# UNIVERSAL HELM CHART

**Karin I.E.[1], Kriuchkov A.Yu.[2] (UAE)**

*[1]Karin Iliia Eduardovich - Master of Information Systems in Economics and Management, DevOps engineer;*
*[2]Kriuchkov Aleksandr Yurievich – master of radio communications, radio broadcasting and television, cloud engineer;*
*INFRASTRUCTURE DEPARTMENT,*
*INVENT INC.*
*DUBAI, UAE*

# УНИВЕРСАЛЬНЫЙ HELM CHART

**Карин И.Э.[1], Крючков А.Ю.[2] (ОАЭ)**

*[1]Карин Илия Эдуардович - магистр информационных систем в экономике и менеджменте, DevOps-инженер;*
*[2]Крючков Александр Юрьевич – магистр радиосвязи, радиовещания и телевидения, облачный инженер;*
*отдел инфраструктуры,*
*INVENT INC,*
*г. Дубай, ОАЭ*

***Аннотация:*** *в статье описано использование авторского метода под названием Universal Helm Chart Technology на базе инструмента HELM и показан пример такого автоматизированного развертывания в Kubernetes.*
***Ключевые слова:*** *helm, bash, облако, информация и технологии, докер, kubernetes, devops, разработка программного обеспечения.*

## Introduction.

Deploying multiple applications or microservices in a complex Kubernetes environment can be a challenging task. The need to manage dependencies, configurations, and versioning across different components can quickly become overwhelming. That's where HELM comes in. In this article, we will explore the power of HELM technology and how it simplifies the process of deploying and managing applications in Kubernetes. Whether you are a developer, an operator, or a system administrator, understanding the pros and cons of HELM will empower you to efficiently package, deploy, and upgrade your applications, while maintaining consistency and scalability. So, let's embark on a journey to discover the capabilities and benefits of HELM, and how it can revolutionize your application deployment workflows.

### HELM what, why, and how?

### What is HELM?

Helm is a package manager for Kubernetes that simplifies the deployment of applications and services onto Kubernetes clusters.

Key points:

- Manage Kubernetes Applications: HELM serves as a comprehensive tool for managing Kubernetes applications. It utilizes HELM Charts, which act as a blueprint for defining, installing, and upgrading even the most complex applications within a Kubernetes environment.
- HELM Charts: These charts are incredibly versatile and user-friendly. They allow developers to create, version, share, and publish application configurations easily. HELM Charts describe applications comprehensively, enabling repeatable installations and serving as a single point of authority for application management.
- Easy Updates: HELM simplifies the process of updating applications by providing in-place upgrades and custom hooks. This eliminates the pain points associated with managing updates, ensuring seamless transitions to newer versions of applications.
- Simple Sharing: Charts are easy to version, share, and host on public or private servers.
- Rollbacks: HELM offers the ability to roll back to a previous version of a release effortlessly. This feature ensures that if any issues arise after an upgrade, the system can quickly revert to a known, stable state.

### Why do we need it?

To answer this question, let's consider the process of deploying an application to Kubernetes.

For instance, deploying a simple Nginx application to Kubernetes requires creating multiple manifest files, such as *deployment.yaml*, *service.yaml*, and *ingress.yaml* (to make the application accessible from the internet). While this approach works, it can become inconvenient and error-prone, especially when dealing with complex applications or managing multiple deployments.

Example:
*deployment.yaml*

```yaml
# Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-service
  labels:
    app.kubernetes.io/name: nginx-service
spec:
  replicas: 2
  selector:
    matchLabels:
      app.kubernetes.io/name: nginx-service
  template:
    metadata:
      labels:
        app.kubernetes.io/name: nginx-service
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        imagePullPolicy: Always
        ports:
        - name: http
          containerPort: 80
```

*service.yaml*

```yaml
# Service
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  selector:
    app.kubernetes.io/name: nginx-service
```

*ingress.yaml*

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
 rules:
  - host: example.com
    http:
      paths:
        - path: /
          pathType: Prefix
          backend:
            service:
              name: nginx-service
              port:
                number: 80
```

Looks simple, but let's say we need to deploy a thousand Nginx applications. At some point, we will need to update them or add additional information to the deployment.

It will lead us to an infinite operations cycle with extreme error possibilities and an impossible process of change control.

Imagine manually modifying each deployment manifest or YAML file for every Nginx application, ensuring consistency across all deployments, and tracking the changes made. The risk of human error increases exponentially, and it becomes challenging to maintain an organized and controlled process.

**What does it mean to deploy an application to the kubernetes cluster?**

It involves the process of a Kubernetes (k8s) administrator opening the CLI, connecting to the cluster, and logging into the cluster if Role-Based Access Control (RBAC) is enabled. Then, for each manifest file created, the administrator needs to use the 'kubectl apply' command. In real-world scenarios, the Kubernetes cluster is often hidden behind a VPN or a bastion/jump host, making it challenging for the administrator to easily pass files and manifests into the restricted area.

```
kubectl apply -f <filename.yaml>
```

For simple applications like Nginx, typically three manifests are sufficient. However, for more complex applications such as databases, business applications, or secure API gateways, deploying six or more manifests per application may be necessary to ensure proper functionality.

Examples of different manifests that could exist in k8s applications:

**Deployment**: Manages the deployment and scaling of a set of identical pods.

**StatefulSet**: Manages the deployment and scaling of stateful applications, providing guarantees about the ordering and uniqueness of pod creation.

**DaemonSet**: Ensures that a copy of a specific pod is running on each node in the cluster, typically used for cluster-wide tasks like log collection or monitoring.

**Job**: Runs a single task to completion, such as a batch job or a one-time data migration.

**CronJob**: Runs a job on a regular schedule, similar to a cron job in traditional Unix-like systems.

**Pod**: Represents a single instance of a running process in the cluster.

**ReplicaSet**: Ensures a specified number of replicas of a pod are running at any given time, allowing for scaling and self-healing.

**Service**: Provides a stable network endpoint to access a set of pods, enabling load balancing and service discovery.

**Ingress**: Exposes HTTP and HTTPS routes from outside the cluster to services within the cluster, performing routing, SSL termination, and load balancing.

**ConfigMap**: Stores non-sensitive configuration data as key-value pairs, which can be consumed by pods as environment variables or mounted as files.

**Secret**: Stores sensitive information such as passwords, API keys, or TLS certificates, which can be consumed by pods as environment variables or mounted as files.

**PersistentVolumeClaim (PVC)**: Requests a specific amount of storage resources from a storage class, used by pods to dynamically provision and attach persistent storage volumes.

**PersistentVolume (PV)**: Represents a physical storage resource in the cluster, which can be dynamically provisioned by a storage class or manually created.

**Namespace**: Provides a way to divide cluster resources into virtual clusters, helping isolate resources and manage access control.

**ServiceAccount**: Provides an identity for processes running in a pod, granting permissions to interact with the Kubernetes API or other resources.

**Role and RoleBinding**: Defines permissions (RBAC) for a set of resources within a namespace.

**ClusterRole and ClusterRoleBinding**: Defines permissions (RBAC) for a set of resources across the entire cluster.

**NetworkPolicy**: Defines rules for network traffic within a cluster, allowing or denying communication between pods based on specified criteria.

**Volume**: Defines a directory that can be mounted by a container, providing storage for the pod.

**Custom Resource Definition (CRD)**: Defines custom resource types in Kubernetes, allowing users to extend the Kubernetes API with their own resources and controllers.

Now, let's talk about HELM. In basic terms, HELM is a tool that simplifies the templating of Kubernetes manifests. It provides a convenient way to manage dependencies and customizable fields by utilizing variables defined in descriptive files called *values.yaml*. With HELM, all the manifests are converted into Helm templates, making it easier to configure and deploy applications.

**HELM templates and file structure.**

The simple file structure for HELM will be similar to this:

```
├── Chart.yaml
├── deploy_script.sh
├── templates
│   ├── 01-pull-secret.yaml
│   ├── 02-secrets_and_secrets-volumes.yaml
│   ├── 03-service.yaml
│   ├── 04-deployment.yaml
│   ├── 05-configmap-volumes.yaml
│   ├── 06-job.yaml
│   ├── 07-claims.yaml
│   ├── 10-rbac.yaml
│   ├── 20-ingress.yaml
└── values-nginx_1.yaml
└── values-nginx_2.yaml
└── values-nginx_3.yaml
└── values-nginx_4.yaml
└── values-nginx_N.yaml
```

The Chart.yaml file serves as a metadata file for a Helm chart. It provides essential information and configuration options about the chart, allowing users to identify and manage the chart effectively. The Chart.yaml file is located at the root directory of a Helm chart and follows a specific structure defined by Helm.

In our example it will look like this:

*Chart.yaml*

```
apiVersion: v2
name: nginx-chart
version: 0.1.0
```

Deployment.yaml now will be located in the templates subfolder and will look like this:

*deployment.yaml*

```
# Deployment
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: {{ .Values.name }}
  labels:
    app.kubernetes.io/name: {{ .Values.name }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app.kubernetes.io/name: {{ .Values.name }}
  template:
    metadata:
      labels:
        app.kubernetes.io/name: {{ .Values.name }}
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          ports:
            - name: http
              containerPort: 80
```

service.yaml

```
# Service
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.name }}
spec:
  type: {{ .Values.service.type }}
  ports:
    - protocol: TCP
      port: {{ .Values.service.port }}
      targetPort: 80
  selector:
    app.kubernetes.io/name: {{ .Values.name }}
```

ingress.yaml

```
{{ if eq .Values.ingress.enabled true }}
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: {{ .Values.name }}
  annotations:
{{- range $key, $value := .Values.ingress.annotations }}
    {{ $key }}: {{ $value | quote }}
{{- end }}
spec:
  rules:
    - host: {{ .Values.ingress.hosts.host | quote }}
      http:
        paths:
          - path: {{ .Values.ingress.hosts.path | quote }}
            pathType: Prefix
            backend:
              service:
                name: {{ .Values.name }}
                port:
                  number: {{ .Values.service.port }}
{{- end }}
```

And new file *values.yaml* which will be the source of real values for HELM chart-based manifests.
values.yaml

```
name: nginx-service
image:
  repository: nginx
  tag: 1.14.2
  imagePullPolicy: Always

replicaCount: 2

service:
  type: LoadBalancer
  port: 80

ingress:
  enabled: true
  annotations:
    kubernetes.io/ingress.class: nginx
  hosts:
    - host: example.com
      path: /
```

And now if we need to deploy multiple nginx applications we can just create different values.yaml changing needed parameters like name and port.
values-nginx_1.yaml

```
name: nginx-service_1
image:
  repository: nginx
  tag: 1.14.2
  imagePullPolicy: Always

replicaCount: 2

service:
  type: LoadBalancer
  port: 8081

ingress:
  enabled: true
  annotations:
    kubernetes.io/ingress.class: nginx
  hosts:
    -                                    host:                                  example1.com
      path: /
```

And install them one by one with the commands:

```
helm install my-nginx-release-name --values values.yaml
helm install my-nginx-release-name --values values-nginx_1.yaml
helm install my-nginx-release-name --values values-nginx_N.yaml
```

Great victory!

However, is there a way to enhance the process beyond deploying and managing thousands of files with individual commands?

Sure…

**UHC - Universal HELM chart.**

The main idea and principle were taken from the DRY coding methodology.

**DRY it now!**

DRY (Don't Repeat Yourself) - is a software development principle that promotes code reuse and reduces duplication. The DRY methodology emphasizes the importance of avoiding unnecessary duplication of information or logic within a system.

The main idea behind DRY is to have a single, authoritative source of truth for each piece of knowledge or logic in a software project. When applying DRY, if you need to make a change to that knowledge or logic, you only need to make it in one place, and it automatically propagates throughout the system.

Here are some key principles associated with the DRY methodology:

Single Source of Truth: Each piece of knowledge or logic should have a single, unambiguous representation within the system. Duplication should be avoided, as it can introduce inconsistencies, increase maintenance efforts, and make the codebase harder to understand.

Code Reuse: Instead of duplicating code or information, reusable components or abstractions should be created. These components can be shared and used across different parts of the system, reducing redundancy and promoting consistency.

Modularity: Breaking down a system into smaller, modular components can help achieve code reuse and encapsulation. Each module should have a clearly defined responsibility and should be independent of other modules as much as possible.

Separation of Concerns: Different concerns (e.g., business logic, presentation, data access) should be separated into distinct modules or layers. This separation improves maintainability and allows for easier modification or replacement of individual components.

Automated Refactoring: Automated tools and techniques, such as refactoring tools or continuous integration pipelines, can be used to identify and eliminate duplication, ensuring adherence to the DRY principle.

By following the DRY methodology, developers can create more maintainable, scalable, and robust software systems. It reduces the likelihood of introducing bugs due to inconsistent or duplicated code and promotes a more efficient development process by minimizing the effort required for making changes and adding new features.

So why repeat values.yaml files when we can join them into one.

**UHC - values.yaml structure**

We can consolidate all the values into a single file for easier parsing in the future.

We recommend using the following structure as a starting point, which can be adjusted to meet specific requirements.

```
# First level. Common variables like namespace.
namespace: example
imagePullSecrets:
  - name: gitlab-registry
# Announce group block for parsing script
groups:
```

```
# Announce blocks with group names if we need to separate different types of applications
  main:
# Inside the group level we are announcing common variables for this group.
    strategyType: RollingUpdate
    repository: nginx
    imagePullPolicy: Always
    service:
      type: LoadBalancer
    image: nginx
    ingress:
      enabled:                                                                                                    true
      annotations:
        kubernetes.io/ingress.class: nginx
# Inside the apps level we are announcing application names with specific variables, we can provide custom
variables or override group vars.
      apps:
          nginx-service_1:
            tag: 1.14.2
            service:
              port: 8081
            ingress:
              hosts:
                -                                     host:                                    example1.com
                  path: /
          nginx-service_2:
            tag: 1.11.2
            service:
              port: 8082
            ingress:
              hosts:
                -                                     host:                                    example2.com
                  path: /
          nginx-service_3:
            tag: 1.13.1
            service:
              port: 8083
            ingress:
              enabled:                                                                                    false

  secondary:
          apache-service_1:
            repository: gitlab.local.org
            image: httpd
            tag: 2.4.58
  jobs:
          job-service_1:
            repository: gitlab.external.org
            image: busybox
            tag: 1.35.0
# etc
```

As you can see, we have significantly reduced the amount of code while describing multiple applications. Now, when it comes to deployment, HELM alone may not be able to parse it. However, a simple bash script can handle the task effectively. Alternatively, we could utilize GitOps tools like ArgoCD or Spinnaker, but for the purpose of our example, a bash script will do the job.

**Bash, the one.**

```
#!/bin/bash
set -e

############################################################
###                                                      ###
###                 Decoration by color                  ###
###                                                      ###
############################################################
PURPLE='\033[0;35m'
RED='\033[0;31m'
NoColor='\033[0m'

############################################################
###                                                      ###
###       The deploy function accepts one argument.      ###
###       first argument=release_name/application_name   ###
###                                                      ###
############################################################
VALUES_FILE="$1.yaml"
deploy() {
  echo -e "${PURPLE} ### Executing deploy function: deploying application $1 ### ${NoColor}"
  echo -e "${PURPLE} # helm upgrade --install $1 ./app --history-max 3 --namespace ${NAMESPACE} --create-namespace --timeout 2m0s --atomic --values $VALUES_FILE ${NoColor}"
  helm upgrade \
    --install $1 ./app \
    --history-max 3 \
    --namespace $NAMESPACE \
    --create-namespace \
    --timeout 2m0s \
    --atomic \
    --values $VALUES_FILE
  echo -e "\n\n"
}

############################################################
###                                                      ###
###          yq/jq function to parse YAML files          ###
###            and reading the release name,             ###
### then use the deploy function with each release name ###
###                                                      ###
############################################################
group_deploy() {
  echo -e "${PURPLE} Parsing $VALUES_FILE to deploy $1 application group! ${NoColor}"
  enabled_group=$(yq eval "$VALUES_FILE" -o=json | jq -r .groups."$1".enabled)
  if [[ $enabled_group == "true" ]]; then
    for DEPLOY_SERVICE_NAME in $(yq eval "$VALUES_FILE" -o=json | jq -r .groups."$1".apps | jq -r "keys_unsorted" | jq -r @sh | tr -d \'); do
      deploy "$DEPLOY_SERVICE_NAME"
    done
  elif [[ $enabled_group == "false" ]]; then
    echo -e "${PURPLE} Group $1 disabled for deploy! ${NoColor}"
  else
    echo -e "${PURPLE}Something wrong, please check your YAML file $VALUES_FILE! ${NoColor}"
    fi
}

###########################################################
###                                                     ###
### Execute group deploy func for apps in the main group ###
###                                                     ###
###########################################################
group_deploy main
```

For this script, we need to pass values file-name, and we should have "yq" and "jq" Linux tools installed to parse our file.

The command to install our chart will be like this:

```
deploy_all.sh $VALUES_FILE
```

Thats it!

Now we can install thousands of applications at once and manage them in one simple file.

Important notes:

It is essential to understand that within every Helm template for any given loop, we are required to define new variables within the context. This is a mandatory requirement when working with Golang templates.

Defining new variables within each Helm template loop is crucial for several reasons:

**Scope Control**: It helps manage variable scope, avoiding conflicts and ensuring variables are only accessible within their intended context.

**Clarity**: Makes the template easier to read and understand by clearly outlining what data a block operates on.

**Reusability**: Enhances template flexibility, allowing for dynamic adjustments based on context, such as different configurations for environments or apps.

**Error Reduction**: Isolating variables reduces the risk of errors from overwrites and aids in debugging.

**Customization**: Supports complex configurations by merging dictionaries and setting defaults, allowing for specific overrides and sensible fallbacks.

This approach is fundamental for maintainable, clear, and flexible Helm templates, ensuring they can cater to various deployment needs efficiently.

Yes, undoubtedly, this approach has many advantages, but there are also several downsides. For example, when the number of services increases to several hundred, it complicates the maintenance of a single file with variables.

**Conclusion.**

In conclusion, deploying and managing applications in a complex Kubernetes environment can be a challenging task. The challenges of handling dependencies, configurations, and versioning across multiple components can quickly become overwhelming. However, with the introduction of HELM, a powerful package manager for Kubernetes, these challenges can be effectively addressed.

With our approach of Universal Helm Chart, we go even further to standardize and reuse Helm Chart that can be applied to multiple applications or services in a Kubernetes environment. It serves as a template that encapsulates common configuration settings, dependencies, and deployment instructions for various applications. The Universal Helm Chart promotes code reuse and consistency by providing a single source of truth for deploying and managing multiple applications within a Kubernetes cluster. By leveraging the Universal Helm Chart, developers, operators, and system administrators can simplify the deployment process, reduce duplication of effort, and ensure consistency and scalability across their Kubernetes-based applications.

## *References / Список литературы*

1. Official HELM project website [Electronic Resource]. URL: https://helm.sh/ (date of access: 19.02.2024).
2. Official GNU Bash project website [Electronic Resource]. URL: https://www.gnu.org/software/bash/ (date of access: 12.02.2024).
3. Official Kubernetes project website [Electronic Resource]. URL: https://kubernetes.io/ (date of access: 10.02.2024).

4. Wikipedia pages: [Electronic Resource]. URL: https://en.wikipedia.org/wiki/Don%27t_repeat_yourself/ (date of access: 05.02.2024).