёрнуто в Namespace `backend-production`.

4.4.5.2.2 Прямое указание имени Namespace

Вместо формирования имени Namespace по специальному шаблону можно указывать Namespace явно для каждой команды (рекомендуется также изменять и имя релиза):

```
d8 dk converge --namespace backend-production --release
backend-production
```

Результат: приложение развёрнуто в Namespace `backend-production`.

4.4.5.2.3 Форматирование имени Namespace

Namespace, сформированный по специальному шаблону или указанный опцией `--namespace`, приводится к формату `RFC 1123 Label Names` автоматически. Отключить автоматическое форматирование можно директивой `deploy.namespaceSlug` файла `werf.yaml`.

Вручную отформатировать любую строку согласно формату `RFC 1123 Label Names` можно командой `d8 dk slugify -f kubernetes-namespace`.

4.4.5.3 Развертывание в разные кластеры Kubernetes

По умолчанию ПО «Deckhouse Platform» развертывает Kubernetes-ресурсы в кластер, на который настроена команда `d8 k`. Для развертывания в разные кластеры можно использовать разные kube-контексты единого kube-config файла (по умолчанию — `$HOME/.kube/config`):

```
d8 dk converge --namespace backend-production --release backend-production
```

… или использовать разные kube-config файлы:

```
d8 dk converge --kube-config "$HOME/.kube/staging.config"
# или $WERF_KUBE_CONFIG=...
d8 dk converge --kube-config-base64 "$KUBE_PRODUCTION_CONFIG_IN_BASE64"
# или $WERF_KUBE_CONFIG_BASE64=...
```

4.4.5.4 Развертывание из-под разных пользователей Kubernetes

По умолчанию ПО «Deckhouse Platform» для развертывания использует пользователя Kubernetes, через которого работает команда `d8 k`. Для развертывания из-под разных пользователей используйте разные kube-контексты:

```
d8 dk converge --kube-context admin
# или $WERF_KUBE_CONTEXT=...
d8 dk converge --kube-context regular-user
```

4.4.6 Порядок развертывания

4.4.6.1 Стадии развертыванияs

Развертывание Kubernetes-ресурсов происходит в следующей последовательности:

1. Развертывание CustomResourceDefinitions из директорий crds подключенных чартов.
2. Развертывание хуков pre-install, pre-upgrade или pre-rollback по одному хуку за раз, от хуков с меньшим весом к большему. Если хук имеет зависимость от внешнего ресурса, то он развернётся только после его готовности.
3. Развертывание основных ресурсов: объединение ресурсов с одинаковым весом в группы (ресурсы без указанного веса имеют вес 0) и развертывание

по одной группе за раз, от групп с ресурсами меньшего веса к группам с ресурсами большего веса. Если ресурс в группе имеет зависимость от внешнего ресурса, то она начнёт развертывание только после его готовности.

4. Развертывание хуков post-install, post-upgrade или post-rollback по одному хуку за раз, от хуков с меньшим весом к большему. Если хук имеет зависимость от внешнего ресурса, то он развернётся только после его готовности.

4.4.6.2 Развертывание CustomResourceDefinitions

Для развертывания CustomResourceDefinitions поместите CRD-манифесты в нешаблонизируемые файлы `crds/*.yaml` в любом из подключенных чартов. При следующем развертывании эти CRD будут развернуты первыми, а хуки и основные ресурсы будут развернуты только после них.

Пример:

```
# .helm/crds/crontab.yaml:
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
# ...
spec:
  names:
    kind: CronTab
```

```
# .helm/templates/crontab.yaml:
apiVersion: example.org/v1
kind: CronTab
# ...
```

```
d8 dk converge
```

Результат: сначала развернут CRD для CronTab-ресурса, а затем развернут сам CronTab-ресурс.

4.4.6.3 Изменение порядка развертывания ресурсов

По умолчанию ПО «Deckhouse Platform» объединяет все основные ресурсы (основные — не являющиеся хуками или CRDs из `crds/*.yaml`) в одну группу, создаёт ресурсы этой группы, а затем отслеживает их готовность.

Создание ресурсов группы происходит в следующем порядке:

- Namespace;

- NetworkPolicy;
- ResourceQuota;
- LimitRange;
- PodSecurityPolicy;
- PodDisruptionBudget;
- ServiceAccount;
- Secret;
- SecretList;
- ConfigMap;
- StorageClass;
- PersistentVolume;
- PersistentVolumeClaim;
- CustomResourceDefinition;
- ClusterRole;
- ClusterRoleList;
- ClusterRoleBinding;
- ClusterRoleBindingList;
- Role;
- RoleList;
- RoleBinding;
- RoleBindingList;
- Service;
- DaemonSet;
- Pod;
- ReplicationController;
- ReplicaSet;
- Deployment;
- HorizontalPodAutoscaler;
- StatefulSet;
- Job;
- CronJob;
- Ingress;
- APIService.

Отслеживание готовности включается для всех ресурсов группы одновременно сразу после создания всех ресурсов группы.

Для изменения порядка развертывания ресурсов можно создать новые группы ресурсов через задание ресурсам веса, отличного от веса по умолчанию 0. Все ресурсы с одинаковым весом объединяются в группы, а затем группы ресурсов развертываются по очереди, от группы с меньшим весом к большему, например:

```
# .helm/templates/example.yaml:
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: database
  annotations:
    werf.io/weight: "-1"
# ...
---
apiVersion: batch/v1
kind: Job
metadata:
  name: database-migrations
# ...
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app1
  annotations:
    werf.io/weight: "1"
# ...
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app2
  annotations:
    werf.io/weight: "1"
# ...
```

```
d8 dk converge
```

Результат: сначала был развернут ресурс `database`, затем — `database-migrations`, а затем параллельно развернулись `app1` и `app2`.

4.4.6.4 Запуск задач перед/после установки, обновления, отката или удаления релиза

Для развертывания определенных ресурсов только перед или после установки, обновления, отката или удаления релиза преобразуйте ресурс в хук аннотацией `helm.sh/hook`, например:

```
# .helm/templates/example.yaml:
apiVersion: batch/v1
kind: Job
metadata:
  name: database-initialization
  annotations:
    helm.sh/hook: pre-install
# ...
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
```

```
d8 dk converge
```

Результат: ресурс `database-initialization` будет развернут только при первой установке релиза, а ресурс `myapp` будет развертываться и при установке, и при обновлении, и при откате релиза.

Аннотация `helm.sh/hook` объявляет ресурс-хуком и указывает, при каких условиях этот ресурс должен развертываться (можно указать несколько условий через запятую). Возможные условия для развертывания хука:

- `pre-install` — при установке релиза до установки основных ресурсов;
- `pre-upgrade` — при обновлении релиза до обновления основных ресурсов;
- `pre-rollback` — при откате релиза до отката основных ресурсов;
- `pre-delete` — при удалении релиза до удаления основных ресурсов;
- `post-install` — при установке релиза после установки основных ресурсов;
- `post-upgrade` — при обновлении релиза после обновления основных ресурсов;

- `post-rollback` — при откате релиза после отката основных ресурсов;
- `post-delete` — при удалении релиза после удаления основных ресурсов.

Для задания хукам порядка развертывания присвойте им разные веса (по умолчанию — 0), чтобы хуки развертывались по очереди, от хука с меньшим весом к большему, например:

```
# .helm/templates/example.yaml:
apiVersion: batch/v1
kind: Job
metadata:
  name: first
  annotations:
    helm.sh/hook: pre-install
    helm.sh/hook-weight: "-1"
# ...
---
apiVersion: batch/v1
kind: Job
metadata:
  name: second
  annotations:
    helm.sh/hook: pre-install
# ...
---
apiVersion: batch/v1
kind: Job
metadata:
  name: third
  annotations:
    helm.sh/hook: pre-install
    helm.sh/hook-weight: "1"
# ...
```

```
d8 dk converge
```

Результат: сначала будет развернут хук `first`, затем хук `second`, затем хук `third`.

По умолчанию при повторных развертываниях того же самого хука старый хук в кластере удаляется прямо перед развертыванием нового хука. Этап удаления старого хука можно изменить аннотацией `helm.sh/hook-delete-policy`, которая принимает следующие значения:

- `hook-succeeded` — удалять новый хук сразу после его удачного развертывания, при неудачном развертывании не удалять совсем;
- `hook-failed` — удалять новый хук сразу после его неудачного развертывания, при удачном развертывании не удалять совсем;
- `before-hook-creation` — (по умолчанию) удалять старый хук сразу перед созданием нового.

4.4.6.5 Ожидание готовности ресурсов, не принадлежащих релизу

Развертываемым в текущем релизе ресурсам могут требоваться ресурсы, которые не принадлежат текущему релизу. ПО «Deckhouse Platform» может дожидаться готовности этих внешних ресурсов благодаря аннотации `<name>.external-dependency.werf.io/resource`, например:

```
# .helm/templates/example.yaml:
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  annotations:
                                secret.external-dependency.werf.io/resource:
secret/my-dynamic-vault-secret
# ...
```

```
d8 dk converge
```

Результат: Deployment `myapp` начнёт развертывание только после того, как Secret `my-dynamic-vault-secret`, создаваемый автоматически оператором в кластере, будет создан и готов.

А так можно ожидать готовности сразу нескольких внешних ресурсов:

```
# .helm/templates/example.yaml:
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  annotations:
      secret.external-dependency.werf.io/resource:
secret/my-dynamic-vault-secret
```

```
                                    database.external-dependency.werf.io/resource:
statefulset/my-database
      # ...
```

По умолчанию ПО «Deckhouse Platform» ищет внешний ресурс в Namespace релиза (если, конечно, ресурс не кластерный). Namespace внешнего ресурса можно изменить аннотацией `<name>.external-dependency.werf.io/namespace`:

```
      # .helm/templates/example.yaml:
      apiVersion: apps/v1
      kind: Deployment
      metadata:
        name: myapp
        annotations:
                                      secret.external-dependency.werf.io/resource:
secret/my-dynamic-vault-secret
          secret.external-dependency.werf.io/namespace: my-namespace
```

Обратите внимание, что ожидать готовность внешнего ресурса будут и все другие ресурсы релиза с тем же весом, так как ресурсы объединяются по весу в группы и развертываются именно группами.

4.4.7 Сценарии развертывания

4.4.7.1 Обычное развертывание

Обычно развертывание осуществляется командой `d8 dk converge`, которая собирает образы и развертывает приложение, но требует запуска из Git-репозитория приложения. Пример:

```
d8 dk converge --repo example.org/mycompany/myapp
```

Если требуется разделить шаги сборки и развертывания, то это можно сделать так:

```
d8 dk build --repo example.org/mycompany/myapp
```

```
d8 dk converge --require-built-images --repo example.org/mycompany/myapp
```

4.4.7.2 Развертывание с использованием произвольных тегов образов

По умолчанию собранные образы получают тег на основе их содержимого, который становится доступен в Values для их дальнейшего использования в шаблонах при

развертывании. Но если возникает необходимость тегировать образы иным тегом, то можно использовать параметр `--use-custom-tag`, например:

```
d8 dk converge --use-custom-tag '%image%-v1.0.0' --repo example.org/mycompany/myapp
```

Результат: образы были собраны и опубликованы с тегами `<имя image>-v1.0.0`, после чего теги этих образов стали доступны в Values, на основе которых были сформированы и применены конечные манифесты Kubernetes.

В имени тега, указываемом в параметре `--use-custom-tag`, можно использовать шаблоны `%image%`, `%image_slug%` и `%image_safe_slug%` для подставления имени образа и `%image_content_based_tag%` для подставления оригинального тега на основе содержимого.

Обратите внимание, что при указании произвольного тега публикуется также и образ с тегом на основе содержимого. В дальнейшем при вызове `d8 dk cleanup` образ с тегом на основе содержимого и образы с произвольными тегами удаляются вместе.

Если требуется разделить шаги сборки и развертывания, то это можно сделать так:

```
d8 dk build --add-custom-tag '%image%-v1.0.0' --repo example.org/mycompany/myapp
```

```
d8 dk converge --require-built-images --use-custom-tag '%image%-v1.0.0' --repo example.org/mycompany/myapp
```

4.4.7.3 Развертывание без доступа к Git-репозиторию приложения

Если нужно развернуть приложение без доступа к Git-репозиторию приложения, то необходимо выполнить три шага:

1. Сборка образов и их публикация в container registry.
2. Добавление переданных параметров и публикация основного чарта в OCI-репозиторий. Чарт содержит указатели на опубликованные в первом шаге образы.
3. Применение опубликованного бандла в кластер.

Первые два шага выполняются командой `d8 dk bundle publish`, находясь в Git-репозитории приложения, например:

```
d8 dk bundle publish --tag latest --repo example.org/mycompany/myapp
```

А третий шаг выполняется командой `d8 dk bundle apply` уже без необходимости находиться в Git-репозитории приложения, например:

```
d8 dk bundle apply --tag latest --release myapp --namespace myapp-production --repo example.org/mycompany/myapp
```

Конечный результат будет тот же самый, что и при использовании `d8 dk converge`.

Если требуется разделить первый и второй шаг, то это можно сделать так:

```
d8 dk build --repo example.org/mycompany/myapp
```

```
d8 dk bundle publish --require-built-images --tag latest --repo example.org/mycompany/myapp
```

4.4.7.4 Развертывание без доступа к Git-репозиторию и container registry приложения

Если нужно развернуть приложение без доступа к Git-репозиторию приложения и без доступа к container registry приложения, то необходимо выполнить пять шагов:

1. Сборка образов и их публикация в container registry приложения.
2. Добавление переданных параметров и публикация основного чарта в OCI-репозиторий. Чарт содержит указатели на опубликованные в первом шаге образы.
3. Экспорт бандла и связанных с ним образов в локальный архив.
4. Импорт заархивированного бандла и его образов в container registry, доступный из Kubernetes-кластера, используемого для развертывания.
5. Применение в кластер бандла, опубликованного в новом container registry.

Первые два шага выполняются командой `d8 dk bundle publish`, находясь в Git-репозитории приложения, например:

```
d8 dk bundle publish --tag latest --repo example.org/mycompany/myapp
```

Третий шаг выполняется командой `d8 dk bundle copy` уже без необходимости находиться в Git-репозитории приложения, например:

```
d8 dk bundle copy --from example.org/mycompany/myapp:latest --to archive:myapp-latest.tar.gz
```

Теперь полученный локальный архив `myapp-latest.tar.gz` переносится удобным способом туда, откуда имеется доступ в container registry, используемый для развертывания в Kubernetes-кластер, и снова выполняется команда `d8 dk bundle copy`, например:

```
d8 dk bundle copy --from archive:myapp-latest.tar.gz --to registry.internal/mycompany/myapp:latest
```

В результате чарт и связанные с ним образы опубликуются в новый container registry, к которому из Kubernetes-кластера уже есть доступ. Осталось только развернуть опубликованный бандл в кластер командой `d8 dk bundle apply`, например:

```
d8 dk bundle apply --tag latest --release myapp --namespace myapp-production --repo registry.internal/mycompany/myapp
```

На этом шаге уже не требуется доступ ни в Git-репозиторий приложения, ни в его первоначальный container registry. Конечный результат развертывания бандла будет тот же самый, что и при использовании `d8 dk converge`.

Если требуется разделить первый и второй шаг, то это можно сделать так:

```
d8 dk build --repo example.org/mycompany/myapp
```

```
d8 dk bundle publish --require-built-images --tag latest --repo example.org/mycompany/myapp
```

4.4.7.5 Развертывание сторонним инструментом

Если нужно выполнить применение конечных манифестов приложения не с ПО «Deckhouse Platform», а с использованием другого инструмента (kubectl, Helm, …), то необходимо выполнить три шага:

1. Сборка образов и их публикация в container registry.
2. Формирование конечных манифестов.
3. Развертывание получившихся манифестов в кластер, используя сторонний инструмент.

Первые два шага выполняются командой `d8 dk render`, находясь в Git-репозитории приложения:

```
d8 dk render --output manifests.yaml --repo example.org/mycompany/myapp
```

Теперь полученные манифесты можно передать в сторонний инструмент для дальнейшего развертывания, например:

```
d8 k apply -f manifests.yaml
```

Обратите внимание, что некоторые специальные возможности ПО «Deckhouse Platform» вроде возможности изменения порядка развертывания ресурсов на основании их

веса (аннотация `werf.io/weight`) скорее всего не будут поддерживаться при применении манифестов сторонним инструментом.

Если требуется разделить первый и второй шаг, то это можно сделать так:

```
d8 dk build --repo example.org/mycompany/myapp
```

```
d8 dk render --require-built-images --output manifests.yaml --repo example.org/mycompany/myapp
```

4.4.7.6 Развертывание сторонним инструментом без доступа к Git-репозиторию приложения

Если нужно выполнить применение конечных манифестов приложения не с ПО «Deckhouse Platform», а с использованием другого инструмента (kubectl, Helm, …), при этом не имея доступа к Git-репозиторию приложения, то необходимо выполнить четыре шага:

Сборка образов и их публикация в container registry.

Добавление переданных параметров и публикация основного чарта в OCI-репозиторий. Чарт содержит указатели на опубликованные в первом шаге образы.

Формирование из бандла конечных манифестов.

Развертывание получившихся манифестов в кластер используя сторонний инструмент.

Первые два шага выполняются командой `d8 dk bundle publish`, находясь в Git-репозитории приложения:

```
d8 dk bundle publish --tag latest --repo example.org/mycompany/myapp
```

А третий шаг выполняется командой `d8 dk bundle render` уже без необходимости находиться в Git-репозитории приложения, например

```
d8 dk bundle render --output manifests.yaml --tag latest --release myapp
--namespace myapp-production --repo example.org/mycompany/myapp
```

Теперь полученные манифесты можно передать в сторонний инструмент для дальнейшего развертывания, например:

```
d8 k apply -f manifests.yaml
```

Обратите внимание, что некоторые специальные возможности ПО «Deckhouse Platform», вроде возможности изменения порядка развертывания ресурсов на основании

их веса (аннотация `werf.io/weight`), скорее всего не будут поддерживаться при применении манифестов сторонним инструментом.

Если требуется разделить первый и второй шаг, то это можно сделать так:

```
d8 dk build --repo example.org/mycompany/myapp
```

`d8 dk bundle publish --require-built-images --tag latest --repo example.org/mycompany/myapp`

4.4.7.7 Сохранение отчета о развертывании

Команды `d8 dk converge` и `d8 dk bundle apply` имеют параметр `--save-deploy-report`, который позволяет сохранить отчёт о последнем развертывании в файл. Отчёт содержит имя релиза, Namespace, статус развертывания и ряд других данных. Пример:

```
d8 dk converge --save-deploy-report
```

Результат: после развертывания появится файл `.werf-deploy-report.json`, содержащий информацию о последнем релизе.

Путь к отчёту о развертывании можно изменить параметром `--deploy-report-path`.

4.4.7.8 Удаление развернутого приложения

Удалить развернутое приложение можно командой `d8 dk dismiss`, запущенной из Git-репозитория приложения, например:

```
d8 dk dismiss --env staging
```

При отсутствии доступа к Git-репозиторию приложения можно явно указать имя релиза и Namespace:

```
d8 dk dismiss --release myapp-staging --namespace myapp-staging
```

… или использовать отчёт о предыдущем развертывании, включаемый опцией `--save-deploy-report` у `d8 dk converge` и `d8 dk bundle apply`, который содержит имя релиза и Namespace:

```
d8 dk converge --save-deploy-report
cp .werf-deploy-report.json /anywhere
cd /anywhere
d8 dk dismiss --use-deploy-report
```

Путь к отчёту о развертывании можно изменить параметром `--deploy-report-path`.

4.4.8 Отслеживание ресурсов

4.4.8.1 Отслеживание состояния ресурсов

Развертывание ресурсов делится на две стадии: применение ресурсов в кластер и отслеживание состояния этих ресурсов. ПО «Deckhouse Platform» реализует продвинутое отслеживание состояния ресурсов благодаря библиотеке `kubedog`.

Отслеживание ресурсов включено по умолчанию для всех поддерживаемых ресурсов, а именно для:

- всех ресурсов релиза;
- некоторых ресурсов, опосредованно создаваемых ресурсами релиза;
- ресурсов вне релиза, указанных в аннотациях `<name>.external-dependency.werf.io/resource`.

Для ресурсов Deployment, StatefulSet, DaemonSet, Job и Flagger Canary задействуются специальные отслеживатели состояния, которые не только точно определяют, удачно или неудачно ресурс был развернут, но и отслеживают состояние дочерних ресурсов, таких как Pod’ы, создаваемые Deployment’ом.

Для остальных ресурсов, не имеющих специальных отслеживателей состояния, задействуется универсальный отслеживатель, который предполагает удачность развертывания ресурса на основании доступной в кластере информации о ресурсе. В редких случаях, если универсальный отслеживатель ошибается в своих предположениях, отслеживание для этого ресурса можно отключить.

4.4.8.2 Изменение критериев неудачного развертывания ресурса

По умолчанию ПО «Deckhouse Platform» прерывает развертывание и помечает его как неудачное, если произошло более двух ошибок при развертывании одного из ресурсов.

Изменить максимальное количество ошибок развертывания для ресурса можно аннотацией `werf.io/failures-allowed-per-replica`, например:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  annotations:
    werf.io/failures-allowed-per-replica: "5"
```

Если ресурс имеет аннотацию `werf.io/fail-mode: HopeUntilEndOfDeployProcess`, то ошибки его развертывания будут учитываться только после того, как все остальные ресурсы удачно развернутся.

А помеченный аннотацией `werf.io/track-termination-mode: NonBlocking` ресурс будет отслеживаться только пока все остальные ресурсы не будут развернуты, после чего этот ресурс автоматически посчитается развернутым, даже если это не так.

Универсальный отслеживатель отсутствие активности у ресурса в течение 4 минут считает ошибкой развертывания. Изменить этот период можно аннотацией `werf.io/no-activity-timeout`, например:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  annotations:
    werf.io/no-activity-timeout: 10m
```

4.4.8.3 Отключение отслеживания состояния и игнорирование ошибок ресурса

Для отключения отслеживания состояния ресурса и игнорирования ошибок его развертывания пометьте ресурс аннотациями `werf.io/fail-mode: IgnoreAndContinueDeployProcess` и `werf.io/track-termination-mode: NonBlocking`, например:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  annotations:
    werf.io/fail-mode: IgnoreAndContinueDeployProcess
    werf.io/track-termination-mode: NonBlocking
```

4.4.8.4 Отображение логов контейнеров

Благодаря библиотеке `kubedog` отображаются логи контейнеров, создаваемых при развертывании Deployment, StatefulSet, DaemonSet и Job.

Выключить отображение логов для ресурса можно аннотацией `werf.io/skip-logs: "true"`, например:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  annotations:
```

```
    werf.io/skip-logs: "true"
```

А в аннотации `werf.io/show-logs-only-for-containers` можно явно перечислить контейнеры, логи которых следует отображать, в то же время скрыв логи всех остальных контейнеров, например:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  annotations:
    werf.io/show-logs-only-for-containers: "backend,frontend"
```

… или наоборот — в аннотации `werf.io/skip-logs-for-containers` перечислить контейнеры, логи которых не следует отображать, в то же время отображая логи всех остальных контейнеров, например:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  annotations:
    werf.io/skip-logs-for-containers: "sidecar"
```

Для отображения только тех строк лога, которые соответствуют регулярному выражению, используйте аннотацию `werf.io/log-regex`, например:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  annotations:
    werf.io/log-regex: ".*ERROR.*"
```

Возможно отфильтровать строки лога согласно регулярному выражению не для всех контейнеров сразу, а только для определённого контейнера, если использовать аннотацию `werf.io/log-regex-for-<имя контейнера>`, например:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  annotations:
    werf.io/log-regex-for-backend: ".*ERROR.*"
```

4.4.8.5 Отображение Events ресурсов

Благодаря библиотеке `kubedog` отображаются Events отслеживаемых ресурсов, если ресурс имеет аннотацию `werf.io/show-service-messages: "true"`, например:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  annotations:
    werf.io/show-service-messages: "true"
```

4.4.9 Управление релизами

4.4.9.1 О релизах

Результатом развертывания является релиз — совокупность развернутых в кластере ресурсов и служебной информации.

Технически релизы ПО «Deckhouse Platform» являются релизами Helm 3 и полностью с ними совместимы. Служебная информация по умолчанию хранится в специальном Secret-ресурсе.

4.4.9.2 Автоматическое формирование имени релиза

По умолчанию имя релиза формируется автоматически по специальному шаблону `[[ project ]]-[[ env ]]`, где `project` — имя проекта ПО «Deckhouse Platform», а `env` — имя окружения, например:

```
# werf.yaml:
project: myapp
```

```
d8 dk converge --env staging
d8 dk converge --env production
```

Результат: созданы релизы `myapp-staging` и `myapp-production`.

4.4.9.3 Изменение шаблона имени релиза

Если вас не устраивает специальный шаблон, из которого формируется имя релиза, вы можете его изменить:

```
# werf.yaml:
project: myapp
deploy:
  helmRelease: "backend-[[ env ]]"
```

```
d8 dk converge --env production
```

Результат: создан релиз `backend-production`.

4.4.9.4 Прямое указание имени релиза

Вместо формирования имени релиза по специальному шаблону можно указывать имя релиза явно для каждой команды:

```
d8 dk converge --release backend-production  # или $WERF_RELEASE=...
```

Результат: создан релиз `backend-production`.

4.4.9.5 Форматирование имени релиза

Имя релиза, сформированное по специальному шаблону или указанное опцией `--release`, приводится к формату `RFC 1123 Label Names` автоматически. Отключить автоматическое форматирование можно директивой `deploy.helmReleaseSlug` файла `werf.yaml`.

Вручную отформатировать любую строку согласно формату `RFC 1123 Label Names` можно командой d8 dk slugify -f helm-rele`ase`.

4.4.9.6 Добавление в релиз уже существующих в кластере ресурсов

ПО «Deckhouse Platform» не позволяет развернуть новый ресурс релиза поверх уже существующего в кластере ресурса, если ресурс в кластере не является частью текущего релиза. Такое поведение предотвращает случайные обновления ресурсов, принадлежащих другому релизу или развернутых без ПО «Deckhouse Platform». Если все же попытаться это сделать, то отобразится следующая ошибка:

`Error: helm upgrade have failed: UPGRADE FAILED: rendered manifests contain a resource that already exists...`

Чтобы добавить ресурс в кластере в текущий релиз и разрешить его обновление, выставьте ресурсу в кластере аннотации `meta.helm.sh/release-name: <имя текущего релиза>`, meta.helm.sh/release-nam`espace: <Namespace текущего релиза>` и лейбл `app.kubernetes.io/managed-by: Helm`, например:

```
d8 k annotate deploy/myapp meta.helm.sh/release-name=myapp-production
d8 k annotate deploy/myapp meta.helm.sh/release-namespace=myapp-production
d8 k label deploy/myapp app.kubernetes.io/managed-by=Helm
```

… после чего перезапустите развертывание:

```
d8 dk converge
```

Результат: ресурс релиза myapp успешно обновил ресурс `myapp` в кластере и теперь ресурс в кластере является частью текущего релиза.

4.4.9.7 Автоматическое аннотирование выкатываемых ресурсов релиза

ПО «Deckhouse Platform» автоматически выставляет следующие аннотации всем ресурсам чарта в процессе развёртывания:

- `"werf.io/version": FULL_WERF_VERSION` — версия ПО «Deckhouse Platform», использованная в процессе запуска команды `d8 dk converge`;
- `"project.werf.io/name": PROJECT_NAME` — имя проекта, указанное в файле конфигурации `werf.yaml`;
- `"project.werf.io/env": ENV` — имя окружения, указанное с помощью параметра `--env` или переменной окружения `WERF_ENV` (аннотация не устанавливается, если окружение не было указано при запуске).

При использовании команды `d8 dk ci-env` с поддерживаемыми CI/CD системами добавляются аннотации, которые позволяют пользователю перейти в связанный пайплайн, задание и коммит при необходимости.

4.4.9.8 Добавление произвольных аннотаций и лейблов для выкатываемых ресурсов релиза

Пользователь может устанавливать произвольные аннотации и лейблы используя CLI-параметры при развёртывании `--add-annotation annoName=annoValue` (может быть указан несколько раз) и `--add-label labelName=labelValue` (может быть указан несколько раз). Аннотации и лейблы так же могут быть заданы с помощью соответствующих переменных `WERF_ADD_LABEL*` и `WERF_ADD_ANNOTATION*` (к примеру, `WERF_ADD_ANNOTATION_1=annoName1=annoValue1` и `WERF_ADD_LABEL_1=labelName1=labelValue1`).

Например, для установки аннотаций и лейблов `commit-sha=9aeee03d607c1eed133166159fbea3bad5365c57`, `gitlab-user-email=vasya@myproject.com` всем ресурсам Kubernetes в чарте, можно использовать следующий вызов команды деплоя:

```
d8 dk converge \
  --add-annotation "commit-sha=9aeee03d607c1eed133166159fbea3bad5365c57" \
  --add-label "commit-sha=9aeee03d607c1eed133166159fbea3bad5365c57" \
  --add-annotation "gitlab-user-email=vasya@myproject.com" \
  --add-label "gitlab-user-email=vasya@myproject.com" \
  --env dev \
  --repo REPO
```

4.5 Дистрибуция

4.5.1 Основы

Обычный цикл доставки приложений с ПО «Deckhouse Platform» выглядит как сборка образов, их публикация и последующее развертывание чартов, для чего бывает достаточно одного вызова команды `d8 dk converge`. Но иногда возникает необходимость разделить дистрибуцию артефактов (образы, чарты) и их развертывание, либо даже реализовать развертывание артефактов вовсе без ПО «Deckhouse Platform», а с использованием стороннего ПО.

В этом разделе рассматриваются способы дистрибуции образов и бандлов (чартов и связанных с ними образов) для их дальнейшего развертывания с ПО «Deckhouse Platform» или без него. Инструкции развертывания опубликованных артефактов можно найти в разделе «Развёртывание».

4.5.1.1 Пример дистрибуции образа

Для дистрибуции единственного образа, собираемого через Dockerfile, который потом будет развернут сторонним ПО, достаточно двух файлов и одной команды `d8 dk export`, запущенной в Git-репозитории приложения:

```
# werf.yaml:
project: myproject
configVersion: 1
---
image: myapp
dockerfile: Dockerfile
```

```
# Dockerfile:
FROM node

WORKDIR /app
COPY . .
RUN npm ci

CMD ["node", "server.js"]
```

```
d8 dk export myapp --repo example.org/myproject --tag
other.example.org/myproject/myapp:latest
```

Результат: опубликован образ приложения `other.example.org/myproject/myapp:latest`, готовый для развертывания сторонним ПО.

4.5.1.2 Пример дистрибуции бандла

Для дистрибуции бандла для дальнейшего его развертывания с ПО «Deckhouse Platform», подключения его как зависимого чарта или развертывания бандла как чарта сторонним ПО в простейшем случае достаточно трёх файлов и одной команды `d8 dk bundle publish`, запущенной в Git-репозитории приложения:

```
# werf.yaml:
project: mybundle
configVersion: 1
---
image: myapp
dockerfile: Dockerfile
```

```
# Dockerfile:
FROM node

WORKDIR /app
COPY . .
RUN npm ci

CMD ["node", "server.js"]
```

```
# .helm/templates/myapp.yaml:
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - image: {{ $.Values.werf.image.myapp }}
```

```
d8 dk bundle publish --repo example.org/bundles/mybundle
```

Результат: опубликован чарт `example.org/bundles/mybundle:latest` и связанный с ним собранный образ.

4.5.2 Образы

4.5.2.1 О дистрибуции образов

Дистрибуция собираемых ПО «Deckhouse Platform» образов для использования сторонними пользователями и/или ПО осуществляется командой `d8 dk export`. Эта команда соберёт и опубликует образы в container registry, при этом убрав все ненужные для стороннего ПО метаданные, чем полностью выведет образы из-под контроля ПО «Deckhouse Platform», позволив организовать их дальнейший жизненный цикл сторонними средствами.

Опубликованные командой `d8 dk export` образы никогда не будут удаляться командой `d8 dk cleanup`, в отличие от образов, опубликованных обычным способом. Очистка экспортированных образов должна быть реализована сторонними средствами.

4.5.2.2 Дистрибуция образа

```
d8 dk bundle publish --repo example.org/bundles/mybundle
```

Результат: образ собран и сначала опубликован с content-based тегом в container registry `example.org/myproject`, а затем опубликован в другой container registry `other.example.org/myproject` как целевой экспортированный образ `other.example.org/myproject/myapp:latest`.

В параметре `--tag` можно указать тот же репозиторий, что и в `--repo`, таким образом используя один и тот же container registry и для сборки, и для экспортированного образа.

4.5.2.3 Дистрибуция нескольких образов

В параметре `--tag` можно использовать шаблоны `%image%`, `%image_slug%` и `%image_safe_slug%` для подставления имени образа из `werf.yaml`, основанном на его содержимом, например:

```
d8 dk export \
```

```
    --repo example.org/mycompany/myproject \
    --tag example.org/mycompany/myproject/%image%:latest
```

4.5.2.4 Дистрибуция произвольных образов

Используя позиционные аргументы и имена образов из `werf.yaml` можно выбрать произвольные образы, например:

```
d8 dk export backend frontend \
    --repo example.org/mycompany/myproject \
    --tag example.org/mycompany/myproject/%image%:latest
```

4.5.2.5 Использование content-based-тега при формировании тега

В параметре `--tag` можно использовать шаблон `%image_content_based_tag%` для использования тега образа, основанном на его содержимом, например:

```
d8 dk export \
    --repo example.org/mycompany/myproject \
  --tag example.org/mycompany/myproject/myapp:%image_content_based_tag%
```

4.5.2.6 Использование content-based-тега при формировании тега

Используя параметр `--add-label` можно добавить произвольное количество дополнительных лейблов к экспортируемому образу(ам), например:

```
d8 dk export \
    --repo example.org/mycompany/myproject \
    --tag registry.werf.io/werf/werf:latest \
                                                                --add-label
io.artifacthub.package.readme-url=https://raw.githubusercontent.com/werf/
werf/main/README.md \
    --add-label org.opencontainers.image.created=2023-03-13T11:55:24Z \
      --add-label org.opencontainers.image.description="Official image to
run ПО «Deckhouse Platform» in containers"
```

4.5.3 Бандлы и чарты

4.5.3.1 О бандлах и чартах

Бандл — это способ дистрибуции чарта и связанных с ним образов как единого целого.

Командой `d8 dk bundle publish` можно опубликовать чарт и связанные образы для дальнейшего развёртывания с ПО «Deckhouse Platform». При этом для развёртывания доступ к Git-репозиторию приложения больше не потребуется.

Эта же команда подходит для публикации чарта. Опубликованный в OCI-репозиторий чарт может использоваться в качестве основного или зависимого чарта с ПО «Deckhouse Platform», Helm, Argo CD, Flux и другими решениями.

При упаковке ПО «Deckhouse Platform» автоматически добавляет следующие данные в чарт:

- имена собираемых образов и их динамических тегов в Values чарта;
- значения, переданных через параметры командной строки или переменные окружения, в Values чарта;
- глобальные пользовательские и служебные аннотации и лейблы для добавления в ресурсы чарта при развёртывании командой `d8 dk bundle apply`.

Опубликованный бандл (чарт и связанные с ним образы) можно копировать в другой репозиторий container registry или выгружать в/из архива с помощью одной команды `d8 dk bundle copy`.

4.5.3.2 Аутентификация в container registry

Перед работой с образами необходимо аутентифицироваться в container registry. Сделать это можно командой `d8 dk cr login`:

```
d8 dk cr login <registry url>
```

Например:

```
# Login with username and password from command line
d8 dk cr login -u username -p password registry.example.com

# Login with token from command line
d8 dk cr login -p token registry.example.com

# Login into insecure registry (over http)
d8 dk cr login --insecure-registry registry.example.com
```

В случае использования команды `d8 dk ci-env` с поддерживаемыми CI/CD-системами аутентификация во встроенные container registry выполняется в рамках команды, поэтому использование команды `d8 dk cr login` в этом случае не требуется.

4.5.3.3 Публикация бандла

Опубликовать бандл в OCI-репозиторий можно следующим способом:

1. Создайте `werf.yaml`, если его ещё нет:

```
# werf.yaml:
project: mybundle
```

```
configVersion: 1
```

2. Разместите файлы в директории основного чарта (по умолчанию `.helm` в корне Git-репозитория). Заметьте, что при публикации в чарт будут включены только следующие файлы и директории

```
.helm/
  charts/
  templates/
  crds/
  files/
  Chart.yaml
  values.yaml
  values.schema.json
  LICENSE
  README.md
```

   Для публикации дополнительных файлов/директорий выставьте переменную окружения `WERF_BUNDLE_SCHEMA_NONSTRICT=1`, после чего будут публиковаться все файлы и директории в директории основного чарта, а не только вышеуказанные.

3. Следующей командой опубликуйте бандл. Соберите и опубликуйте описанные в `werf.yaml` образы (если таковые есть), затем опубликуйте содержимое `.helm` в виде OCI-образа `example.org/bundles/mybundle:latest`:

```
d8 dk bundle publish --repo example.org/bundles/mybundle
```

4.5.3.4 Публикация нескольких бандлов из одного Git-репозитория

Разместите `.helm` с содержимым чарта и соответствующий ему `werf.yaml` в отдельную директорию для каждого бандла:

```
bundle1/
  .helm/
    templates/
    # ...
  werf.yaml
bundle2/
  .helm/
    templates/
    # ...
  werf.yaml
```

Теперь опубликуйте каждый бандл по отдельности:

```
cd bundle1
d8 dk bundle publish --repo example.org/bundles/bundle1

cd ../bundle2
d8 dk bundle publish --repo example.org/bundles/bundle2
```

4.5.3.5 Исключение файлов или директорий из публикуемого чарта

Файл `.helmignore`, находящийся в корне чарта, может содержать фильтры по именам файлов, при соответствии которым файлы или директории не будут добавляться в чарт при публикации. Формат правил такой же, как и в `.gitignore`, за исключением:

`**` не поддерживается;

`!` в начале строки не поддерживается;

`.helmignore` не исключает сам себя по умолчанию.

Также опция `--disable-default-values` для команды `d8 dk bundle publish` позволяет исключить из публикуемого чарта файл `values.yaml`.

4.5.3.6 Указание версии чарта при публикации

По умолчанию чарт публикуется с тегом `latest`. Указать иной тег, например, семантическую версию для публикуемого чарта, можно опцией `--tag`:

```
d8 dk bundle publish --repo example.org/bundles/mybundle --tag v1.0.0
```

Результат: опубликован чарт `example.org/bundles/mybundle:v1.0.0`.

Если при публикации будет обнаружено, что в OCI-репозитории уже существует чарт с таким тегом, то чарт в репозитории будет перезаписан.

4.5.3.7 Изменение версии опубликованного чарта

Для изменения тега уже опубликованного чарта скопируйте бандл с новым тегом с помощью команды `d8 dk bundle copy`, например:

```
d8 dk bundle copy --from example.org/bundles/mybundle:v1.0.0 --to
example.org/bundles/renamedbundle:v2.0.0
```

4.5.3.8 Копирование бандла в другой репозиторий

Для удобного копирования бандла в другой репозиторий имеется команда `d8 dk bundle copy`. Кроме непосредственного копирования чарта и связанных с ним образов эта команда также обновит сохранённые в чарте Values, указывающие на путь к образам.

Пример:

```
d8 dk bundle copy --from example.org/bundles/mybundle:v1.0.0 --to
example.org/bundles/renamedbundle:v2.0.0
```

4.5.3.9 Экспорт бандла из container registry в архив

После публикации бандл может быть экспортирован из репозитория в локальный архив для дальнейшей дистрибуции иными способами с помощью команды `d8 dk bundle copy`, например:

```
d8 dk bundle copy --from example.org/bundles/mybundle:v1.0.0 --to
example.org/bundles/renamedbundle:v2.0.0
```

4.5.3.10 Импорт бандла из архива в репозиторий

Экспортированный в архив бандл можно снова импортировать в тот же или другой OCI-репозиторий командой `d8 dk bundle copy`, например:

```
d8 dk bundle copy --from archive:archive.tar.gz --to
other.example.org/bundles/mybundle:v1.0.0
```

После этого вновь опубликованный бандл (чарт и его образы) снова можно использовать привычными способами.

4.5.3.11 Container registries, поддерживающие публикацию бандлов

Для публикации бандлов требуется container registry, поддерживающий спецификацию `OCI (Open Container Initiative)`. Список наиболее популярных container registries, совместимость с которыми была проверена:

Container registry	**Поддерживает публикацию бандлов**
AWS ECR	+
Azure CR	+
Docker Hub	+
GCR	+
GitHub Packages	+
GitLab Registry	+
Harbor	+

Container registry	Поддерживает публикацию бандлов
JFrog Artifactory	+
Yandex container registry	+
Nexus	+
Quay	-

4.6 Очистка

4.6.1 Очистка container registry

4.6.1.1 Обзор

Количество образов может стремительно расти, занимая больше места в container registry и, соответственно, значительно увеличивая его стоимость. Для контроля и поддержания приемлемого роста места, ПО «Deckhouse Platform» предлагает свой подход к очистке, который позволяет учитывать используемые образы в Kubernetes, а также актуальность образов по истории в Git.

Команда `d8 dk cleanup` рассчитана на периодический запуск. Удаление производится в соответствии с политиками очистки и является безопасной процедурой.

Вероятнее всего, политики очистки по умолчанию полностью покроют потребности проекта и дополнительная настройка не потребуется.

Важно отметить, что фактически размер занимаемого образами места в container registry не сокращается после очистки. ПО «Deckhouse Platform» удаляет теги неактуальных образов (манифесты), а для очистки связанных с ними данных необходимо периодически запускать сборщик мусора container registry.

4.6.1.2 Автоматизация очистки container registry

Для того, чтобы автоматизировать очистку неактуальных образов в container registry, необходимо выполнить следующие действия:

- Настроить периодический запуск `d8 dk cleanup` для удаления неактуальных тегов из container registry.
- Настроить периодический запуск сборщика мусора для непосредственного освобождения места в container registry.

4.6.1.3 Игнорирование образов, используемых в Kubernetes

ПО «Deckhouse Platform» подключается **ко всем кластерам** Kubernetes, описанным **во всех контекстах** конфигурации kubectl, и собирает имена образов для следующих типов объектов: `pod`, `deployment`, `replicaset`, `statefulset`, `daemonset`, `job`, `cronjob`, `replicationcontroller`.

Пользователь может регулировать поведение следующими параметрами (и связанными переменными окружения):

`--kube-config`, `--kube-config-base64` для определения конфигурации kubectl (по умолчанию используется пользовательская конфигурация `~/.kube/config`).

`--kube-context` для выполнения сканирования только в определённом контексте.

`--scan-context-namespace-only` для сканирования только связанного с контекстом namespace (по умолчанию все).

Сканирование Kubernetes отключается с помощью соответствующей директивы в `werf.yaml`:

```
cleanup:
  disableKubernetesBasedPolicy: true
```

Пока в кластере Kubernetes существует объект использующий образ, он никогда не удалится из container registry. Другими словами, если что-то было запущено в вашем кластере Kubernetes, то используемые образы ни при каких условиях не будут удалены при очистке.

4.6.1.4 Игнорирование свежесобранных образов

При удалении ПО «Deckhouse Platform» игнорирует образы, собранные в заданный период времени (по умолчанию за прошедшие 2 часа). При необходимости можно изменить период или совсем отключить политику соответствующими директивами в `werf.yaml`:

```
cleanup:
  disableBuiltWithinLastNHoursPolicy: false
  keepImagesBuiltWithinLastNHours: 2
```

4.6.1.5 Конфигурация политик очистки по истории Git

Конфигурация очистки состоит из набора политик, `keepPolicies`, по которым выполняется выборка значимых образов на основе истории git. Таким образом, в результате очистки **неудовлетворяющие политикам образы удаляются**.

Каждая политика состоит из двух частей:

- `references` определяет множество references, git-тегов или git-веток, которые будут использоваться при сканировании.
- `imagesPerReference` определяет лимит искомых образов для каждого reference из множества.

Любая политика должна быть связана с множеством Git-тегов (`tag`) либо Git-веток (`branch`). Можно указать определённое имя reference или задать специфичную группу, используя синтаксис регулярных выражений golang.

```
cleanup:
  disableBuiltWithinLastNHoursPolicy: false
  keepImagesBuiltWithinLastNHours: 2
```

При сканировании описанный набор git-веток будет искаться среди origin remote references, но при написании конфигурации префикс `origin/` в названии веток опускается.

Заданное множество references можно лимитировать, основываясь на времени создания git-тега или активности в git-ветке. Группа параметров `limit` позволяет писать гибкие и эффективные политики под различные workflow.

```
- references:
    branch: /^features\/.*/
    limit:
      last: 10
      in: 168h
      operator: And
```

В примере описывается выборка из не более чем 10 последних веток с префиксом `features/` в имени, в которых была какая-либо активность за последнюю неделю.

- Параметр `last` позволяет выбирать последние `n` reference’ов из определённого в `branch/tag` множества.
- Параметр `in` (синтаксис доступен в документации) позволяет выбирать Git-теги, которые были созданы в указанный период, или Git-ветки с активностью в рамках периода. Также для определённого множества `branch/tag`.

- Параметр `operator` определяет, какие reference'ы будут результатом политики: удовлетворяющие оба условия или любое из них (`And` по умолчанию).

По умолчанию при сканировании reference количество искомых образов не ограничено, но поведение может настраиваться группой параметров `imagesPerReference`:

```
imagesPerReference:
  last: int
  in: string
  operator: string
```

- Параметр `last` определяет количество искомых образов для каждого reference. По умолчанию количество не ограничено (`-1`).
- Параметр `in` (синтаксис доступен в документации) определяет период, в рамках которого необходимо выполнять поиск образов.
- Параметр `operator` определяет, какие образы сохранятся после применения политики: удовлетворяющие оба условия или любое из них (`And` по умолчанию).

Для Git-тегов проверяется только HEAD-коммит и значение `last >1` не имеет никакого смысла, является невалидным.

4.6.1.5.1 Приоритет политик для конкретного reference

При описании группы политик необходимо идти от общего к частному. Другими словами, `imagesPerReference` для конкретного reference будет соответствовать последней политике, под которую он подпадает:

```
- references:
    branch: /.*/
  imagesPerReference:
    last: 1
- references:
    branch: master
  imagesPerReference:
    last: 5
```

В данном случае, для reference master справедливы обе политики и при сканировании ветки `last` будет равен 5.

4.6.1.5.2 Политики по умолчанию

В случае, если в `werf.yaml` отсутствуют пользовательские политики очистки, используются политики по умолчанию, соответствующие следующей конфигурации:

```
cleanup:
```

```
keepPolicies:
- references:
    tag: /.*/
    limit:
      last: 10
- references:
    branch: /.*/
    limit:
      last: 10
      in: 168h
      operator: And
  imagesPerReference:
    last: 2
    in: 168h
    operator: And
- references:
    branch: /^(main|master|staging|production)$/
  imagesPerReference:
    last: 10
```

Разберём каждую политику по отдельности:

1. Сохранять по одному образу для 10 последних тегов (по дате создания).
2. Сохранять по не более чем два образа, опубликованных за последнюю неделю, для не более 10 веток с активностью за последнюю неделю.
3. Сохранять по 10 образов для веток main, master, staging и production.

4.6.1.5.3 Политики по умолчанию

Если очистка по истории Git не требуются, то её можно отключить в `werf.yaml` с помощью специальной директивы:

```
cleanup:
  disableGitHistoryBasedPolicy: true
```

4.6.1.6 Особенности работы с различными container registries

По умолчанию при удалении тегов ПО «Deckhouse Platform» использует Docker Registry API и от пользователя требуется только авторизация с использованием доступов с достаточным набором прав. Если же удаление посредством Docker Registry API не поддерживается и оно реализуется в нативном API container registry, то от пользователя могут потребоваться специфичные для используемого container registry действия.

ПО «Deckhouse Platform» пытается автоматически определить используемый container registry, используя заданный адрес репозитория (опция `--repo`). Пользователь может явно задать container registry опцией `--repo-container-registry` или переменной окружения `WERF_REPO_CONTAINER_REGISTRY`.

4.6.1.6.1 AWS ECR

При удалении тегов ПО «Deckhouse Platform» использует AWS SDK, поэтому перед очисткой container registry необходимо выполнить одно из следующих действий:

- Установить AWS CLI и выполнить конфигурацию (`aws configure`) или
- Определить `AWS_ACCESS_KEY_ID` и `AWS_SECRET_ACCESS_KEY` переменные окружения.

4.6.1.6.2 Azure CR

При удалении тегов ПО «Deckhouse Platform» использует Azure CLI, поэтому перед очисткой container registry необходимо выполнить следующие действия:

- Установить Azure CLI (`az`).
- Выполнить авторизацию (`az login`).

Пользователю необходимо иметь одну из следующих ролей: `Owner, Contributor` или `AcrDelete` (подробнее Azure CR roles and permissions).

4.6.1.6.3 Docker Hub

При удалении тегов ПО «Deckhouse Platform» использует Docker Hub API, поэтому при очистке container registry необходимо определить token или username и password.

Для получения token можно использовать следующий скрипт:

```
HUB_USERNAME=username
HUB_PASSWORD=password
HUB_TOKEN=$(curl -s -H "Content-Type: application/json" -X POST -d
'{"username": "'${HUB_USERNAME}'", "password": "'${HUB_PASSWORD}'"}'
https://hub.docker.com/v2/users/login/ | jq -r .token)
```

В качестве token нельзя использовать personal access token, т.к. удаление ресурсов возможно только при использовании основных учётных данных пользователя

Для того чтобы задать параметры, следует использовать следующие опции или соответствующие им переменные окружения:

- `--repo-docker-hub-token` или
- `--repo-docker-hub-username` и `--repo-docker-hub-password`.

4.6.1.6.4 GitHub Packages

При организации CI/CD в Github Actions мы рекомендуем использовать наш набор actions, который решит за вас большинство задач.

При удалении тегов ПО «Deckhouse Platform» использует GitHub API, поэтому при очистке container registry необходимо определить token с read:packages и delete:packages scopes.

Для того чтобы задать токен, следует использовать опцию --repo-github-token или соответствующую переменную окружения.

4.6.1.6.5 GitLab Registry

При удалении тегов ПО «Deckhouse Platform» использует GitLab container registry API или Docker Registry API в зависимости от версии GitLab.

Для удаления тега прав временного токена CI-задания (`$CI_JOB_TOKEN`) недостаточно, поэтому пользователю необходимо создать специальный токен в разделе Access Token (в секции Scope необходимо выбрать `api`) и выполнить авторизацию с ним.

4.6.1.6.6 Selectel CRaaS

При очистке ПО «Deckhouse Platform» использует Selectel CR API, поэтому при очистке container registry необходимо определить username/password, account and vpc or vpcID.

Для того чтобы задать параметры, следует использовать следующие опции или соответствующие им переменные окружения:

- `--repo-selectel-username`
- `--repo-selectel-password`
- `--repo-selectel-account`
- `--repo-selectel-vpc or`
- `--repo-selectel-vpc-id`

4.6.1.6.7 Известные проблемы

- Иногда Selectel не отдаёт токен при использовании VPC ID. Попробуйте использовать имя VPC.
- CR API не позволяет удалять теги, которые хранятся в корне registry.
- Небольшой лимит запросов в API. При активной разработке могут быть проблемы с очисткой.

4.6.1.7 Особенности работы с различными container registries

По умолчанию при удалении тегов ПО «Deckhouse Platform» использует Docker Registry API и от пользователя требуется только авторизация с использованием доступов с достаточным набором прав. Если же удаление посредством Docker Registry API не поддерживается и оно реализуется в нативном API container registry, то от пользователя могут потребоваться специфичные для используемого container registry действия.

Зона ответственности очистки ПО «Deckhouse Platform» — удаление тегов образов (манифестов), а непосредственное удаление связанных данных выполняется с помощью сборщика мусора container registry (GC).

При вызове сборщика мусора container registry должен быть переведён в режим read-only или полностью отключен. Иначе есть высокая вероятность, что опубликованные во время процедуры образы не будут учтены сборщиком и будут повреждены.

Подробнее о сборщике мусора и способах его эксплуатации можно прочитать в документации используемого container registry. Например:

- Docker Registry GC.
- GitLab CR GC.
- Harbor GC.

4.6.2 Автоматическая очистка хоста

4.6.2.1 Обзор

Очистка хоста удаляет неактуальных данные и сокращает размер кеша **автоматически** в рамках вызова основных команд ПО «Deckhouse Platform» и сразу для **всех проектов**. При необходимости очистку можно выполнять в ручном режиме с помощью команды `d8 dk host cleanup`.

4.6.2.2 Переопределение директории хранилища Docker

Параметр `--docker-server-storage-path` (или переменная окружения `WERF_DOCKER_SERVER_STORAGE_PATH`) позволяет явно задать директорию хранилища Docker в случае, если ПО «Deckhouse Platform» не может правильно определить её автоматически.

4.6.2.3 Изменение порога занимаемого места и глубины очистки хранилища Docker

Параметр `--allowed-docker-storage-volume-usage` (`WERF_ALLOWED_DOCKER_STORAGE_VOLUME_USAGE`) позволяет изменить порог занимаемого места на томе, при достижении которого выполняется очистка хранилища Docker (по умолчанию 70%).

Параметр `--allowed-docker-storage-volume-usage-margin` (`WERF_ALLOWED_DOCKER_STORAGE_VOLUME_USAGE_MARGIN`) позволяет установить глубину

очистки относительно установленного порога занимаемого места хранилища Docker (по умолчанию 5%).

4.6.2.4 Изменение порога занимаемого места и глубины очистки локального кэша

Параметр `--allowed-local-cache-volume-usage` `(WERF_ALLOWED_LOCAL_CACHE_VOLUME_USAGE)` позволяет изменить порог занимаемого места на томе, при достижении которого выполняется очистка локального кэша (по умолчанию 70%).

Параметр `--allowed-docker-storage-volume-usage-margin` `(WERF_ALLOWED_LOCAL_CACHE_VOLUME_USAGE_MARGIN)` позволяет установить глубину очистки относительно установленного порога занимаемого места локального кэша (по умолчанию 5%).

4.6.2.5 Выключение автоматической очистки

Пользователь может выключить автоматическую очистку неактуальных данных хоста с помощью параметра `--disable-auto-host-cleanup` `(WERF_DISABLE_AUTO_HOST_CLEANUP)`. В этом случае рекомендуется добавить команду `d8 dk host cleanup` в cron, например, следующим образом:

```
# /etc/cron.d/d8-d-host-cleanup
SHELL=/bin/bash
*/30 * * * * gitlab-runner d8 dk host cleanup
```

4.7 Справочник

4.7.1 werf.yaml

```
# Секция мета-информации
---
# Уникальное имя проекта приложения
project: string
# Версия конфигурации. На данный момент поддерживается единственная версия 1
configVersion: int
build: # Общие настройки сборки
     # Общий список целевых платформ для всех образов (например ['linux/amd64', 'linux/arm64',
  # 'linux/arm/v8'])
  platform: [ string, ... ]
deploy: # Настройки выката
  # Путь до директории helm чарта проекта (значение по умолчанию .helm)
  helmChartDir: string
  # Шаблон имени релиза (значение по умолчанию [[ project ]]-[[ env ]])
  helmRelease: string
  # Слагификация имени релиза (значение по умолчанию true)
  helmReleaseSlug: bool
```

```
  # Шаблон Kubernetes namespace (значение по умолчанию [[ project ]]-[[
env ]])
  namespace: string
  # Слагификация Kubernetes namespace (значение по умолчанию true)
  namespaceSlug: bool
# Настройка удаления неактульных образов
cleanup:
  # Отключить политику очистки, которая позволяет не удалять запущенные в
Kubernetes образы из
  # container registry
  disableKubernetesBasedPolicy: bool
   # Отключить политику очистки, которая позволяет не удалять образы с
учётом пользовательских
  # политик по истории Git (keepPolicies)
  disableGitHistoryBasedPolicy: bool
   # Отключить политику очистки, которая позволяет не удалять образы,
собранные в рамках заданного
  # периода времени (keepImagesBuiltWithinLastNHours)
  disableBuiltWithinLastNHoursPolicy: bool
  # Минимальное количество часов, которое должно пройти с момента сборки
образа (значение по
  # умолчанию 2)
  keepImagesBuiltWithinLastNHours: uint
  # Набор политик для выборки актуальных образов, используя историю Git
  keepPolicies:
  # Набор references, который будет использоваться при сканировании
  - references:
      # Множество git origin веток
      branch: string || /REGEXP/
      # Множество git origin тегов
      tag: string || /REGEXP/
        # Набор правил, по которым можно ограничить описанное множество
references, основываясь на
      # времени создания git-тега или активности в git-ветке
      limit:
          # Выборка последних n references из определённого в branch/tag
множества (значение по
        # умолчанию -1)
        last: int
         # Выборка git-тегов, которые были созданы в указанный период, или
git-веток с активностью
        # в рамках периода. Также для определённого множества branch/tag
        in: duration string
           # Определяет какие references будут результатом политики, те
которые удовлетворяют оба
        # условия или любое из них (значение по умолчанию And)
        operator: And || Or
    # Лимит искомых образов для каждого reference из множества
    imagesPerReference:
        # Количество искомых образов для каждого reference (значение по
умолчанию -1)
      last: int
      # Период, в рамках которого необходимо выполнять поиск образов
      in: duration string
       # Определяет какие образы сохранятся после применения политики, те
которые удовлетворяют оба
      # условия или любое из них (значение по умолчанию And)
      operator: And || Or
# Настройки связанные с работой ПО «Deckhouse Platform» с рабочей
директорией git проекта
```

```
gitWorktree:
   # Принудительно позволить ПО «Deckhouse Platform» использовать shallow
clone несмотря на ограничения
  # данного подхода
  forceShallowClone: bool
     # Разрешить процессу ПО «Deckhouse Platform» автоматически
преобразовать shallow clone проекта в
    # полный clone в процессе сборки по необходимости (значение по
умолчанию true)
  allowUnshallow: bool
    # Разрешить процессу ПО «Deckhouse Platform» автоматически скачать
новые ветки и теги из origin в
  # процессе cleanup по необходимости (значение по умолчанию true)
  allowFetchOriginBranchesAndTags: bool
# Секция Dockerfile image: может использоваться произвольное количество
секций
---
# Одно или несколько уникальных имён для образа
image: string || [ string, ... ] || ~
# Путь к Dockerfile относительно директории контекста
dockerfile: string
# Включить послойное кеширование Dockerfile-инструкций в container
registry
staged: bool
# Путь к контексту внутри папки проекта
context: string
# Список целевых платформ для данного образа (например ['linux/amd64',
'linux/arm64',
# 'linux/arm/v8'])
platform: [ string, ... ]
# Добавление нехранящихся в git файлов и директорий в сборочный контекст.
Пути должны быть
# относительно директории контекста
contextAddFiles: [ string, ... ]
# Конкретная стадия Dockerfile (по умолчанию — последняя, подобно docker
build --target)
target: string
# Переменные для ARG Dockerfile-инструкций (подобно docker build
--build-arg)
args: { name string: value string, ... }
# Установить связь host-to-IP (host:ip) (подобно docker build --add-host)
addHost: [ string, ... ]
# Сетевой режим для инструкций RUN во время сборки (подобно docker build
--network)
network: string
# Сокет агента SSH или ключи для сборки определённых слоёв (только если
используется BuildKit)
# (подобно docker build --ssh)
ssh: string
# Образы-зависимости для текущего образа
dependencies:
# Имя зависимого образа, который должен быть собран до сборки текущего
образа
- image: string
   # Определить аргументы для импорта информации о зависимом образе в
текущий образ используя
  # Dockerfile build-args (опционально)
  imports:
   # Тип импортируемой информации об образе: ImageName, ImageDigest,
ImageRepo или ImageTag
```

```
  - type: string
      # Имя аргумента (Dockerfile build-args), который будет содержать
указанный тип информации об
    # образе
    targetBuildArg: string
# Набор директив для изменения манифеста образа.
docker:
   # Включить использование незаэкранированных символов (например кавычки
и пробелы) в значениях
  # опций.
  exactValues: bool
   # Имя пользователя (или UID) и опционально пользовательская группа (или
GID).
  USER: string
  # Рабочая директория.
  WORKDIR: string
  # Точки монтирования.
  VOLUME: [ string, ... ]
  # Переменные окружения.
  ENV: { name string: value string, ... }
  # Метаданные.
  LABEL: { name string: value string, ... }
   # Описание сетевых портов, которые будут прослушиваться в запущенном
контейнере.
  EXPOSE: [ string, ... ]
   # Команда по умолчанию, которая будет выполнена при запуске контейнера
(форма записи shell или
  # exec).
  ENTRYPOINT: string | [ string, ... ]
  # Аргументы по умолчанию для ENTRYPOINT (форма записи shell или exec).
  CMD: string | [ string, ... ]
     # Инструкции, которые Docker может использовать для проверки
работоспособности запущенного
  # контейнера.
  HEALTHCHECK: string
# Точки монтирования.
mount:
# Имя служебной директории
- from: tmp_dir || build_dir
  # Абсолютный или относительный путь до произвольного файла на хосте
  fromPath: string
  # Абсолютный путь в образе
  to: string
# Импортирование из образов и артефактов.
import:
# Имя артефакта, из которого выполнять копирование файлов
- artifact: string
  # Имя образа, из которого выполнять копирование файлов
  image: string
   # Имя стадии, из которой выполнять копирование файлов (по умолчанию
последняя)
  stage: string
   # Выбор стадии импортирования файлов при сборке, до стадии install или
setup
  before: string
   # Выбор стадии импортирования файлов при сборке, после стадии install
или setup
  after: string
  # Абсолютный путь до файла или директории в выбранном образе/артефакте
  add: string
```

```
   # Абсолютный путь в конечном образе. По умолчанию соответствует пути
add
  to: string
  # Имя или UID владельца
  owner: string
  # Имя или GID группы
  group: string
  # Глобы добавления
  includePaths: [ glob, ... ]
  # Глобы исключения
  excludePaths: [ glob, ... ]
# Образы-зависимости для текущего образа
dependencies:
# Имя зависимого образа, который должен быть собран до сборки текущего
образа
- image: string
   # Выбор стадии перед которой должна быть импортирована информация об
образе (требуется указать
   # install или setup). Указанные переменные окружения будут доступны в
пользовательских стадиях
  # после указанной данной директивой стадии.
  before: string
   # Выбор стадии после которой должна быть импортирована информация об
образе (требуется указать
   # install или setup). Указанные переменные окружения будут доступны в
пользовательских стадиях
  # после указанной данной директивой стадии.
  after: string
   # Определить аргументы для импорта информации о зависимом образе в
текущий образ используя
  # переменные окружения (опционально)
  imports:
    # Тип импортируемой информации об образе: ImageName, ImageDigest,
ImageRepo или ImageTag
  - type: string
      # Имя переменной окружения, которая будет содержать указанный тип
информации об образе
    targetEnv: string
```

4.7.2 werf-giterminism.yaml

```
# Версия конфигурации. На данный момент поддерживается единственная
версия 1
giterminismConfigVersion: int
# Правила ослабления гитерминизма для CLI
cli:
  # Разрешить опцию --use-custom-tag
  allowCustomTags: bool
# Правила ослабления гитерминизма для конфигурации ПО «Deckhouse
Platform» (werf.yaml)
config:
  # Читать конфигурационный файл из директории проекта, не сверяя контент
с файлом из текущего
  # коммита и игнорируя исключения в .gitignore
  allowUncommitted: bool
  # Читать определённые шаблоны конфигурационного файла (.werf/**/*.tmpl)
из директории проекта,
```

```
  # не сверяя контент с файлами текущего коммита и игнорируя исключения в
.gitignore
  allowUncommittedTemplates: [ glob, ... ]
  # Правила для функций Go-шаблонизатора
  goTemplateRendering:
       # Разрешить определённые переменные окружения (при использовании
функции env).
    allowEnvVariables: [ string || /REGEXP/, ... ]
     # Читать определённые конфигурационные файлы из директории проекта,
не сверяя контент с
       # файлами текущего коммита и игнорируя исключения в .gitignore
(используя функции .Files.Get и
    # .Files.Glob)
    allowUncommittedFiles: [ glob, ... ]
  # Правила для dockerfile-образа
  dockerfile:
     # Читать определённые dockerfiles из директории проекта, не сверяя
контент с файлами текущего
    # коммита и игнорируя исключения в .gitignore
    allowUncommitted: [ glob, ... ]
     # Читать определённые .dockerignore-файлы из директории проекта, не
сверяя контент с файлами
    # текущего коммита и игнорируя исключения в .gitignore
    allowUncommittedDockerignoreFiles: [ glob, ... ]
      # Разрешить использование определённых файлов или директорий из
директории проекта при
    # использовании директивы contextAddFiles.
    allowContextAddFiles: [ string, ... ]
# Правила ослабления гитерминизма для helm-файлов (.helm)
helm:
   # Читать определённые helm-файлы из директории проекта, не сверяя
контент с файлами текущего
  # коммита и игнорируя исключения в .gitignore
  allowUncommittedFiles: [ glob, ... ]
```

4.7.3 Шаблонизатор werf.yaml

При чтении конфигурации `werf.yaml`, ПО «Deckhouse Platform» использует встроенный движок шаблонов Go (text/template), а также расширяет набор функций с помощью Sprig и функций ПО «Deckhouse Platform».

При организации конфигурации она может быть разбита на отдельные файлы в директории шаблонов.

4.7.4 Встроенные возможности шаблонизатора Go

Для эффективной работы мы рекомендуем изучить все возможности шаблонизатора или хотя бы ознакомиться со следующими разделами:

- Actions, основные возможности шаблонизатора.
- Переменные.
- Работа с текстом и пробелами.
- Функции.

4.7.5 Функции Sprig

Библиотека Sprig предоставляет более 70 функций:

- Строковые функции.
- Математические функции.
- Функции кодирования.
- Работа с данными в формате ключ/значение (dict).
- Работа с файловыми путями.
- Остальное.

ПО «Deckhouse Platform» не поддерживает функцию expandenv и имеет свою собственную реализацию для функции env.

4.7.6 Дополнительные функции

4.7.6.1 Различные окружения

.Env

Переменная `.Env` позволяет организовывать конфигурацию для нескольких сред (testing, staging, production и т.п.) и переключаться между ними с помощью опции `--env=<environment_name>`.

В helm шаблонах таким же образом может использоваться переменная `.Values.werf.env`.

4.7.6.2 Информация о текущем коммите

.Commit.Hash

`{{ .Commit.Hash }}` возвращает SHA текущего коммита. Если есть возможность, лучше избегать использование `.Commit.Hash`, т. к. это может приводить к ненужным пересборкам слоев.

.Commit.Date

`{{ .Commit.Date.Human }}` возвращает дату текущего коммита в человекопонятной форме.

`{{ .Commit.Date.Unix }}` возвращает дату текущего коммита в формате Unix epoch.

4.7.6.3 Шаблонизация

include

Функция include позволяет переиспользовать общие части конфигурации, а также разбивать конфигурацию на несколько файлов.

Синтаксис:

```
{{ include "<TEMPLATE_NAME>" <VALUES> }}
```

tpl

Функция `tpl` позволяет обрабатывать строки как Go-шаблоны.

Синтаксис:

{{ tpl "<STRING>" <VALUES> }}

`<STRING>` — контент файла проекта, значение переменной окружения либо произвольная строка.

`<VALUES>` — значения шаблона. При использовании текущего контекста, ., в шаблоне можно использовать все шаблоны и значения (в том числе, описанные в файлах директории шаблонов).

4.7.6.4 Переменные окружения

env

Функция `env` читает переменную окружения.

Синтаксис:

```
{{ env "<ENV_NAME>" }}
{{ env "<ENV_NAME>" "default_value" }}
```

По умолчанию, использование функции `env` запрещено гитерминизмом

4.7.6.5 Файлы проекта

.Files.Get

Функция `.Files.Get` получает содержимое определенного файла проекта.

Синтаксис:

```
{{ .Files.Get "<FILE_PATH>" }}
```

По умолчанию, использование файлов, которые имеют незакоммиченные изменения, запрещено гитерминизмом

.Files.Glob

Функция `.Files.Glob` позволяет работать с файлами проекта и их содержимым по глобу.

Функция поддерживает `shell pattern matching` и `**`. Результаты вызова функции можно объединить, используя sprig-функцию `merge` (к примеру, `{{ $filesDict := merge (.Files.Glob "glob1") (.Files.Glob "glob2") }}`)

Синтаксис:

```
{{ .Files.Glob "<GLOB>" }}
```

По умолчанию, использование файлов, которые имеют незакоммиченные изменения, запрещено гитерминизмом

4.7.6.6 Другие

required

Функция `required` проверяет наличие значения у определённой переменной. Если значение пусто, то рендеринг шаблона завершится ошибкой с заданным пользователем сообщением.

Синтаксис:

```
value: {{ required "<ERROR_MSG>" <VALUE> }}
```

fromYaml

Функция `fromYaml` декодирует YAML-документ в структуру.

Синтаксис:

```
value: {{ fromYaml "<STRING>" }}
```

4.7.7 Директория шаблонов

Файлы шаблонов могут храниться в зарезервированной директории (по умолчанию `.werf`) с расширением `.tmpl` (поддерживается произвольная вложенность `.werf/**/*.tmpl`).

Все файлы шаблонов и основная конфигурация определяют общий контекст:

- Файл шаблонов является полноценным шаблоном и может использоваться функцией `include` по относительному пути (`{{ include "directory/partial.tmpl" . }}`).
- Шаблон определённый с помощью `define` в одном файле шаблонов доступен в любом другом, включая основную конфигурацию.

4.7.8 Аннотации для деплоя

Данный раздел содержит описание аннотаций, которые меняют поведение механизма отслеживания ресурсов в процессе выката с помощью ПО «Deckhouse Platform». Все аннотации должны быть объявлены в шаблонах чарта.

- `werf.io/weight` — задает вес ресурса, который определяет порядок развертывания ресурсов.
- `<any-name>.external-dependency.werf.io/resource` — дождаться, пока указанная внешняя зависимость будет запущена, и только после этого приступить к развертыванию аннотированного ресурса.
- `<any-name>.external-dependency.werf.io/namespace` — задать пространство имен для внешней зависимости.
- `werf.io/replicas-on-creation` — задаёт количество реплик, которое должно быть установлено при первичном создании ресурса (полезно при использовании HPA).
- `werf.io/track-termination-mode` — определяет условие при котором ПО «Deckhouse Platform» остановит отслеживание ресурса.
- `werf.io/fail-mode` — определяет как ПО «Deckhouse Platform» обработает ресурс в состоянии ошибки. Ресурс в свою очередь перейдет в состояние ошибки после превышения порога допустимых ошибок, обнаруженных при отслеживании этого ресурса в процессе выката.
- `werf.io/failures-allowed-per-replica` — определяет порог ошибок, обнаруживаемых при отслеживании этого ресурса в процессе выката, после превышения которого ресурс перейдет в состояние ошибки. ПО «Deckhouse Platform» обработает это состояние в соответствии с настройкой fail mode.
- `werf.io/ignore-readiness-probe-fails-for-CONTAINER_NAME` — переопределить высчитываемый автоматически период, в течение которого неуспешные readiness-пробы будут игнорироваться и не будут переводить ресурс в состояние ошибки.
- `werf.io/no-activity-timeou`t — переопределить период неактивности, по истечении которого ресурс перейдет в состояние ошибки.
- `werf.io/log-regex` — показывать в логах только те строки вывода ресурса, которые подходят под указанный шаблон.

- `werf.io/log-regex-for-CONTAINER_NAME` — показывать в логах только те строки вывода для указанного контейнера, которые подходят под указанный шаблон.
- `werf.io/skip-logs` — выключить логирование вывода для ресурса.
- `werf.io/skip-logs-for-containers` — выключить логирование вывода для указанного контейнера.
- `werf.io/show-logs-only-for-containers` — включить логирование вывода только для указанных контейнеров ресурса.
- `werf.io/show-service-messages` — включить вывод сервисных сообщений и событий Kubernetes для данного ресурса.

Resource weight

`werf.io/weight: "NUM"`

Пример:

```
werf.io/weight: "10"
werf.io/weight: "-10"
```

Может быть положительным числом, отрицательным числом или нулем. Значение передается в виде строки. По умолчанию `weight` имеет значение 0. Работает только для ресурсов, не относящихся к хукам. Для хуков используйте `helm.sh/hook-weight`, логика работы которого почти такая же.

Этот параметр задает вес ресурсов, определяя порядок их развертывания. Сначала ПО «Deckhouse Platform» группирует ресурсы в соответствии с их весом, а затем последовательно развертывает их, начиная с группы с наименьшим весом. В этом случае ПО «Deckhouse Platform» не будет приступать к развертыванию следующей группы ресурсов, пока развертывание предыдущей не завершено успешно.

External dependency resource

`<any-name>.external-dependency.werf.io/resource: type[.version.group]/name`

Пример:

```
secret.external-dependency.werf.io/resource: secret/config
someapp.external-dependency.werf.io/resource: deployments.v1.apps/app
```

Задает внешнюю зависимость для ресурса. Ресурс с аннотацией будет развернут только после создания и готовности внешней зависимости.

External dependency namespace

`<any-name>.external-dependency.werf.io/namespace: name`

Указывает пространство имен для внешней зависимости, заданной соответствующей аннотацией. Префикс <any-name> должен быть таким же, как у аннотации, определяющей внешнюю зависимость.

Replicas on creation

Когда для ресурса включён HPA, использование `spec.replicas` может привести к непредсказуемому поведению, потому что каждый раз когда происходит converge для ПО «Deckhouse Platform» chart через CI/CD количество реплик ресурса будет сброшено к статически заданному в шаблонах чарта значению `spec.replicas`, даже если это значение изменил HPA в рантайме.

Одно из рекомендованных решений — совсем удалить spec.replicas из шаблонов чарта. Однако если необходимо установить начальное значение реплик при создании ресурса, можно воспользоваться аннотацией `"werf.io/replicas-on-creation"`.

`"werf.io/replicas-on-creation": "NUM"`

Задаёт число реплик, которые должны быть установлены для ресурса при его первичном создании.

`"NUM"` должно быть указано строкой (в двойных кавычках), потому что аннотации не поддерживают передачу других типов данных кроме строк, аннотации с другим типом данных будут проигнорированы.

Track termination mode

`"werf.io/track-termination-mode": WaitUntilResourceReady|NonBlocking`

Определяет условие остановки отслеживания ресурса в процессе деплоя:

- `WaitUntilResourceReady` (по умолчанию) — весь процесс деплоя будет отслеживать и ожидать готовности ресурса с данной аннотацией. Т.к. данный режим включен по умолчанию, то, по умолчанию, процесс деплоя ждет готовности всех ресурсов.
- `NonBlocking` — ресурс с данной аннотацией отслеживается только пока есть другие ресурсы, готовности которых ожидает процесс деплоя.

Используйте аннотации — `"werf.io/track-termination-mode": NonBlocking` и `"werf.io/fail-mode": IgnoreAndContinueDeployProcess`, когда описываете в релизе объект Job, который должен быть запущен в фоне и не влияет на процесс деплоя.

Используйте аннотацию `"werf.io/track-termination-mode": NonBlocking`, когда описываете в релизе объект `StatefulSet` с ручной стратегией выката (параметр `OnDelete`) и не хотите блокировать весь процесс деплоя из-за этого объекта, дожидаясь его обновления.

Fail mode

`"werf.io/fail-mode": FailWholeDeployProcessImmediately|HopeUntilEndOfDeployProcess|IgnoreAndContinueDeployProcess`

Определяет как ПО «Deckhouse Platform» будет обрабатывать ресурс в состоянии ошибки, которое возникает после превышения порога ошибок, возникающих во время отслеживания данного ресурса в процессе деплоя:

- `FailWholeDeployProcessImmediately` (по умолчанию) — в случае ошибки при деплое ресурса с данной аннотацией, весь процесс деплоя будет завершен с ошибкой.
- `HopeUntilEndOfDeployProcess` — в случае ошибки при деплое ресурса с данной аннотацией его отслеживание будет продолжаться, пока есть другие ресурсы, готовности которых ожидает процесс деплоя, либо все оставшиеся ресурсы имеют такую-же аннотацию. Если с ошибкой остался только этот ресурс или несколько ресурсов с такой-же аннотацией, то в случае сохранения ошибки весь процесс деплоя завершается с ошибкой.
- `IgnoreAndContinueDeployProcess` — ошибка при деплое ресурса с данной аннотацией не влияет на весь процесс деплоя.

Failures allowed per replica

`"werf.io/failures-allowed-per-replica": "NUMBER"`

По умолчанию, при отслеживании статуса ресурса допускается срабатывание ошибки 1 раз, прежде чем весь процесс деплоя считается ошибочным. Этот параметр влияет на поведение настройки Fail mode: определяет порог срабатывания, после которого начинает работать режим реакции на ошибки.

Ignore readiness probe failures for container

`"werf.io/ignore-readiness-probe-fails-for-CONTAINER_NAME": "TIME"`

Эта аннотация позволяет переопределить высчитываемый автоматически период, в течение которого неуспешные readiness-пробы не станут переводить ресурс в состояние ошибки, т. е. будут проигнорированы. По умолчанию период игнорирования неудачных readiness-проб автоматически вычисляется на основе конфигурации readiness-пробы. Заметим, что если в конфигурации readiness-пробы указано `failureThreshold: 1`, тогда первая же неудачная readiness-проба переведет ресурс в состояние ошибки, независимо от периода игнорирования.

Пример: `"werf.io/ignore-readiness-probe-fails-for-backend": "20s"`

No activity timeout

`werf.io/no-activity-timeout: "TIME"`

По умолчанию: `4m`

Пример:

```
werf.io/no-activity-timeout: "8m30s"
werf.io/no-activity-timeout: "90s"
```

При отсутствии новых событий и обновлений ресурса в течение `TIME` ресурс перейдет в состояние ошибки.

Log regex

`"werf.io/log-regex": RE2_REGEX`

Определяет `Re2 regex` шаблон, применяемый ко всем логам всех контейнеров всех подов ресурса с этой аннотацией. ПО «Deckhouse Platform» будет выводить только те строки лога, которые удовлетворяют regex-шаблону. По умолчанию ПО «Deckhouse Platform» выводит все строки лога.

Log regex for container

`"werf.io/log-regex-for-CONTAINER_NAME": RE2_REGEX`

Определяет `Re2 regex` шаблон, применяемый к логам контейнера с именем `CONTAINER_NAME` всех подов с данной аннотацией. ПО «Deckhouse Platform» будет выводить только те строки лога, которые удовлетворяют regex-шаблону. По умолчанию ПО «Deckhouse Platform» выводит все строки лога.

Skip logs

`"werf.io/skip-logs": "true"|"false"`

Если установлена в `"true"`, то логи всех контейнеров пода с данной аннотацией не выводятся при отслеживании. Отключено по умолчанию.

Skip logs for containers

`"werf.io/skip-logs-for-containers": CONTAINER_NAME1,CONTAINER_NAME2,CONTAINER_NAME3...`

Список (через запятую) контейнеров пода с данной аннотацией, для которых логи не выводятся при отслеживании.

Show logs only for containers

`"werf.io/show-logs-only-for-containers": CONTAINER_NAME1,CONTAINER_NAME2,CONTAINER_NAME3...`

Список (через запятую) контейнеров пода с данной аннотацией, для которых выводятся логи при отслеживании. Для контейнеров, чьи имена отсутствуют в списке, логи не выводятся. По умолчанию выводятся логи для всех контейнеров всех подов ресурса.

Show service messages

`"werf.io/show-service-messages": "true"|"false"`

Если установлена в `"true"`, то при отслеживании для ресурсов будет выводиться дополнительная отладочная информация, такая как события Kubernetes. По умолчанию, ПО «Deckhouse Platform» выводит такую отладочную информацию только в случае если ошибка ресурса приводит к ошибке всего процесса деплоя.

5 Описание параметров (настроек) безопасности

5.1 Настройка сканирования на уязвимости

Сканирование на уязвимости осуществляется с помощью модуля operator-trivy. Модуль позволяет получать информацию о проблемах в настройке кластера или отдельных объектов кластера, а также информацию об уязвимостях, найденных в используемых образах контейнеров.

Чтобы получать информацию о наличии уязвимостей, необходимо:

- включить модуль сканирования;
- обновить базы уязвимостей (см. п. 4.1.1)
- установить на пространствах имен, содержащих подлежащие сканированию поды, аннотацию security-scanning.deckhouse.io/enabled.

Для включения модуля operator-trivy выполните следующую команду (должна быть настроена конфигурация kubectl на работу с кластером):

d8 platform module enable operator-trivy

Для установки аннотации *security-scanning.deckhouse.io/enabled* на пространство имен, содержащее поды, подлежащие сканированию на уязвимости, выполните, например, следующую команду (должна быть настроена конфигурация kubectl на работу с кластером):

d8 k label namespace <ПРОСТРАНСТВО_ИМЕН> security-scanning.deckhouse.io/enabled="

Информация о найденных уязвимостях обновляется после включения модуля operator-trivy или установки аннотации на пространство имен и далее каждые 6 часов.

Просмотр отчетов о сканировании осуществляется в веб-интерфейсе Grafana в дашбордах *CIS Kubernetes Benchmark* и *Trivy Image Vulnerability Overview,* сгруппированных в папке Security (см.п.4.5):

5.1.1 Обновление базы уязвимостей

Обновление базы уязвимостей необходимо выполнять периодически. Рекомендуемая частота обновления баз – один раз в сутки.

Первичным источником базы уязвимостей является хранилище образов контейнеров по адресу registry-cse.deckhouse.ru. Обновление баз уязвимостей подразумевает копирование образов контейнеров с базой уязвимостей из первичного источника в промежуточный файл, а затем из промежуточного файла в хранилище образов контейнеров, используемое для работы ПО «Deckhouse Platform».

Для обновления базы уязвимостей, необходим лицензионный ключ (входит в поставку ПО «Deckhouse Platform»).

Для скачивания обновлений базы уязвимостей в файл, выполните следующую команду на узле, с которого производили установку или с выделенного хоста для обновления базы (укажите имя файла и лицензионный ключ):

```
d8 mirror pull \
  ${PACKAGE_DIR_PATH} \
  --source="registry-cse.deckhouse.ru/deckhouse/cse" \
  --license="${LICENSE_KEY}" \
  --gost-digest \
  --deckhouse-tag="v1.67.4" \
  --no-modules \
  --no-platform
```

Для загрузки обновлений базы уязвимостей из файла, выполните следующую команду на master-узле (укажите имя файла с базой обновлений и данные хранилища образа контейнеров):

d8 mirror push <PACKAGE_DIR_PATH> <LOCAL_REGISTRY_URL> --registry-login <LOCAL_REGISTRY_LOGIN> --registry-password <LOCAL_REGISTRY_PASSWORD>

После загрузки базы уязвимостей в хранилище образов контейнеров модуль operator-trivy обновит внутреннюю базу данных об уязвимостях (раз в 4 часа), и будет использовать обновленные данные при сканировании.

5.2 Настройка политик безопасности.

В ПО «Deckhouse Platform» предусмотрено использование следующих политик безопасности:

- *Privileged* — не ограничивающая политика с максимально широким уровнем разрешений;
- *Baseline* — минимально ограничивающая политика, которая предотвращает наиболее известные и популярные способы повышения привилегий. Позволяет использовать стандартную (минимально заданную) конфигурацию пода;
- *Restricted* — политика со значительными ограничениями. Предъявляет самые жесткие требования к подам.

Политика может быть определена для использования по умолчанию, а также для конкретных пространств имен.

При установке ПО «Deckhouse Platform» политика по умолчанию установлена как *Baseline*.

Политика по умолчанию определяется в параметре *podSecurityStandards.defaultPolicy* в ModuleConfig *admission-policy-engine* (см. п. 4.2.1).

При необходимости изменить политику на конкретном пространстве имен, на него нужно установить лейбл *security.deckhouse.io/pod-policy=<НАЗВАНИЕ_ПОЛИТИКИ>*.

Пример команды установки политики Restricted на пространство имен my-namespace (должна быть настроена конфигурация kubectl на работу с кластером):

d8 k label ns my-namespace security.deckhouse.io/pod-policy=restricted

Действие (режим работы политик), которое будет выполнено по результатам проверки ограничений политики, может стать одним из следующих:

- *Deny* — запрет действия;
- *Dryrun* — отсутствие действия. Применяется при отладке. Информацию о событии можно посмотреть в Grafana или консоли с помощью kubectl;
- *Warn* — аналогично Dryrun, но дополнительно к информации о событии будет выведена информация о том, из-за какого ограничения (constraint) был бы запрет действия, если бы вместо Warn использовался Deny.

По умолчанию, политики применяются в режиме “*Deny*”. В этом режиме поды приложений, не удовлетворяющие политикам, не могут быть запущены. Режим работы политик может быть задан как глобально для кластера, так и для каждого пространства имен отдельно.

Глобальный режим работы политик определяется в параметре *podSecurityStandards. enforcementAction* в ModuleConfig *admission-policy-engine* (см. п. 4.2.1). В случае если необходимо переопределить глобальный режим политик для конкретного пространства имен, на него нужно установить лейбл *security.deckhouse.io/pod-policy-action =<РЕЖИМ_ПОЛИТИКИ>* на соответствующем namespace. Список допустимых режимом политик состоит из: “dryrun”, “warn”, “deny”.

Пример команды установки режима *Warn* на пространство имен my-namespace (должна быть настроена конфигурация kubectl на работу с кластером):

d8 k label ns my-namespace security.deckhouse.io/pod-policy-action=warn

5.2.1 Ресурс ModuleConfig

- apiVersion: deckhouse.io/v1alpha1
- kind: ModuleConfig

- metadata:
  - name: admission-policy-engine
- spec:
  - version: 1
  - enabled: true
  - settings
    - settings.denyVulnerableImages *объект*

      Настройки trivy-провайдера.

      Trivy-провайдер запрещает

      создание Pod/Deployment/StatefulSet/DaemonSet с образами, которые имеют уязвимости в пространствах имен с

      лейблом security.deckhouse.io/trivy-provider: "".

      - settings.denyVulnerableImages.enabled *булевый*

        Включить trivy-провайдер.

      По умолчанию: false

      - settings.denyVulnerableImages.registrySecrets *массив объектов*

        Список дополнительных секретов приватных registry.
        По умолчанию для загрузки образов для сканирования используется секрет deckhouse-registry.

        По умолчанию: []

      - settings.denyVulnerableImages.registrySecrets.name *строка*

        ОБЯЗАТЕЛЬНЫЙ ПАРАМЕТР

      - denyVulnerableImages.registrySecrets.namespace *строка*

        ОБЯЗАТЕЛЬНЫЙ ПАРАМЕТР

    - podSecurityStandards *объект*

      Настройки политик Pod Security Standards (PSS).

      - podSecurityStandards.defaultPolicy *строка*

        Определяет политику Pod Security Standards по умолчанию для всех несистемных пространств имен:

- Privileged — политика без ограничений. Данная политика допускает эскалацию привилегий;
- Baseline — политика с минимальными ограничениями, ограничивающая использование эскалаций привилегий;
- Restricted — политика с максимальными ограничениями, соответствующая актуальным рекомендациям по безопасному запуску приложений в кластере.

По умолчанию:

- Baseline — при первичной установке Deckhouse версии v1.64 и выше;
- Privilged — при первичной установке Deckhouse версии ниже v1.55 (обновление Deckhouse в кластере на версию v1.64 и выше **не меняет** политику по умолчанию на Baseline).

Допустимые значения: Privileged, Baseline, Restricted

- podSecurityStandards.enforcementAction *строка*

  Действие, которое будет выполнено по результатам проверки ограничений:

  - Deny — запрет;
  - Dryrun — отсутствие действия. Применяется при отладке. Информацию о событии можно посмотреть в Grafana или консоли с помощью kubectl;
  - Warn — аналогично Dryrun, но дополнительно к информации о событии будет выведена информация о том, из-за какого ограничения (constraint) был бы запрет действия, если бы вместо Warn использовался Deny.

  По умолчанию: "Deny"

  Допустимые значения: Warn, Deny, Dryrun

- podSecurityStandards.policies *объект*

  Определяет дополнительные параметры политик

  - podSecurityStandards.policies.hostPorts *объект*

Настройки ограничения HostPort.

- podSecurityStandards.policies.hostPorts.knownRanges *массив объектов*

  Список диапазонов портов, которые будут разрешены в привязке hostPort.

  - podSecurityStandards.policies.hostPorts.knownRanges.max
  - podSecurityStandards.policies.hostPorts.knownRanges.min

5.3 Настройка уведомлений о событиях безопасности на почту

Чтобы настроить уведомлений о событиях безопасности на почту, необходимо выполнить следующую команду:

d8 k apply -f email.yaml

Пример файла email.yaml:

```
apiVersion: deckhouse.io/v1alpha1
kind: CustomAlertmanager
spec:
  internal:
    receivers:
    - emailConfigs:
      - authPassword:
          key: password
          name: alertmanager-email
        authUsername: stand@mg.flant.dev
        from: stand@mg.flant.dev
        sendResolved: true
        smarthost: smtp.mailgun.org:25
        to: example@flant.com
      name: email
    route:
      groupBy:
      - job
      groupInterval: 5m
```

```
    groupWait: 30s
    receiver: email
    repeatInterval: 12h
  type: Internal
```

5.4 Настройка доступа к журналам событий безопасности

Чтобы настроить доступ пользователям к Grafana и Prometheus, необходимо выполнить следующую команду:

d8 k apply -f prometheus.yaml

Пример файла prometheus.yaml:

```
apiVersion: deckhouse.io/v1alpha1
kind: ModuleConfig
metadata:
  name: prometheus
spec:
  enabled: true
  settings:
    auth:
      allowedUserGroups:
      - administrator
      - security
  version: 2
```

- auth

Опции, связанные с аутентификацией или с авторизацией в приложении.

- auth.allowedUserGroups

Массив групп, пользователям которых позволен доступ в Grafana и Prometheus.

5.5 Просмотр журналов событий безопасности

Просмотр журналов событий безопасности осуществляется в веб-интерфейсе Grafana. Необходимые дашборды сгруппированы в папке Security:

- *Admission policy engine.* Содержит информацию, связанную с работой политик безопасности. В том числе: количество событий запрета выполнения действий из-за нарушения политики безопасности; разбивку запретов выполнения действий по типу запрета; журнал событий.

  Журнал событий безопасности, связанных с политиками безопасности, находится в окне OPA Violations.

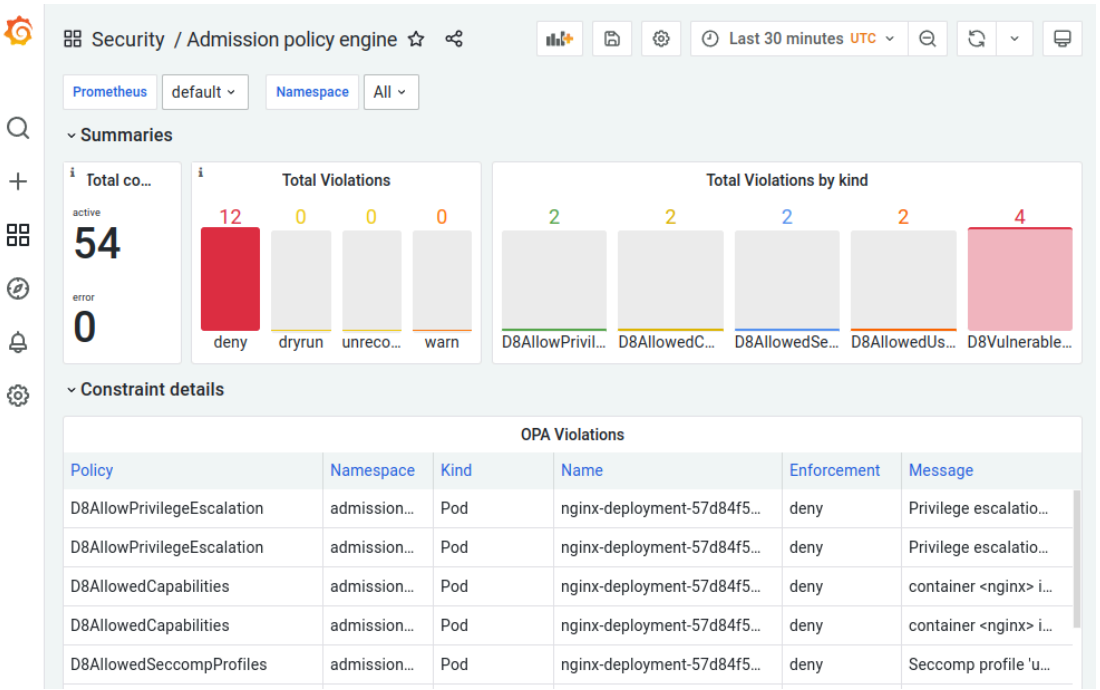

Рис. 5.5.1. Пример дашборда Admission policy engine

- *CIS Kubernetes Benchmark.* Дашборд с информацией о результатах работы сканера проверок конфигурации кластера на соответствие принятым подходам (лучшим практикам). Содержит сводную информацию о результатах проверки, без возможности детализации. Дашборд доступен при включенном модуле operator-trivy (см. п.4.1.).

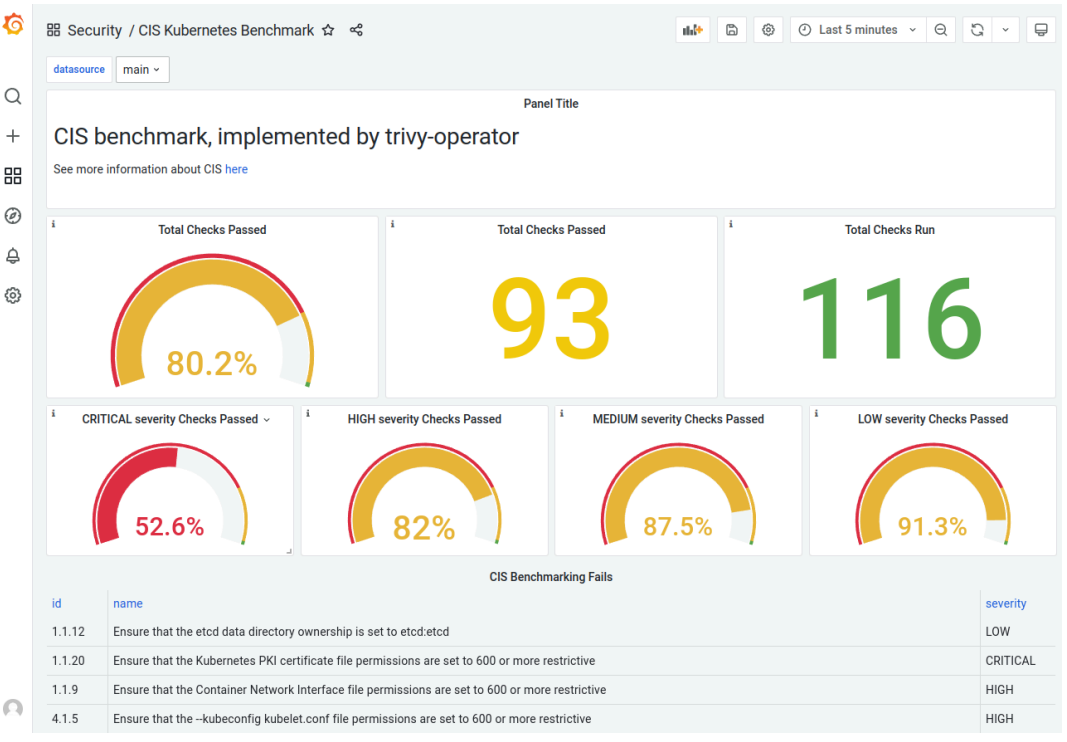

Рис. 5.5.3. Пример дашборда CIS Kubernetes Benchmark

- *Kubernetes audit logs.* Журнал регистрации обращений к API-серверу. Содержит записи о всех обращениях к API-серверу кластера в JSON-формате.

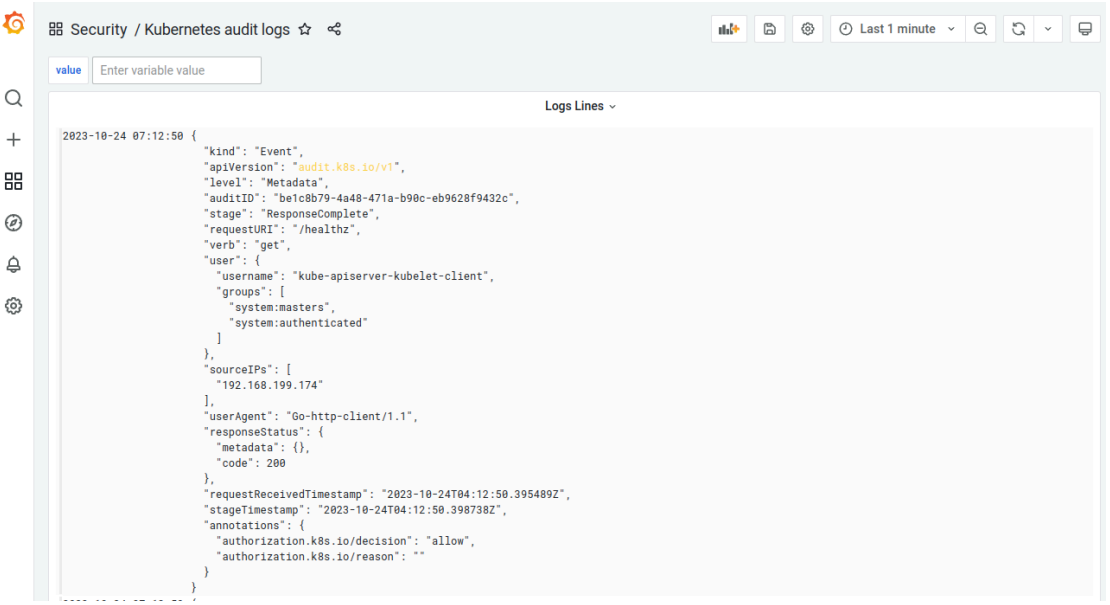

Рис. 5.5.4. Пример дашборда Kubernetes audit logs

- *Runtime audit engine logs.* Журнал регистрации событий безопасности аудита работы ядра Linux и API-сервера кластера.

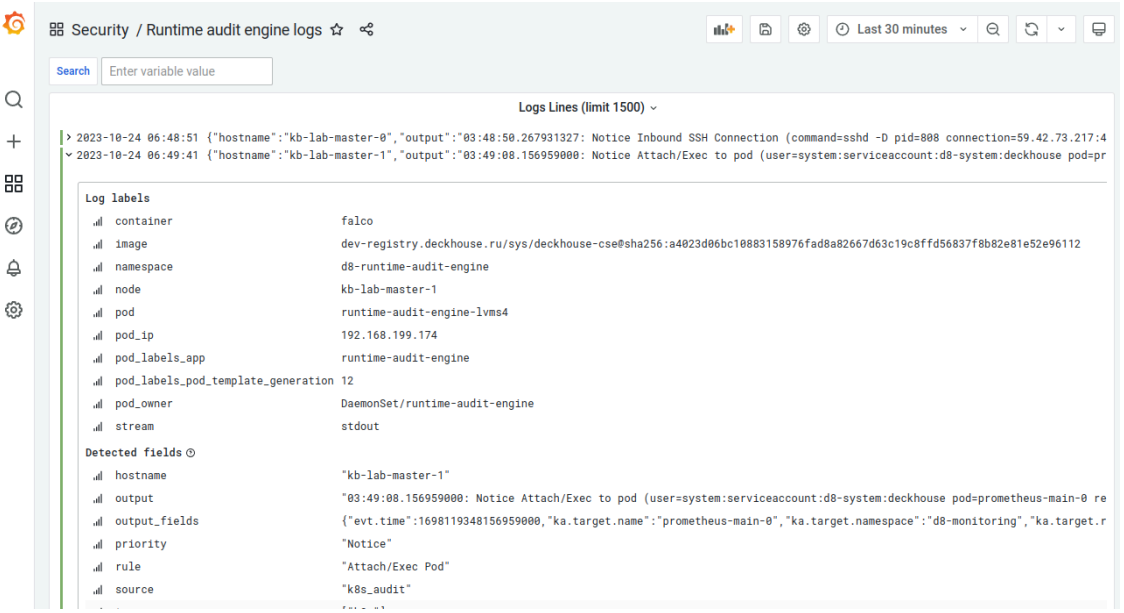

Рис. 5.5.5. Пример дашборда Runtime audit engine logs

- *Trivy Image Vulnerability Overview.* Дашборд со сводной и детализированной информацией о сканировании образов контейнеров подов в пространствах имен, отмеченных аннотацией *security-scanning.deckhouse.io/enabled* (см. п.4.1).

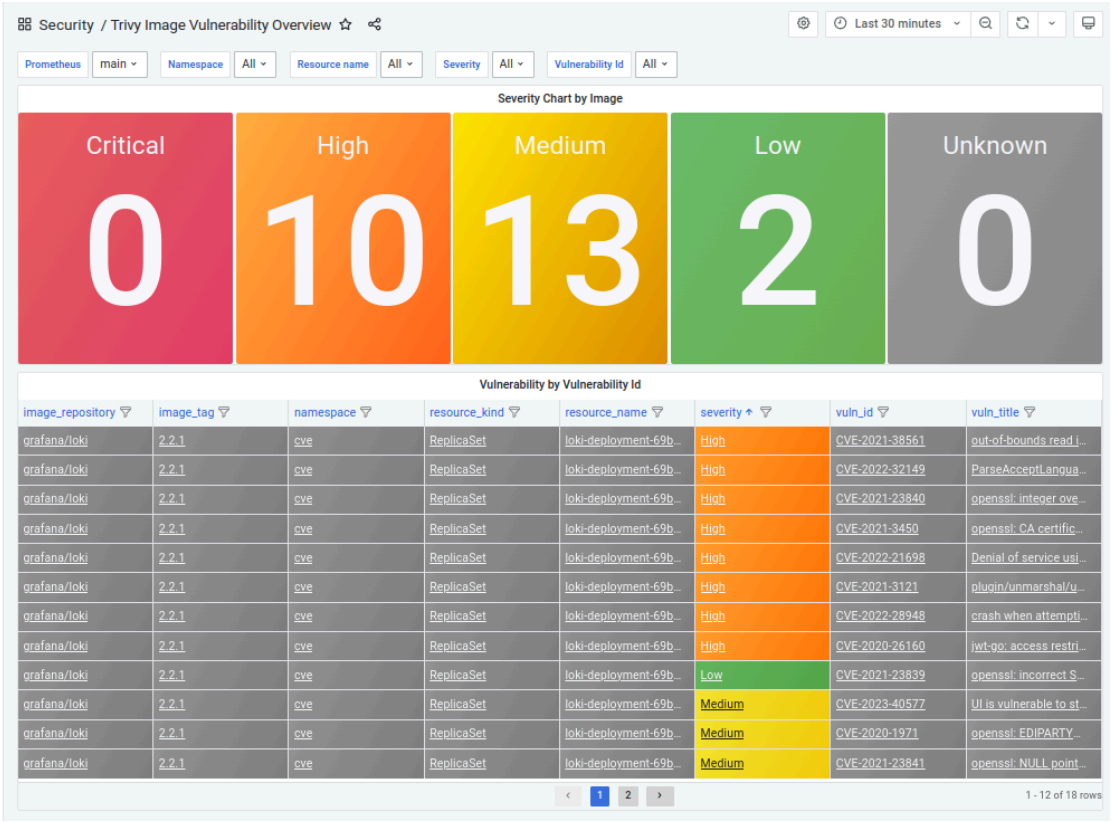

Рис. 5.5.6. Пример дашборда Trivy Image Vulnerability Overview

5.6 Контроль целостности

Для проверки целостности образа используется контрольная сумма, рассчитанная по алгоритму Стрибог (ГОСТ Р 34.11-2012).

Должен быть включен модуль gost-integrity-controller, если модуль не включен, то для включения модуля используйте команду:

d8 platform module enable gost-integrity-controller

Проверяем, что под `gost-digest-webhook` запустился:

d8 k get po -A |grep gost

Должно выйти примерно следующее:

d8-system gost-digest-webhook-56f59c48bb-b5njj 2/2 Running 0 160m

Чтобы устанавливаемые образы проверялись, необходимо добавить метку gost-integrity-controller.deckhouse.io/gost-digest-validation-enabled: true в пространство имен кластера, где необходимо производить контроль целостности образа.

Например командой:

d8 k label ns test-gost gost-integrity-controller.deckhouse.io/gost-digest-validation-enabled=true

Если в ходе проверки контрольная сумма образа окажется некорректной, установка образа будет отклонена, и вы получите соответствующее сообщение.

Для расчета контрольной суммы берется список контрольных сумм слоев образа. Список сортируется в порядке возрастания и склеивается в одну строку. Затем производится расчет контрольной суммы от этой строки по алгоритму Стрибог (ГОСТ Р 34.11-2012).

Пример расчета контрольной суммы образа nginx:1.25.2:

Контрольные суммы слоев отсортированные в порядке возрастания

[
"sha256:27e923fb52d31d7e3bdade76ab9a8056f94dd4bc89179d1c242c0e58592b4d5c",
"sha256:360eba32fa65016e0d558c6af176db31a202e9a6071666f9b629cb8ba6ccedf0",
"sha256:72de7d1ce3a476d2652e24f098d571a6796524d64fb34602a90631ed71c4f7ce",
"sha256:907d1bb4e9312e4bfeabf4115ef8592c77c3ddabcfddb0e6250f90ca1df414fe",
"sha256:94f34d60e454ca21cf8e5b6ca1f401fcb2583d09281acb1b0de872dba2d36f34",
"sha256:c5903f3678a7dec453012f84a7d04f6407129240f12a8ebc2cb7df4a06a08c4f",
"sha256:e42dcfe1730ba17b27138ea21c0ab43785e4fdbea1ee753a1f70923a9c0cc9b8"
]

Склеенная строка из контрольных сумм

"sha256:27e923fb52d31d7e3bdade76ab9a8056f94dd4bc89179d1c242c0e58592b4d5c
sha256:360eba32fa65016e0d558c6af176db31a202e9a6071666f9b629cb8ba6ccedf0
sha256:72de7d1ce3a476d2652e24f098d571a6796524d64fb34602a90631ed71c4f7ce
sha256:907d1bb4e9312e4bfeabf4115ef8592c77c3ddabcfddb0e6250f90ca1df414fe
sha256:94f34d60e454ca21cf8e5b6ca1f401fcb2583d09281acb1b0de872dba2d36f34
sha256:c5903f3678a7dec453012f84a7d04f6407129240f12a8ebc2cb7df4a06a08c4f
sha256:e42dcfe1730ba17b27138ea21c0ab43785e4fdbea1ee753a1f70923a9c0cc9b8"

Контрольная сумма образа

2f538c22adbdb2ca8749cdafc27e94baed8645c69d4f0745fc8889f0e1f5a3f9

Для подписания образов в вашем репозитории необходима утилита imagedigest, которую можно скопировать из дистрибутива ПО «Deckhouse Platform Certified Security Edition». Из образа install. Для этого необходимо перейти в каталог с дистрибутивом, распаковать архив deckhouse-cse-1.67.4.tar с помощью следующий команды:

tar -xvf /distr/deckhouse-cse-1.67.4.tar -C /distr/deckhouse-cse-1.67.4

Перейти в каталог deckhouse-cse-1.67.4 распаковать архив imagedigest с помощью следующей команды:

tar -xvf ./install/blobs/sha256/cdd1432f175c5fc40cb7175230271af6b222cc8974443218fc12fa4ea581246e

Из каталога ./usr/bin/ скопировать бинарный файл imagedigest к себе на ЭВМ.

С помощью утилиты imagedigest выполняется расчет контрольной суммы образа, добавление контрольной суммы в метаданные образа и проверка контрольной суммы образа.

imagedigest calculate <имя_образа> - расчет контрольной суммы образа.

imagedigest add <имя_образа> - добавление контрольной суммы в метаданные образа.

imagedigest validate <имя_образа> - проверка контрольной суммы.

Для каждого образа администратор должен фиксировать в журнал идентификатор пользователя, подписавшего образ, и вычисленную контрольную сумму образа.

6 Действия по реализации функций безопасности среды функционирования средства

В среде функционирования ПО «Deckhouse Platform» должны быть реализованы следующие функции безопасности:

- физическая защита;
- доверенная загрузка ПО «Deckhouse Platform»;
- обеспечение условий безопасного функционирования ПО «Deckhouse Platform»;
- обеспечение доверенного маршрута;
- обеспечение доверенного канала.

Для реализации функций безопасности среды функционирования ПО «Deckhouse Platform» должны выполняться следующие действия:

- необходимо настроить SSH-доступ по ключу. Для этого необходимо подготовить и скопировать публичный ключ в каталог `"$HOME/.ssh/`
- отключить маршруты по умолчанию (default route) и оставить только маршрут (route) к registry и к bastion host для подключения по ssh. Для этого в файле /etc/rc.local прописать параметры, пример которых представлен ниже:

```
# tf.bastion
ip.ro add 95.217.68.252/32 vim 192.168.199.1
# registry
ip.ro add 5.182.5.140/32 vim 192.168.199.1
# cluster networks
ip.ro add 10.111.0.0/16 vim 192.168.199.1
ip.ro add 10.222.0.0/16 vim 192.168.199.1
# remove all routes
ip.ro add 5.182.5.140/32 vim 192.168.199.1
ip ro del default
```

- для осуществления безопасной настройки ОС на узлах кластера после установки ПО «Deckhouse Platform» необходимо применить в кластере YAML-файл, приведенный в электронном приложении к настоящему документу, каталог «Электронные приложения» - «Deckhouse Platform. Руководство администратора» - ngc.yaml. Для этого выполните следующую команду:

  d8 k create ngc.yaml

Данный файл подготовлен в соответствии с методическим документом ФСТЭК России от 25 декабря 2022 г. «Рекомендации по безопасной настройке операционных систем Linux» (https://fstec.ru/dokumenty/vse-dokumenty/spetsialnye-normativnye-dokumenty/metodicheskij-dokument-ot-25-dekabrya-2022-g)

- необходимо регулярное обновление всех сред функционирования ПО «Deckhouse Platform» до актуальных версий с применением всех необходимых патчей безопасности с официальных сайтов разработчиков сред функционирования;
- установка, конфигурирование и управление ПО «Deckhouse Platform» должно осуществляться в соответствии с эксплуатационной документацией;
- доступ к объектам доступа ПО «Deckhouse Platform» должен осуществляться с учетом минимально необходимых прав и привилегий в соответствии с ролевой моделью ПО «Deckhouse Platform»;
- должна быть обеспечена физическая сохранность серверной платформы с установленным ПО «Deckhouse Platform» и исключение возможности физического доступа к ней посторонних лиц;
- ПО «Deckhouse Platform» должно использоваться только на совместимых с ним аппаратных мощностях и средствах;
- предоставление пользователям прав доступа к объектам доступа информационной системы обеспечивается, основываясь на задачах, решаемых пользователями в ПО «Deckhouse Platform» и взаимодействующими с ней информационными системами;
- каналы передачи данных ПО «Deckhouse Platform» должны быть либо расположены в пределах контролируемой зоны и защищены с использованием организационно-технических мер, либо, в случае их выхода за пределы контролируемой зоны, должны быть защищены путем применения средств криптографической защиты информации, сертифицированных в системе сертификации ФСБ России.
- запрещается использовать функции фильтрации модуля istio. Модуль istio можно использовать только для защиты внутреннего трафика (mTLS).

7 Модули и параметры, отвечающие за реализацию функций безопасности

Список модулей, выключение которых не позволяет ПО «Deckhouse Platform» выполнять заявленные функции безопасности. Выключение указанных модулей недопустимо:

- admission-policy-engine
- cni-cilium
- control-plane-manager
- deckhouse
- ingress-nginx
- kube-dns
- log-shipper
- loki
- node-manager
- operator-prometheus
- operator-trivy
- priority-class
- prometheus
- registrypackages
- registry-packages-proxy
- runtime-audit-engine
- user-authn
- user-authz
- gost-integrity-controller

Параметры модулей или ресурсы, изменение которых влияет на реализацию ПО «Deckhouse Platform» заявленных функций безопасности.

Модуль	Параметры модуля или ресурс (объект)	Функция безопасности	Пояснения
admission-policy-engine	- denyVulnerableImages.enabled - podSecurityStandards.enforcementAction	- запрещать создание образов контейнеров, содержащих известные уязвимости критического и	- параметр podSecurityStandards.enforcementAction должен быть установлен в Deny (значение по умолчанию).

		высокого уровня опасности	- параметр denyVulnerableImages.enabled должен быть установлен в true.
	- podSecurityStandards.defaultPolicy - podSecurityStandards.enforcementAction	- ограничение прав прикладного программного обеспечения, выполняемого внутри контейнера, на использование периферийных устройств, устройств хранения данных и съемных машинных носителей информации (блочных устройств), входящих в состав информационной (автоматизированной) системы - ограничение прав прикладного программного обеспечения, выполняемого внутри контейнера, на использование вычислительных ресурсов (оперативной памяти, операций ввода-вывода за период времени) хостовой операционной системы - монтирование корневой файловой системы хостовой операционной системы в режиме «только для чтения»	- параметр podSecurityStandards.enforcementAction должен быть установлен в Deny (значение по умолчанию). - параметр podSecurityStandards.defaultPolicy НЕ должен быть установлен в Privileged.
loki	- storageClass - storeSystemLogs	- контролировать целостность сведений о событиях безопасности самостоятельно - - осуществлять сбор и хранение записей в журнале событий безопасности, которые позволяют определить, когда и какие действия происходили	- параметр storageClass должен быть определен, если не определен глобальный параметр modules.storageClass. - параметр storeSystemLogs должен быть установлен в true.

operator-trivy	- severities	- выявление известных уязвимостей при создании, установке образа контейнера в информационной (автоматизированной) системе и хранении образов контейнеров - - оповещать о выявленных уязвимостях в образах контейнеров разработчика образов контейнеров и администратора безопасности информационной (автоматизированной) системы	Параметр severities должен содержать набор всех уровней критичности - UNKNOWN, LOW, MEDIUM, HIGH, CRITICAL (значение по умолчанию).
runtime-audit-engine, prometheus	CustomPrometheusRules	- контроль целостности образов контейнеров и исполняемых файлов контейнеров - контролировать целостность параметров настройки ПО «Deckhouse Platform Certified Security Edition»	Должен быть создан объект CustomPrometheusRules, настраивающий отправку информации о событии аудита в систему мониторинга
gost-integrity-controller		- контролировать целостность образов контейнеров и параметров настройки ПО «Deckhouse Platform Certified Security Edition» при установке образа контейнера в информационной (автоматизированной) системе и далее периодически за счет применения цифровой подписи самостоятельно	На объект Namespace, необходимого пространства имен, необходимо установить метку ***gost-integrity-controller.deckhouse.io/gost-digest-validation-enabled: true***, чтобы выполнялась функция безопасности

Лист регистрации изменений

Номер изменения	Номера страниц				Всего страниц в документе	Наименоdвание и номер документа, вводящего изменения	Подпись, Ф.И.О. внесшего изменения в данный экземпляр	Дата внесения изменения в данный экземпляр	Дата введения изменения
	измененных	замененных	новых	аннулиро-ванных					
1	2	3	4	5	6	7	8	9	10