Helm Interview Questions and Answers

What is Helm?

Helm is a package manager for Kubernetes. It allows you to easily install, upgrade, and manage applications on Kubernetes. Helm packages applications into charts, which are YAML files that contain all of the configuration files and Kubernetes resources needed for the application.

What are the benefits of using Helm?

There are many benefits to using Helm, including:

- **Ease of use:** Helm makes it easy to install, upgrade, and manage applications on Kubernetes.
- **Consistency:** Helm charts ensure that applications are deployed consistently across environments.
- **Reusability:** Helm charts can be reused to deploy applications to different Kubernetes clusters.
- **Community:** Helm has a large and active community that develops and maintains charts for a variety of applications.

What are the different types of Helm charts?

There are two main types of Helm charts:

- **Application charts:** Application charts contain the resources for a single application.
- **Library charts:** Library charts provide reusable components that can be used by other charts.

What is a Helm chart?

A Helm chart is a package that contains all of the configuration files and Kubernetes resources needed for an application. Helm charts are YAML files that are organized into a specific directory structure.

What is a Helm release?

A Helm release is an instance of a Helm chart that has been deployed to a Kubernetes cluster. Each release has a unique name and can be upgraded or deleted.

What is a Helm repository?

A Helm repository is a storage location where Helm charts are stored and shared. Helm repositories can be local or remote.

What is Tiller?

Tiller is the server-side component of Helm. It manages the lifecycle of releases and provides the API that the Helm client uses to interact with Kubernetes.

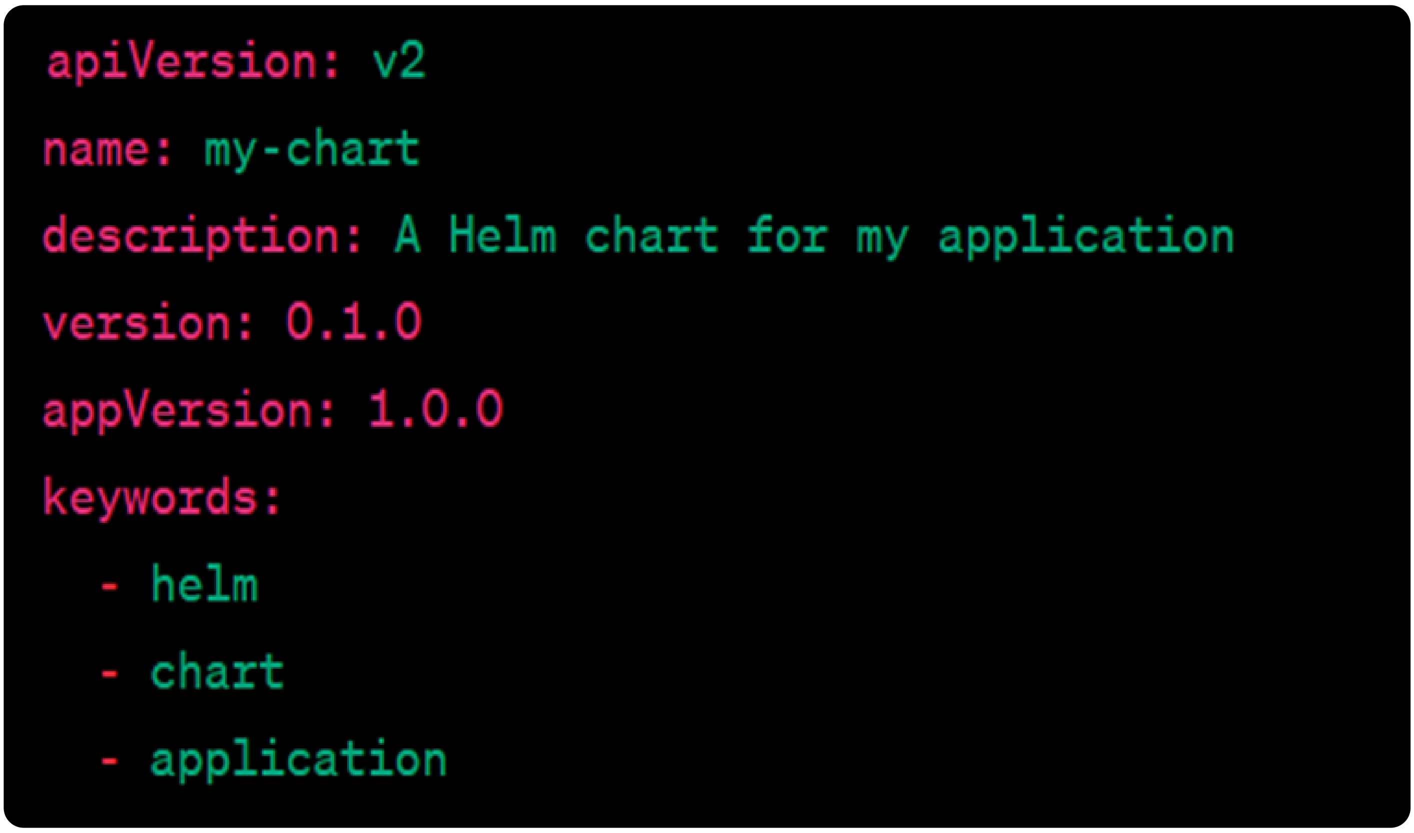
Explain Helm Chart Structure in detail

Helm Chart Structure Directory

```
my-chart/
|-- Chart.yaml
|-- values.yaml
|-- charts/
|    |-- dependency-chart/
|        |-- charts/
|        |-- templates/
|-- templates/
|    |-- deployment.yaml
|    |-- service.yaml
|    |-- config.yaml
|    |-- secrets.yaml
|    |-- _helpers.tpl
|-- requirements.yaml
|-- NOTES.txt
|-- values.schema.json
|-- _config/
|    |-- additional-config.yaml
```

1. Chart.yaml

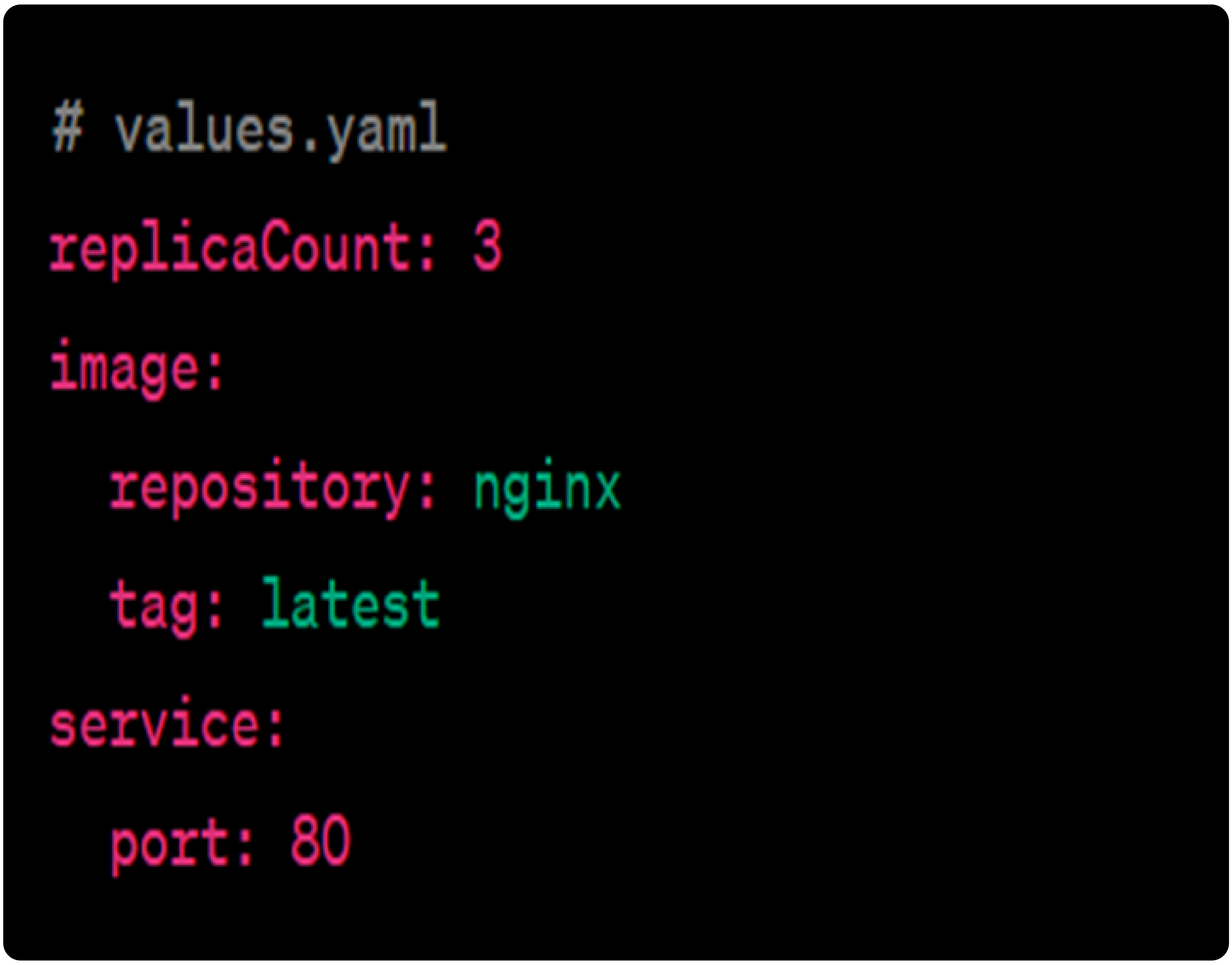
1. The Chart.yaml file is a metadata file that plays a central role in Helm charts.
2. Enables Helm to understand the chart, making it shareable, versioned, and easy to manage.
3. This is where other charts on which the helm chart that is being structured depends on are stored.



```yaml
apiVersion: v2
name: my-chart
description: A Helm chart for my application
version: 0.1.0
appVersion: 1.0.0
keywords:
  - helm
  - chart
  - application
```

2. values.yaml

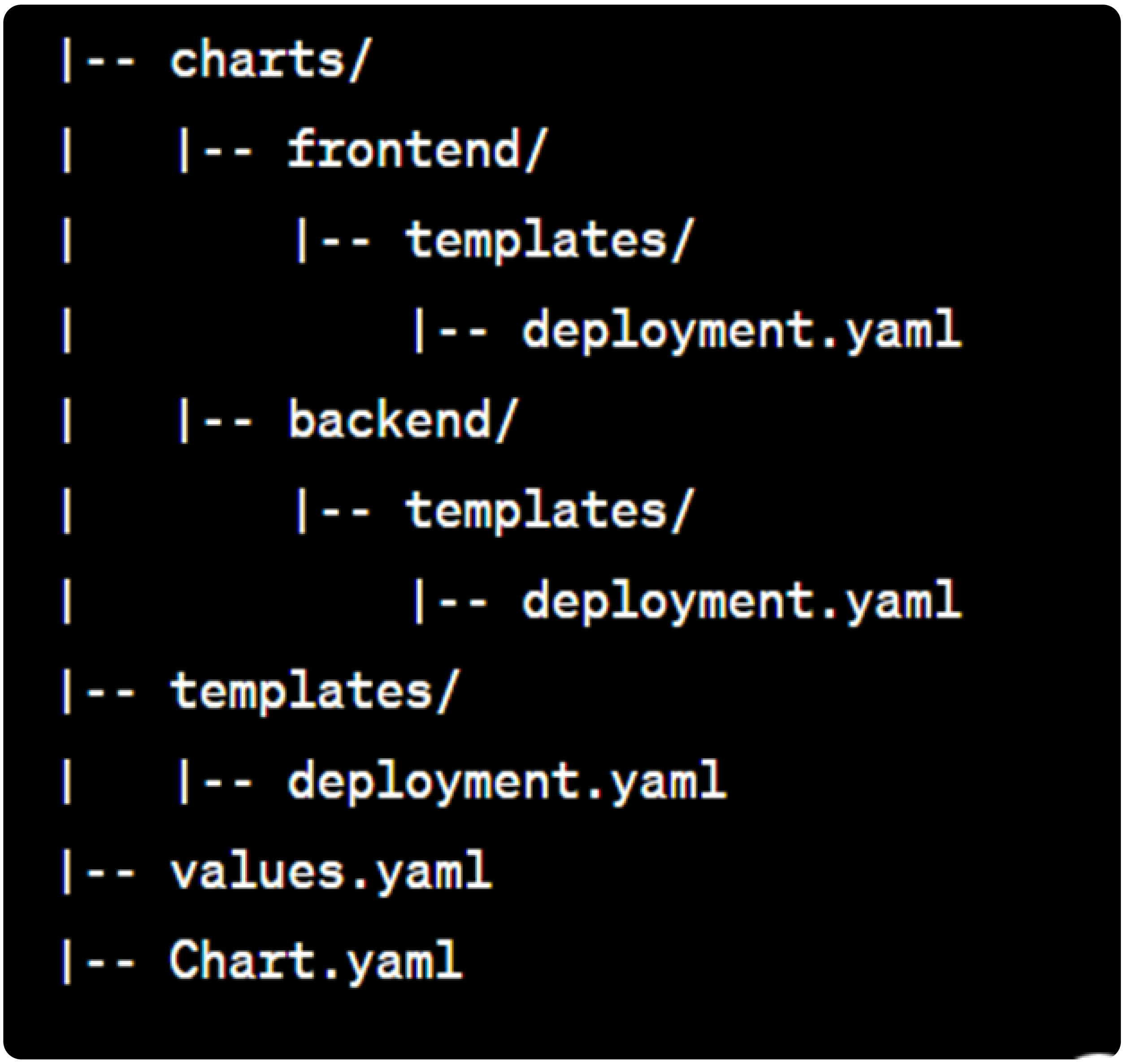
1. This is the file in which all the values that are to be injected into the templates are defined. Similar to terraform, Values.yaml is the same as helms variable.tf file.
2. The values.yaml file plays a pivotal role in Helm charts, serving as a configuration file that encapsulates default values for the various parameters used within the chart’s templates.
3. When installing a Helm chart, users have the flexibility to override these default values.



```yaml
# values.yaml
replicaCount: 3
image:
  repository: nginx
  tag: latest
service:
  port: 80
```

3. Charts/

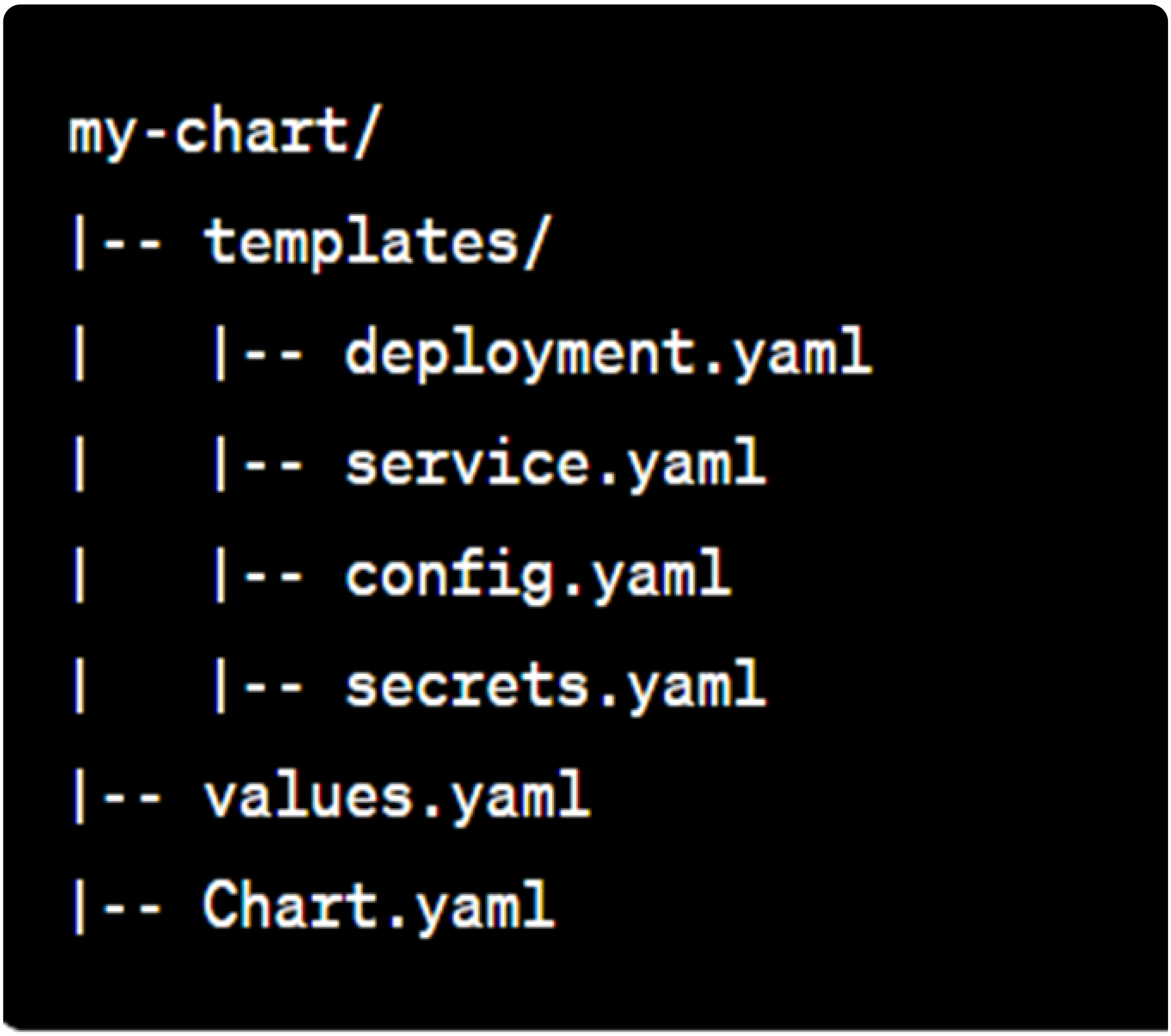
1. This is where other charts on which the helm chart that is being structured depends on are stored. There might be a need to call another chart for letting the chart function properly.
2. The charts/ directory within a Helm chart serves as a container for subcharts—other charts upon which the primary Helm chart depends.



```
|-- charts/
|   |-- frontend/
|       |-- templates/
|           |-- deployment.yaml
|   |-- backend/
|       |-- templates/
|           |-- deployment.yaml
|-- templates/
|   |-- deployment.yaml
|-- values.yaml
|-- Chart.yaml
```

4. templates/

1. The templates/ directory is a core component within Helm charts, housing Kubernetes resource definitions.
2. This is the folder where the actual manifest that is being deployed with the chart is put.
3. For instance,for deploying an nginx deployment that needs a service, configmap and secrets, there would be a deployment.yaml, service.yaml, config.yaml and secrets.yaml all in the template dir.
4. They will all get their values from values.yaml from these.



```
my-chart/
|-- templates/
|    |-- deployment.yaml
|    |-- service.yaml
|    |-- config.yaml
|    |-- secrets.yaml
|-- values.yaml
|-- Chart.yaml
```

/templates files:-

- **deployment.yaml:** Contains the configuration for a Kubernetes Deployment, specifying how the application should run and scale.
- **service.yaml:** Defines a Kubernetes Service, exposing the application to other services within the cluster.
- **config.yaml:** Represents a ConfigMap, a mechanism to inject configuration data into the application.
- **secrets.yaml:** Describes Kubernetes Secrets, which are used to store sensitive information securely.

These files in the templates/ directory collectively define the Kubernetes resources necessary for deploying and configuring the application encapsulated by the Helm chart.

4.1 deployment.yaml

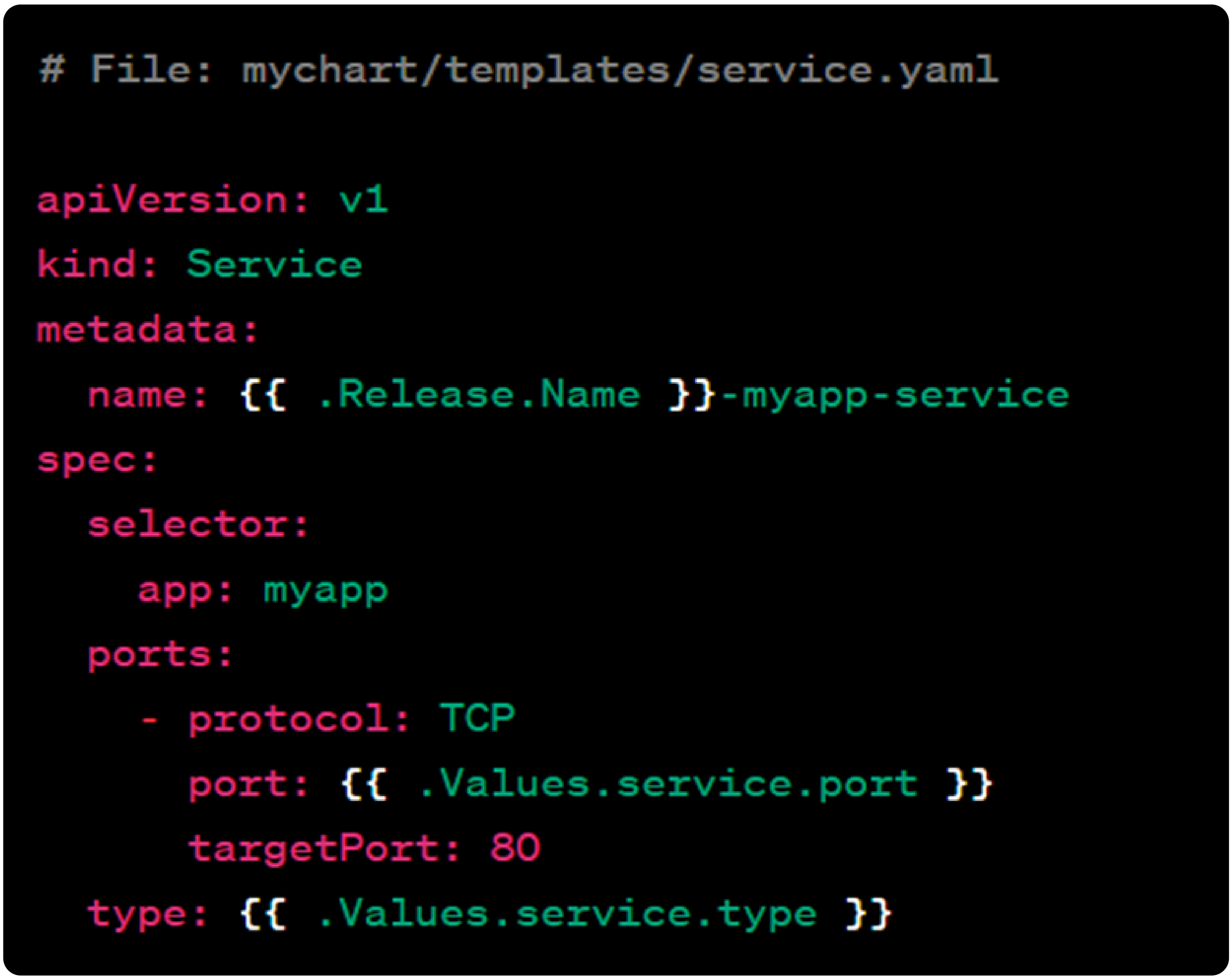
- This file defines a Kubernetes Deployment, which is a resource object in Kubernetes that provides declarative updates to applications. A Deployment allows you to describe an application’s life cycle, such as which images to use for the app, the number of pod replicas, and the way to update them.

```yaml
# File: mychart/templates/deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-myapp
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp-container
        image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
        ports:
        - containerPort: {{ .Values.service.port }}
```

- ‘apiVersion’ and ‘kind’ specify the Kubernetes API version and resource kind, respectively.
- ‘metadata’ contains the name of the Deployment, which is generated based on the Helm release name.
- ‘spec’ defines the desired state of the Deployment.
- ‘replicas’ specifies the number of replicas (pods) to run.
- u‘selector’ is used to match the pods controlled by this Deployment.
- ‘template’ is the pod template for the Deployment.
- ‘metadata’ contains labels that match the selector.
- ‘containers’ defines the container(s) running in the pod, including the image and ports.

4.2 service.yaml

- This file defines a Kubernetes Service, which enables external access to the application and routes traffic to the pods managed by a Deployment.

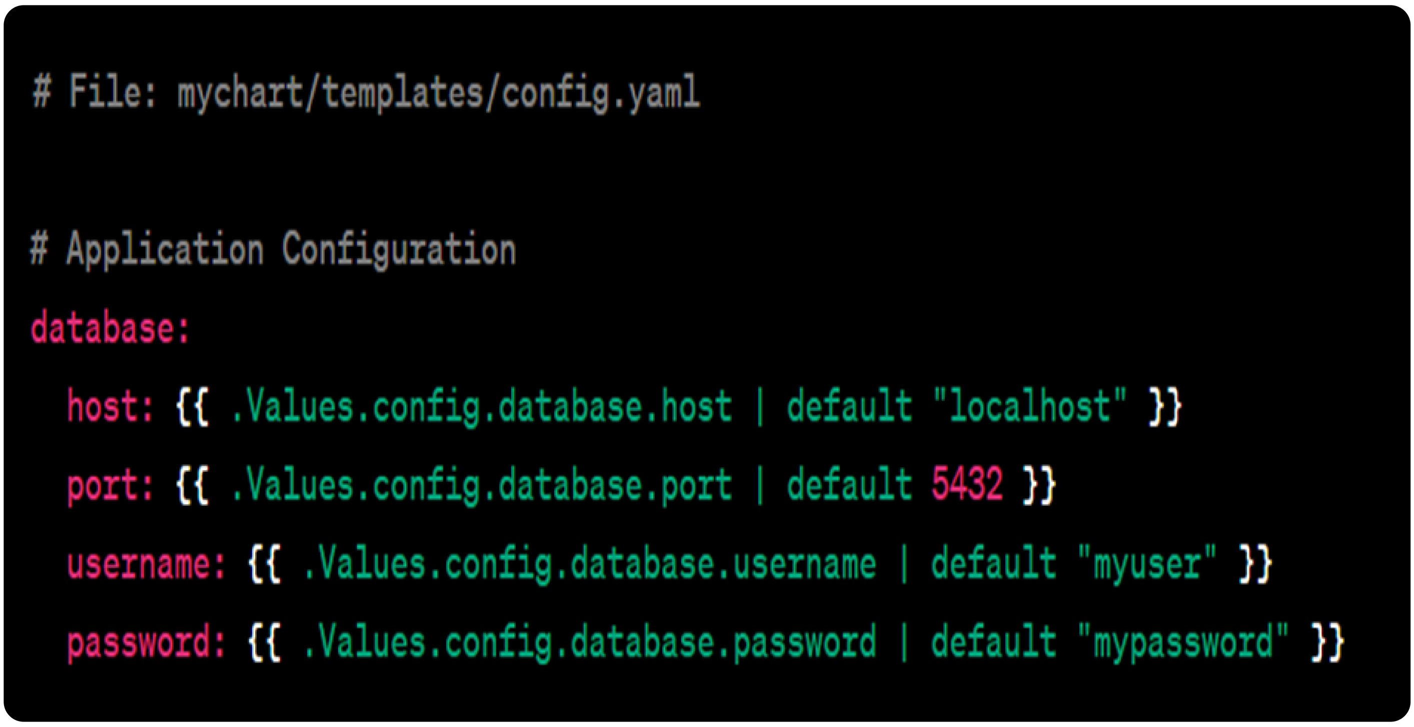


```yaml
# File: mychart/templates/service.yaml

apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}-myapp-service
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: {{ .Values.service.port }}
      targetPort: 80
  type: {{ .Values.service.type }}
```

- ‘apiVersion’ and ‘kind’ specify the Kubernetes API version and resource kind, respectively.
- ‘metadata’ contains the name of the Service, which is generated based on the Helm release name.
- ‘spec’ defines the desired state of the Service.
- ‘selector’ is used to match the pods controlled by the corresponding Deployment.
- ‘ports’ specifies the ports on which the Service will listen.
- ‘type’ indicates the type of Service, such as ClusterIP, NodePort, or LoadBalancer. The type is sourced from the Helm values file.

4.3 config.yaml

- The config.yaml file is often used to manage configuration settings for an application. It allows you to separate configuration from code, makin it easier to customize the behavior of your application without modifying the application itself.

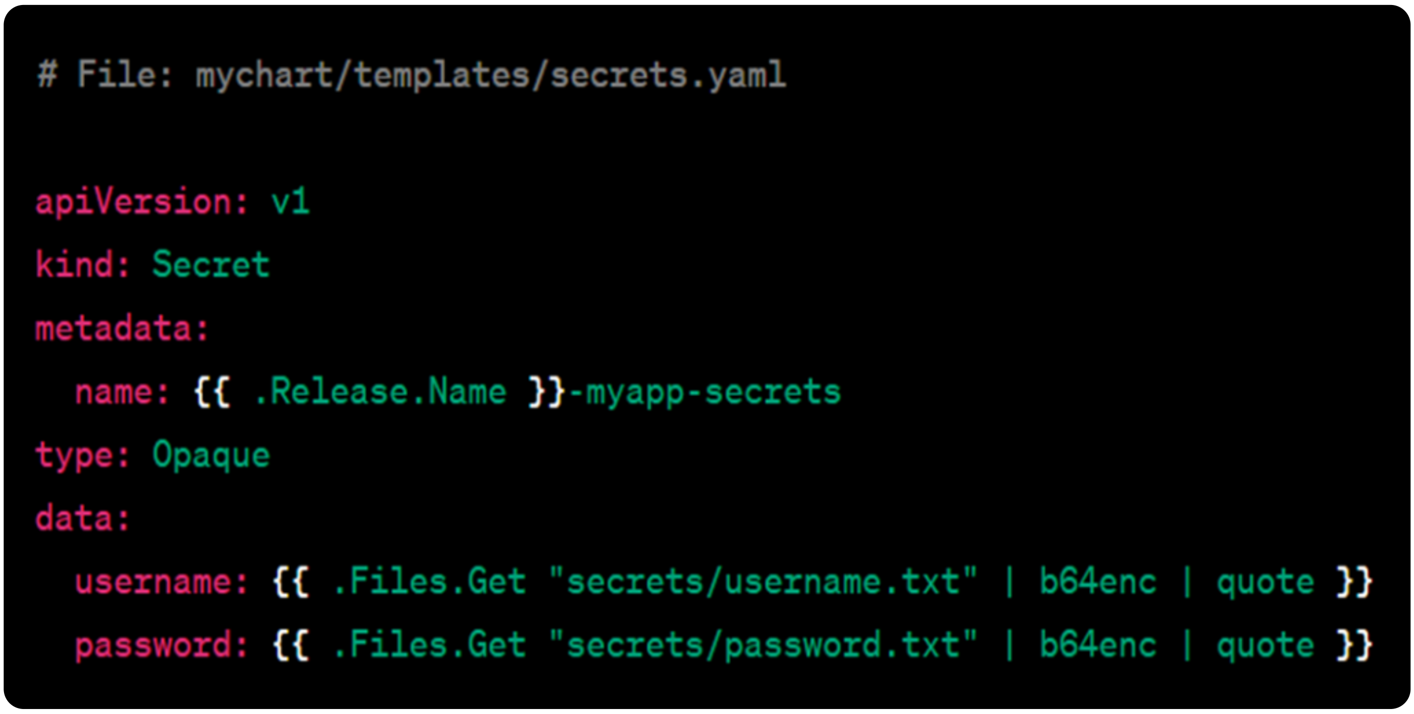


```yaml
# File: mychart/templates/config.yaml

# Application Configuration
database:
  host: {{ .Values.config.database.host | default "localhost" }}
  port: {{ .Values.config.database.port | default 5432 }}
  username: {{ .Values.config.database.username | default "myuser" }}
  password: {{ .Values.config.database.password | default "mypassword" }}
```

- ‘.Values.config.database.host’, ‘.Values.config.database.port’,‘.Values.config.database.username’, and ‘.Values.config.database.password’ are placeholders for configuration values.
- The ‘|’ default filter is used to provide default values if the corresponding values are not specified in the Helm values file.

4.4 secrets.yaml

- The ‘secrets.yaml’ file is typically used to manage sensitive information securely. Secrets can be mounted as files or used as environment variables within your application, and Helm helps manage these secrets during deployment.



```yaml
# File: mychart/templates/secrets.yaml

apiVersion: v1
kind: Secret
metadata:
  name: {{ .Release.Name }}-myapp-secrets
type: Opaque
data:
  username: {{ .Files.Get "secrets/username.txt" | b64enc | quote }}
  password: {{ .Files.Get "secrets/password.txt" | b64enc | quote }}
```

- ‘.Release.Name’ is used to generate a unique name for the secret based on the Helm release name.
- ‘type: Opaque’ indicates a generic secret type.
- ‘data’ contains the actual secret information. Here, it loads the contents of two files (‘username.txt’ and ‘password.txt’) from the ‘secrets/’ directory, encodes them in base64, and includes them in the secret.

5. Other helm chart files

1. _helpers.tpl: A Helm template file that contains reusable template snippets and functions. These can be shared across multiple charts for consistency.
2. requirements.yaml: Specifies dependencies for the Helm chart, allowing you to declare other charts that must be present for the primary chart to function properly.
3. NOTES.txt: A file containing post-installation and post-upgrade notes that are displayed to the user after the Helm chart is deployed.
4. LICENSE: This file includes the license information for the Helm chart.
5. README.md: This file provides documentation and information about the Helm chart.
6. .helmignore: Similar to .gitignore, this file specifies files or directories that should be ignored when packaging the chart.
7. tests/: This directory may contain test files and scripts to validate the Helm chart.

5.1 _helpers.tpl

- The _helpers.tpl file in a Helm chart is a template helper file that contains reusable template snippets or functions. It allows you to define reusable pieces of code that can be included in other template files in your Helm chart.
- The filename typically starts with an underscore (_) to indicate that it’s a helper file and not meant to be rendered as a standalone Kubernetes resource.

```
# File: mychart/templates/_helpers.tpl

{{/*
Create a helper function to generate labels for resources.
Usage: {{ include "mychart.labels" $labels | indent 4 }}
*/}}
{{- define "mychart.labels" -}}
  {{- $labels := .Values.labels | merge .Chart.labels | merge $ }}
  {{- with .Values.podLabels -}}
    {{- $labels = $labels | merge . -}}
  {{- end -}}
  labels:
    {{- range $key, $value := $labels }}
    {{ $key }}: {{ quote $value }},
    {{- end }}
{{- end }}
```

5.2 requirements.yaml

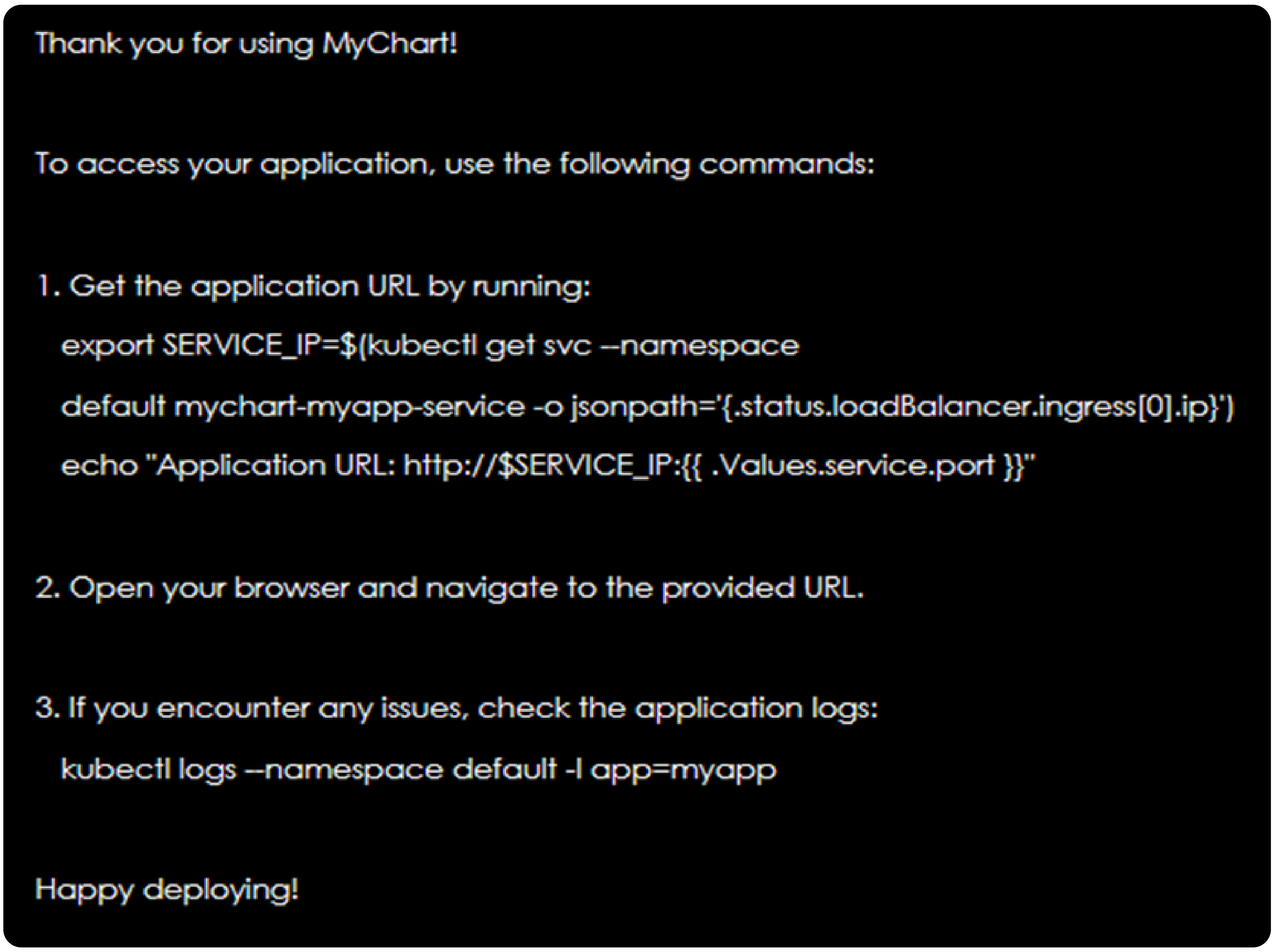
- The ‘requirements.yaml’ file is used to declare dependencies for your Helm chart. It specifies other charts that your chart depends on, including the chart name, version constraints, and any additional settings.

```
# File: mychart/requirements.yaml

dependencies:
  - name: nginx
    version: "1.2.3"
    repository: "https://charts.example.com/stable"
```

- dependencies is a list of dependencies that your chart relies on.
- name is the name of the dependency chart.
- version is the version constraint for the dependency. Helm will attempt to use a version that satisfies this constraint.
- repository is the URL of the Helm chart repository where the dependency can be found.

5.3 NOTES.txt

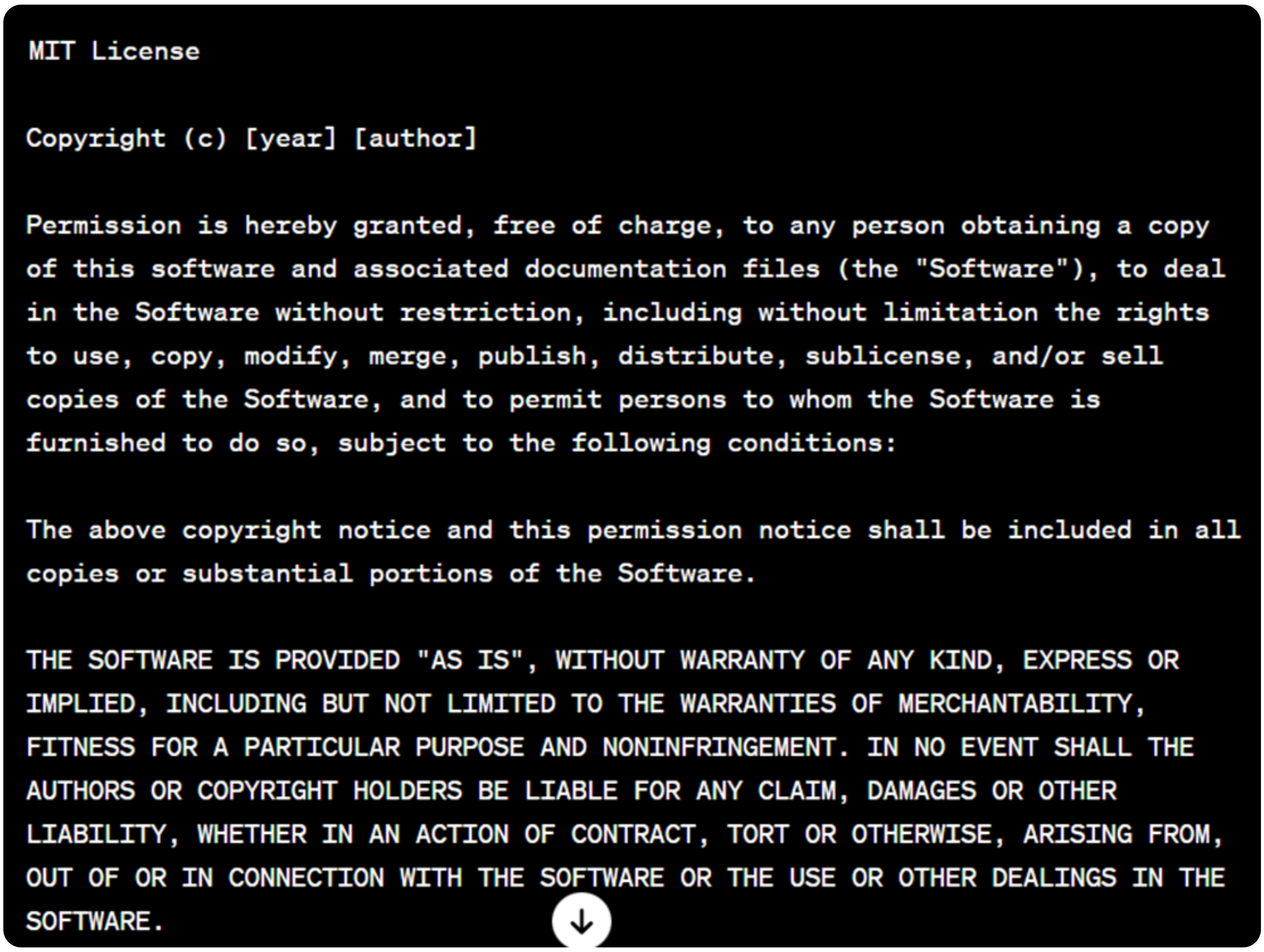
- The NOTES.txt file in a Helm chart is used to provide post-installation notes or information that is displayed to the user after a Helm chart is deployed. These notes can include instructions, tips, or any other relevant information that users should be aware of after the deployment process.
- When users install your Helm chart, Helm will automatically display the content of the NOTES.txt file in the command line. Users can follow the provided instructions and notes to access and interact with the deployed application.



```
Thank you for using MyChart!

To access your application, use the following commands:

1. Get the application URL by running:
  export SERVICE_IP=$(kubectl get svc –namespace
  default mychart-myapp-service -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
  echo "Application URL: http://$SERVICE_IP:{{ .Values.service.port }}"

2. Open your browser and navigate to the provided URL.

3. If you encounter any issues, check the application logs:
  kubectl logs –namespace default -l app=myapp

Happy deploying!
```

5.4 LICENSE

- In a Helm chart, the LICENSE file typically contains information about the licensing terms and conditions for the Helm chart. It provides users and developers with details about how they can use, modify, and distribute the chart.



```
MIT License

Copyright (c) [year] [author]

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

- MIT License: Specifies the type of license (in this case, the MIT License).
- Copyright: States the copyright holder or holders.
- Permission is hereby granted…: Outlines the permissions granted to users under the license.
- The above copyright notice and this permission notice…: Specifies that the license terms must be included in all copies or substantial portions of the software.
- THE SOFTWARE IS PROVIDED "AS IS"…: Includes a disclaimer of warranty.
- IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS…: Includes a limitation of liability.

5.5 README.md

- The README.md file in a Helm chart serves as documentation to provide users with information about the chart, its purpose, how to use it, and any other relevant details.
- Replace my-release with the desired release name.

~~~
# MyChart

MyChart is a Helm chart for deploying and managing MyApplication on Kubernetes.

## Installation

To install MyChart, use the following Helm command:

```bash
helm install my-release ./mychart
~~~

5.6 .helmignore

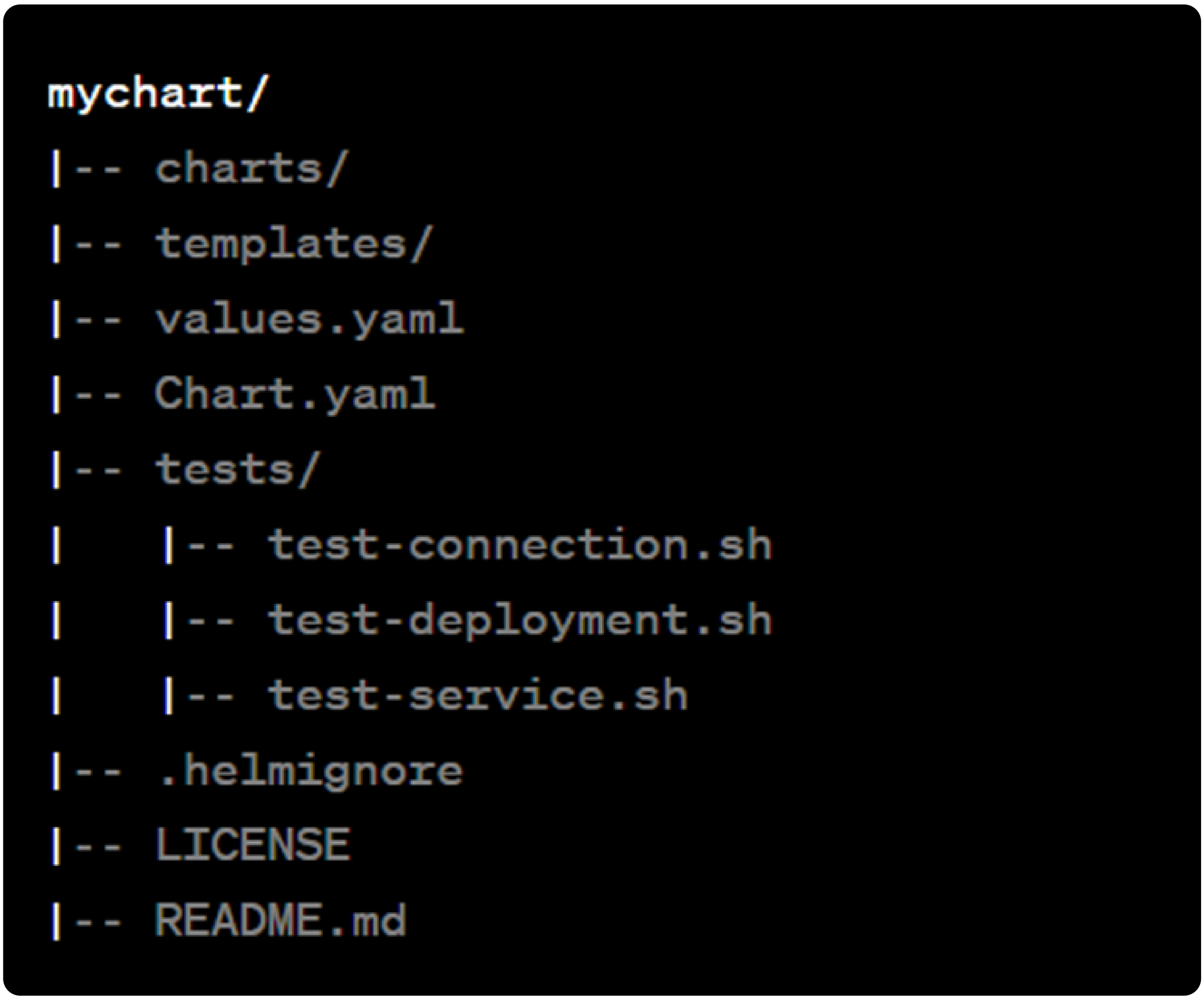
- It specifies patterns of files and directories that Helm should ignore when packaging the chart. Helm uses the ‘.helmignore’ file to determine which files to exclude when creating a chart package (tar.gz file) using the helm package command.

```
# Ignore files and directories generated during development or build
node_modules/
dist/
build/

# Ignore editor-specific files
.vscode/
.idea/

# Ignore temporary files
*.swp
*~
```

- node_modules/, dist/, and build/ are directories that might be generated during development or build processes and are typically not needed in the packaged Helm chart.
- .vscode/ and .idea/ are directories specific to Visual Studio Code and IntelliJ IDEA, respectively, and are not necessary in the packaged Helm chart.
- *.swp and *~ are examples of temporary files created by certain editorsand are generally not needed in the packaged Helm chart.

5.7 tests/

- In a Helm chart, the tests/ directory is commonly used to store test files and scripts that can be run to verify the correctness and functionality of the Helm chart. These tests are often automated and can be executed as part of a continuous integration (CI) pipeline or manually by users.



```
mychart/
|-- charts/
|-- templates/
|-- values.yaml
|-- Chart.yaml
|-- tests/
|    |-- test-connection.sh
|    |-- test-deployment.sh
|    |-- test-service.sh
|-- .helmignore
|-- LICENSE
|-- README.md
```

- tests/ is a directory containing test scripts.
- ‘test-connection.sh’, ‘test-deployment.sh’ and ‘test-service.sh’are examples of test scripts that can verify aspects of the chart, such as connectivity, deployment, and service accessibility.

Helm Commands Interview Questions and Answers

What is the command to install a Helm chart?

The command to install a Helm chart is `helm install <chart_name>`. This command will download the chart from the specified repository and deploy it to the Kubernetes cluster.

What is the command to upgrade a Helm release?

The command to upgrade a Helm release is `helm upgrade <release_name> <chart_name>`. This command will update the release with the latest version of the chart.

What is the command to delete a Helm release?

The command to delete a Helm release is `helm delete <release_name>`. This command will delete the release from the Kubernetes cluster.

Scenario Based Helm Interview Questions and Answers

How do you manage sensitive data in Helm charts?

You can use Helm secrets to manage sensitive data in Helm charts. Helm secrets are encrypted files that are stored separately from the chart. When a chart is deployed, the secrets are decrypted and injected into the Kubernetes resources.

How do you create and publish Helm charts?

You can use the `helm create` command to create a new Helm chart. Once you have created a chart, you can publish it to a public or private Helm repository.

How do you use Helm in a continuous integration and continuous delivery (CI/CD) pipeline?

You can use Helm to automate the deployment of applications to Kubernetes in a CI/CD pipeline. This can be done by integrating Helm with your CI/CD tool of choice.

Explain the purpose of values.yaml in a Helm Chart.

`values.yaml` is a file in a Helm Chart that defines default values for customizable parameters. During installation, users can override these values, allowing for dynamic and flexible deployments.

Explain the difference between Helm and Kubectl.

`kubectl` is a command-line tool for interacting with Kubernetes clusters, whereas Helm is a higher-level package manager specifically designed for deploying and managing applications on Kubernetes. Helm uses `kubectl` under the hood but adds functionality such as templating and release management.

Explain Helm chart best practices?

Modularization for Scalability: Utilize the charts/ directory to modularize Helm charts, promoting component reuse and enhancing scalability.

Centralized Configuration with values.yaml: Maintain a centralized values.yaml file for clear configuration management, ensuring default values and user overrides are easily accessible.

Organized Resource Definitions: Organize Kubernetes resource definitions in the templates/ directory for clarity and easy reference. Consider using subdirectories for better organization.

Harness Helm Templating: Leverage Helm templating in the templates/ directory for dynamic and customizable resource creation.Inject values from values.yaml for flexibility during deployment.

Scenario 1:

Question: You are tasked with deploying a complex microservices-based application to a Kubernetes cluster. Explain how you would use Helm to manage the deployment efficiently.

Answer: I would create a Helm chart for the application, organizing the different microservices as subcharts. The Helm chart would include templates for Kubernetes manifests, values files for customization, and a Chart.yaml file for metadata. By using Helm, I can easily deploy, upgrade, and roll back the entire application or individual microservices. Helm’s templating also allows for parameterization, making it easy to customize deployments for different environments.

Scenario 2:

Question: Your team is working on an application that requires different configurations for development, staging, and production environments. How would you structure your Helm charts to handle environment-specific configurations?

Answer: I would use Helm’s values files to manage environment-specific configurations. Each environment would have its own values file containing the necessary overrides. Additionally, I could leverage Helm’s release feature to manage releases for each environment separately, ensuring a clean and organized approach to environment-specific configurations.

Scenario 3:

Question: You have a Helm chart for a web application that includes a frontend and a backend. How would you handle dependencies between these components using Helm?

Answer: I would define the frontend and backend components as separate subcharts within the main Helm chart. Helm allows for dependencies between charts, so I would specify the backend as a dependency for the frontend. This way, Helm would install and manage the backend before deploying the frontend, ensuring that all dependencies are satisfied.

Scenario 4:

Question: Your team follows a GitOps workflow, and you want to integrate Helm into your continuous integration pipeline for automatic deployments. Explain how you would set up this pipeline.

Answer: In the CI pipeline, I would use Helm commands such as `helm install` or `helm upgrade` to deploy the application based on changes to the Helm chart or values files. The Helm chart and values files would be version-controlled in the Git repository. Changes to these files would trigger the CI pipeline, which would, in turn, deploy or upgrade the application in the Kubernetes cluster using Helm commands.

Scenario 5:

Question: Your application requires a database, and you want to use Helm to deploy a database along with your application. How would you manage the database dependencies using Helm?

Answer: I would create a separate Helm chart for the database and include it as a dependency in the main application Helm chart. Helm allows for defining dependencies in the `requirements.yaml` file. This way, when deploying the main application, Helm would automatically manage the deployment of the database chart as well, ensuring that both components are deployed and configured correctly.

Scenario 6:

Question: Your team needs to perform a rollback of a recent deployment due to issues in the current release. Explain how you would use Helm to rollback to a previous version.

Answer: I would use the `helm rollback` command, specifying the release name and the revision number of the desired version to which I want to roll back. Helm maintains a history of releases, and the rollback command allows me to revert to a specific version, undoing any changes introduced in the problematic release.

These scenario-based questions and answers are designed to assess your practical understanding of how to use Helm in various real-world situations. They cover aspects such as structuring Helm charts, managing dependencies, handling environment-specific configurations, and integrating Helm into CI/CD pipelines.

Conclusion:

We have covered Helm Interview Questions and Answers,Helm Commands Interview Questions and Answers,Scenario Based Helm Interview Questions and Answers.