

hey i have a problem , i was learning helm , but i referred 2 sources, can you make god level notes with codes and all split into around 10 message by message by merging the 2 source notes strategically and constructively with codes and all, heres the guide to make g

💡 A Guide to "God-Level" Note-Taking

The philosophy is simple: Notes should not be a transcript. They should be a structured, easy-to-read guide that helps you re-build the ground up. This method is built on 5 core steps.

Step 1: 💡 Identify the Core Concept (The "Why")

Your first and most important task is to find the central purpose of the lecture. Before you write down any details, you must be able to answer the following questions in one or two simple sentences:

What is the single biggest problem this lecture is trying to solve?

If I could only remember one thing from this, what should it be?

What is the high-level "thesis" or main idea?

This answer becomes your Core Concept section. It acts as the anchor for all the other details and ensures you know why you're learning this.

Step 2: ⚙️ Break Down the Process (The "How")

This is where you capture all the practical, step-by-step actions. Your goal is to create a guide that someone (especially your future self) can replicate the results exactly.

Look for:

Every terminal command.

Every specific code change.

Every configuration step (in a file or a UI).

The exact sequence these actions happen in.

Special Rule for Code:

Do not just save the final code. Capture all iterations of the code the instructor writes. For each version, document why they made changes. (e.g., "We hard-coded the value. Then, we refactored it to use a variable for flexibility.") This is how you learn the process of coding, not just the final result.

Step 3: 🧠 Structure and Organize for Clarity

Raw text is hard to read. This step transforms a "wall of text" into a scannable, well-organized document that is easy to digest at a glance.

Use these tools generously:

Headings (##, ###): To create a clear hierarchy (e.g., "Part 1: Setup", "Part 2: Training the Model").

Emojis (💡, ⚙️, 🧠, 🌟): As visual signposts for each section.

Bolding: To make key terms, commands, and filenames stand out.

Code Blocks: To keep all code, commands, and config files clean and easy to copy.

Bullet Points: To break down lists or supporting ideas.

Step 4: 🧠 Explain the Underpinning Theory

This is what separates "good" notes from "god-level" notes. You must add the conceptual context behind the practical steps. For each step, you should be able to answer "Why?"

Answer questions like:

Why did we choose this algorithm over another?

What is the conceptual difference between command A and command B?

What problem does this specific line of code solve?

Why did this error happen, and what does it mean?

This section adds the crucial context that turns simple instructions into lasting knowledge.

Step 5: 🌟 Distill the Key Takeaways

Finally, review your entire set of notes and distill the 3-5 most critical, must-remember lessons into a concise summary. This section is for your future review.

This should be a short, powerful list of the main points, warnings, or "aha!" moments from the lecture.

This 5-step process ensures you end up with a set of notes that are not just a record of what was said, but a comprehensive learning tool for exams and real-world projects. start by providing the modules you will split them into then we can start with module 1



Hello! This is an excellent project. You've got two high-quality, dense sources of information, and merging them is the perfect way to build a deep understanding.

I have analyzed both sources and your "God-Level" note-taking guide. I've strategically merged the practical, project-based flow of the first source with the strong conceptual "why" (like "Snowflake vs. Phoenix" clusters) of the second.

The result is a comprehensive, 20-module learning path that takes you from a complete beginner to a professional Helm developer. We will cover one module at a time, and I will generate the "God-Level" notes for each one.

Here is the complete syllabus we will follow:

Your "God-Level" Helm Syllabus

Part 1: The "Why" & Environment Setup

- **Module 1:** The Core Problem & What is Helm?
- **Module 2:** The "Wild West": Finding Charts & Repositories (OCI vs. Classic)
- **Module 3:** Local Lab Setup (WSL, Minikube, kubectl & VS Code)

Part 2: The "User" Workflow (Using Public Charts)

- **Module 4:** Installing Your First Chart (`helm install` , `ls` , `uninstall`)
- **Module 5:** The "Snowflake" Problem & Basic Overrides (`helm show values` , `--set`)
- **Module 6:** The "Phoenix" Solution: Custom `values.yaml` (`-f` flag)
- **Module 7:** The "Phoenix" Workflow 1: Taking Control with `helm pull`
- **Module 8:** The "Phoenix" Workflow 2: Generating Raw YAML with `helm template`
- **Module 9:** Managing Live Releases: Upgrades & Rollbacks (`helm upgrade` , `history` , `rollback`)

Part 3: The "Developer" Workflow (Authoring Your Own Charts)

- **Module 10:** Why Write Your Own Chart? (The "Configuration Drift" Problem)
- **Module 11:** Creating Your First Chart (`helm create` , `Chart.yaml` , `{{ .Values }}`)
- **Module 12:** Go Template Deep Dive 1: Functions & Pipelines (`|` , `default` , `quote`)
- **Module 13:** Go Template Deep Dive 2: Flow Control (`if/else` , `eq` , `with`)
- **Module 14:** Go Template Deep Dive 3: Iteration with `range`
- **Module 15:** Reusable Code: Named Templates (`_helpers.tpl` , `define` , `include` , `indent`)

Part 4: Advanced Concepts & Capstone Project

- **Module 16:** Managing Dependencies & Subcharts (`dependencies` in `Chart.yaml`)
- **Module 17:** Project: Building a Two-Tier App (Part 1 - The App & Dockerfile)
- **Module 18:** Project: Building a Two-Tier App (Part 2 - Wiring the DB Dependency)
- **Module 19:** Advanced Lifecycle Management: Chart Hooks & Tests (`helm test`)
- **Module 20:** Deconstructing a "Pro" Chart (`{{ .Release.Name }}` & Library Charts)

This structure builds your knowledge brick by brick, ensuring you understand *why* you're doing something before you learn *how*.

When you are ready, just say the word, and we will begin with **Module 1: The Core Problem & What is Helm?**



Here are the "God-Level" notes for the first module.

💡 Module 1: The Core Problem & What is Helm?

Core Concept (The "Why")

The core problem is that **installing applications on Kubernetes is complex**.

A single modern application (like WordPress or MySQL) isn't just one thing. It's a collection of many interconnected Kubernetes resources:

- A `Deployment` for the app itself.
- A `Service` to expose it.
- `ConfigMaps` for configuration.
- `Secrets` for passwords.
- A `StatefulSet` or another `Deployment` for the database.
- `PersistentVolumeClaims` for storage.

Managing all these 5-10 separate YAML files manually for every app is difficult, error-prone, and impossible to replicate consistently.

Helm solves this by acting as the package manager for Kubernetes. It bundles all the necessary YAML files and configurations into a **Chart**. This allows you to install, upgrade, and manage a complete, complex application with a single command.

⚙️ The Process (The "How")

This module is conceptual, but the core workflow difference is simple:

- **Without Helm:** You (the user) are responsible for the complex process:
 1. `kubectl apply -f 01-database-secret.yaml`
 2. `kubectl apply -f 02-database-pvc.yaml`
 3. `kubectl apply -f 03-database-deployment.yaml`
 4. `kubectl apply -f 04-wordpress-deployment.yaml`
 5. `kubectl apply -f 05-wordpress-service.yaml`
 6. ...and so on, in the correct order.
 - **With Helm:** Helm handles all the complexity for you:
 1. `helm install my-blog bitnami/wordpress`
-

🔴 Underpinning Theory

The Package Manager Analogy This is the best way to think about Helm. It is to Kubernetes what other package managers are to their ecosystems.

- `apt` or `yum` is to Linux
- `pip` is to Python
- `brew` is to macOS

It simplifies finding, installing, and managing the software (Kubernetes applications).

Key Terminology

- **Chart:** The Helm package. It's a folder containing all the YAML templates and default configuration files for an application.
- **Repository:** A server that hosts and shares charts. (e.g., the `bitnami` repository we'll see later).
- **Release:** A specific *instance* of a chart that has been installed in your cluster. You can install the same `mysql` chart twice: once with the release name `dev-db` and again with the release name `prod-db`. They are two separate releases.

CRITICAL: Helm 3 vs. Helm 2 (and Tiller) You will see old tutorials (pre-2020) that are now dangerously wrong.

- **Helm 2** required a server-side component called **Tiller**. This was a pod that ran inside your cluster with high-level permissions, creating a deployment and maintenance headache.
- **Helm 3 (which we use) is "Tiller-less"**. It is a simple client-side tool. The `helm` command-line interface (CLI) uses your local `kubernetes` credentials to talk *directly* to the Kubernetes API server, just like `kubectl` does. This is much simpler and more secure.
- Helm 3 stores its information about releases as `Secret` resources directly inside your Kubernetes cluster.

🔑 Key Takeaways

- Helm is the **package manager** for Kubernetes.
- It solves application complexity by bundling all of an app's YAML files into a single package called a **Chart**.
- A **Release** is a single, specific installation of a Chart in your cluster.
- Modern Helm (v3+) is **Tiller-less** (client-only), simpler, and more secure.

We are all set for **Module 2: The "Wild West": Finding Charts & Repositories (OCI vs. Classic)**. Just let me know when you're ready!



Here are the "God-Level" notes for Module 2.

💡 Module 2: The "Wild West": Finding Charts & Repositories

Core Concept (The "Why")

The core problem is: **how do you find and trust a Helm chart?**

The original, central, and trusted `stable` repository (which was like an official app store) is now **dead**. It was deprecated because it was too large for open-source Helm maintainers to vet and maintain every chart.

This has turned the Helm ecosystem into a decentralized "Wild West". The responsibility of finding and *trusting* a chart has moved from the community entirely to **you**, the user. Your central tool for this is `artifacthub.io`, which acts as a search engine (like Google) for all the different repositories.

⚙️ The Process (The "How")

Your workflow for finding and using a chart has three parts: Research, Add, and Install.

1. Research (Finding the Chart)

- You **search** for charts on `artifacthub.io`.
- **Crucial:** Artifact Hub is **not** a repository. It's a *search engine* that indexes charts from many different, independent repositories.

2. Assess Trust (The "Detective Work")

- A search for "mysql" will return many results from different organizations.
- You **must vet the publisher**.
- **Check the Organization:** Is it from a trusted publisher like **Bitnami** (owned by VMware) or a random individual?
- **Check Popularity:** Look at the "stars" or download counts to see community adoption.

3. Use the Chart (The Two Methods) There are two "protocols" for storing charts. You must know which one you're using.

- **Method A: The "Classic" Repository (e.g., `prometheus-community`)** This is the traditional method for community charts.
 1. `helm repo add`: You must first "add" the remote repository to your local client, giving it a short name (a "label"). You only do this

Bash

```
# Command: helm repo add [LOCAL_LABEL] [REPO_URL]
```



```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
```

2. `helm repo update` : This command is like `apt update` . It fetches the latest "catalog" of charts and versions from *all* your added repos.
3. `helm install` : You can now install using your local label.

Bash

```
helm install monitoring prometheus-community/kube-prometheus-stack
```

- **Method B: The "New" OCI Registry (e.g., `bitnami`)** This is the modern method, where charts are stored in container registries (like Docker Hub or GitHub Container Registry). This is the new standard as it reuses existing secure OCI infrastructure. Most major vendors (like Bitnami) have moved to this.
 1. **No "add" or "update"**: You don't need to add a repo.
 2. `helm install` : You install *directly* using the `oci://` URL.

Bash

```
# Command: helm install [RELEASE_NAME] oci://[REGISTRY_URL]
helm install my-blog oci://registry-1.docker.io/bitnamicharts/wordpress
```

🔗 Underpinning Theory

OCI vs. Classic Repos

- **Classic Repos** are simple web servers. They host the chart files (`.tgz`) and a single `index.yaml` file that acts as a catalog. `helm repo update` downloads this `index.yaml` file.
- **OCI Repos** are modern container registries (like Docker Hub or GitHub Container Registry). This is the new standard as it reuses existing secure OCI infrastructure. Most major vendors (like Bitnami) have moved to this.

Your Local Repository List

- You can see all the "Classic" repos you've added by running `helm repo list` .
- You can search *only* in your added repos using `helm search repo [query]` .
- **Note:** The `stable` repo (which may still appear in `helm repo list`) is **useless and deprecated**.

🌟 Key Takeaways

- The central, official `stable` repository is **dead**.
- The Helm ecosystem is a decentralized "Wild West".
- `artifacthub.io` is your **search engine** to find charts.
- You are **responsible for vetting publishers**. Stick to trusted sources like **Bitnami**.
- There are two ways to get charts:
 1. **Classic:** `helm repo add` -> `helm repo update` -> `helm install` .
 2. **OCI:** `helm install oci://...` (one step).

Ready for **Module 3: Local Lab Setup (WSL, Minikube, kubectl & VS Code)**? Just let me know.



Here are the "God-Level" notes for Module 3.

💡 Module 3: Local Lab Setup (WSL, Minikube, kubectl & VS Code)

Core Concept (The "Why")

The core problem is: **how do you safely and cheaply practice Kubernetes and Helm?** You can't just use a production cluster, and cloud

The solution is to build a complete, local Kubernetes development lab on your own laptop. We will use **Minikube** to create a free, single-node right on your machine. For Windows users, we'll use **WSL (Windows Subsystem for Linux)** to create the real Linux environment needed for tools.

⚙️ The Process (The "How")

Follow these steps to build your lab. All commands should be run inside your **WSL/Ubuntu terminal**.

Part 1: [Windows Users] Install WSL This gives you a real Ubuntu terminal on Windows.

1. Open PowerShell or Command Prompt **as an Administrator**.
2. Run the install command:

```
Bash
```

```
wsl --install
```

3. Restart your machine. You will now have an "Ubuntu" app. **Use this Ubuntu terminal for all subsequent steps.**

Part 2: Install Minikube This is your local Kubernetes cluster.

```
Bash
```

```
# 1. Download the latest binary
curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64 [cite: 1308]

# 2. Make it executable
chmod +x minikube [cite: 1310]

# 3. Move it to your system's path
sudo mv minikube /usr/local/bin/ [cite: 1311]
```

Part 3: Install Docker (Minikube's "Driver") Minikube needs a "driver" to create the cluster. The best one is Docker.

- **On Windows:** Install **Docker Desktop**. Go to its **Settings > Resources > WSL Integration** and make sure "Enable integration with my ON.
- **On Mac/Linux:** Install Docker Engine.

Part 4: Start Your Cluster This command tells Minikube to create a new cluster using the Docker driver.

```
Bash
```

```
# Start your cluster
minikube start --driver=docker [cite: 1317]

# Test it
minikube status
# Should show 'Running'
```

Part 5: Install kubectl This is the command-line tool for talking to Kubernetes.

Pro-Tip: Minikube bundles its own version. The easiest way is to create an alias to use it.

```
Bash
```

```
# 1. Create the alias for your current session
alias kubectl="minikube kubectl --" [cite: 1324]

# 2. Add this alias to your .bashrc to make it permanent
echo 'alias kubectl="minikube kubectl --"' >> ~/.bashrc [cite: 1326]

# 3. Reload your bash profile
source ~/.bashrc [cite: 1327]
```



```
# 4. Test it
kubectl get nodes
# Should show your 'minikube' node [cite: 1330]
```

Part 6: Install Helm This is the Helm client tool.

Bash

```
# Download and run the official install script
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash [cite: 1336]

# Test it
helm version
# Should show a v3.x.x version [cite: 1340]
```

Part 7: Install & Configure VS Code This is your code editor.

1. **Install VS Code on Windows/Mac** (not inside WSL).
2. Open VS Code, go to the **Extensions** tab.
3. Install the **"WSL" extension** from Microsoft.
4. **The Magic:** Close VS Code. Go to your **WSL/Ubuntu terminal**, create a project folder (`mkdir helm-projects`), and `cd` into it.
5. In that terminal, type:

Bash

```
code .
```

6. This will re-open VS Code *on Windows*, but it will be **remotely connected** to your WSL Linux environment. The bottom-left corner will
7. Install these extensions (inside VS Code) for Helm development:

- **Kubernetes** (by Microsoft)
- **Helm** (by Microsoft)
- **YAML** (by Red Hat)

🌸 Underpinning Theory

- **kubectl vs. helm** : `kubectl` is the low-level tool for managing individual resources (e.g., `kubectl get pods`). `helm` is the high-level the entire application *lifecycle* (e.g., `helm install wordpress`).
- **VS Code + WSL**: This setup is the professional standard. Your editor (VS Code) runs on your host OS (Windows), but all the tools, files, the correct Linux environment (WSL). The extension creates a seamless bridge.
- **Minikube Driver**: Minikube doesn't run Kubernetes directly. It creates a *node* (a virtual machine or a container) and installs Kubernetes. `driver=docker` flag tells it to create that node as a Docker container, which is fast and lightweight.

🌸 Key Takeaways

- Your complete lab setup is: **VS Code (Editor)** -> **WSL (Linux Environment)** -> **Helm/kubectl (Tools)** -> **Minikube (K8s Cluster)**.
- Always run your `helm` and `kubectl` commands from the **WSL/Ubuntu terminal**, not from Windows CMD or PowerShell.
- Connecting VS Code to WSL with the `code .` command is the key to a smooth development workflow.

Ready to use your new lab? Just let me know when you're ready for **Module 4: Installing Your First Chart**.



Here are the "God-Level" notes for Module 4.

💡 Module 4: Installing Your First Chart (`helm install` , `ls` , `uninstall`)

Core Concept (The "Why")

The core problem is to install a complex application (like MySQL or WordPress) with all its interconnected parts (`StatefulSets` , `Services` manually applying each YAML file .

The solution is to use the `helm install` command. This command takes a pre-packaged **Chart** (e.g., `bitnami/mysql`), creates a unique specific installation), and deploys all the necessary Kubernetes resources automatically.

This module covers the 3 fundamental commands of the Helm user workflow:

1. `helm repo add` (to add a "store")
2. `helm install` (to buy an "item")
3. `helm uninstall` (to return an "item")

⚙️ The Process (The "How")

We will find and install a MySQL chart from the trusted Bitnami repository.

Part 1: Find and Add the Chart Repository Before you can install a chart, you must tell Helm where to find it.

1. **Find the Chart:** Search `artifacthub.io` for "mysql". The Bitnami chart is a popular, trusted choice.
2. **Add the Repo:** Add the Bitnami repository and give it the local "label" `bitnami` . (This is the "Classic" method from Module 2).

Bash

```
# Command: helm repo add [LOCAL_LABEL] [REPO_URL]
helm repo add bitnami https://charts.bitnami.com/bitnami
```

You only need to run this command once per repository.

3. **Update Cache:** Fetch the latest "catalog" from all your added repos.

Bash

```
helm repo update
```

Part 2: Install the Chart This command deploys MySQL to your cluster.

Bash

```
# Command: helm install [RELEASE_NAME] [CHART_NAME]
helm install my-sql-release bitnami/mysql
```

Command Breakdown:

- `helm install` : The action to perform.
- `my-sql-release` : This is your custom **Release Name**. It's *your* name for *this specific installation*. You will use this name to manage (lis this app.
- `bitnami/mysql` : This is the **Chart Name**. `bitnami` refers to the repository label you added, and `mysql` is the name of the chart wit

Part 3: Verify the Installation

1. **Check Helm Releases:** Use `helm list` (or `helm ls`) to see what Helm has installed.

Bash

```
helm list
```

```
# OUTPUT:
```


#	NAME	NAMESPACE	REVISION	UPDATED	STATUS	CHART	APP VERSION
#	my-sql-release	default	1	...	deployed	mysql-9.10.0	8.0.32

This shows your `my-sql-release` is `deployed`.

2. **Check Kubernetes Resources:** Use `kubectl` to see the actual pods Helm created.

```
Bash

kubectl get pods

# OUTPUT:
# NAME                READY   STATUS    RESTARTS   AGE
# my-sql-release-0    1/1     Running   0           2m
```

You can also run `kubectl get all` to see the `StatefulSet` and `Services` Helm created for you.

Part 4: Uninstall the Release This command completely removes the application and all its Kubernetes resources.

```
Bash

# Command: helm uninstall [RELEASE_NAME]
helm uninstall my-sql-release
```

- **CRITICAL TRAP:** Helm will delete the `StatefulSet` and `Pods`, but most database charts are configured to **not** delete the `PersistentVolumeClaim` (PVC) by default. This protects your data.
- You must **manually delete the PVC** if you want to fully clean up:

```
Bash

kubectl get pvc
# (Find the PVC name, e.g., 'data-my-sql-release-0')

kubectl delete pvc data-my-sql-release-0
```

🌸 Underpinning Theory

- **Chart vs. Release:** This is the most important concept.
 - **Chart:** The package; the "recipe" (e.g., `bitnami/mysql`).
 - **Release:** The installed instance; the "cooked meal" (e.g., `my-sql-release`).
- **Release Name:** The name (like `my-sql-release`) is a unique identifier. It's how Helm tracks the set of resources that belong to that s. This is what allows `helm uninstall` to know exactly what to delete.

🌸 Key Takeaways

- The 3 core commands are `helm repo add`, `helm install`, and `helm uninstall`.
- The install command syntax is: `helm install [YOUR_RELEASE_NAME] [REPOSITORY/CHART_NAME]`.
- `helm list` (or `helm ls`) shows all your **Releases**.
- `helm uninstall [YOUR_RELEASE_NAME]` removes a release but may leave PVCs (storage) behind.

Ready for **Module 5: The "Snowflake" Problem & Basic Overrides** (`helm show values`, `--set`)?



Here are the "God-Level" notes for Module 5.

💡 Module 5: The "Snowflake" Problem & Basic Overrides (`helm show values` , `--set`)

Core Concept (The "Why")

This module addresses two problems:

1. **The "How":** A chart's default installation is almost never what you want. How do you find the available configuration options (like passwords) and change them?
2. **The "Why" (CRITICAL):** The `helm install my-sql bitnami/mysql` command we just ran is an **ad-hoc, unreproducible command** because the command creates a **"Snowflake Cluster"**—a unique, manually configured server that is not version-controlled and is impossible to reproduce.

The solution begins with `helm show values` to see the defaults and `--set` to *change* them. This is a basic form of customization, but it's the first step to solving the "Snowflake" anti-pattern. We learn it so we can improve upon it.

⚙️ The Process (The "How")

We'll learn to find the configurable options and change them using the simple `--set` flag.

Part 1: Finding the Default Configuration Every chart comes with a `values.yaml` file that defines its "menu" of default options.

1. **Use `helm show values`:** This command prints the *entire* default `values.yaml` file from the chart's repository.

Bash

```
# Command: helm show values [REPO_LABEL]/[CHART_NAME]
helm show values bitnami/mysql
```

2. **Redirect to a File (Pro-Tip):** The output is often huge. It's much easier to save it to a file for investigation.

Bash

```
helm show values bitnami/mysql > mysql-defaults.yaml
```

3. **Investigate the File:** Open `mysql-defaults.yaml` in VS Code. You can now search for keys. For example, in the `bitnami/mysql` chart,

YAML

```
auth:
  rootPassword: "YOUR_PASSWORD"
  database: "my_database"
  username: "my_user"
  password: "YOUR_USER_PASSWORD"
# ...and hundreds of other options
```

This hierarchical structure (e.g., `auth` -> `rootPassword`) is the most important detail.

Part 2: Overriding Values with `--set` The `--set` flag lets you change one of these values when you install.

1. **Uninstall the Old Release:**

Bash

```
helm uninstall my-sql-release
kubectl delete pvc data-my-sql-release-0 # Don't forget the PVC!
```

2. **Install with Overrides:** Let's set the root password and create a database.

Bash

```
helm install my-sql-release bitnami/mysql \
  --set auth.rootPassword=my-secret-password \
  --set auth.database=my-new-db
```


- **CRITICAL:** You **must** use dot notation for the hierarchy.
 - **MISTAKE:** `--set rootPassword=...` would **not** work.
 - **CORRECT:** `--set auth.rootPassword=...` works because it matches the `values.yaml` structure.
3. **Upgrading an Existing Release:** You can also use `--set` with `helm upgrade` to change a live release.

Bash

```
# (This would change the replicas on a live release)
helm upgrade my-sql-release bitnami/mysql --set primary.replicaCount=2
```

🔴 Underpinning Theory

The "Snowflake Server" Anti-Pattern

- A "Snowflake Server" is a server built manually over time. A sysadmin logs in, installs Apache, and then logs out. A week later, someone applies a patch.
- The problem: The server is not reproducible. It's a "unique snowflake". When it dies, it can't be rebuilt, because its configuration is just versioned, ad-hoc changes.
- **Helm Analogy:** Running `helm install ... --set ...` is the exact same anti-pattern. Your command-line history is not a reproducible plan.

The `values.yaml` File

- This file is **NOT** Kubernetes YAML. It is an *input* for Helm.
- Helm uses this file as a "dictionary" to fill in the blanks in its templates *before* it generates the final Kubernetes YAML.

The `--set` Flag

- `--set` is a simple, inline way to override values.
- It's great for quick, simple tests.
- It's **terrible** for production because it's ad-hoc and not stored in version control.

📌 Key Takeaways

- `helm show values [chart]` is the command to see all default, configurable options for a chart.
- `--set` is the flag to override a default value (e.g., `helm install ... --set auth.rootPassword=...`).
- You **MUST** use dot notation (e.g., `auth.rootPassword`) to match the `values.yaml` hierarchy.
- Relying on `--set` commands creates a "**Snowflake Cluster**", which is a massive anti-pattern.

We will solve the "Snowflake" problem in the next module. Ready for **Module 6: The "Phoenix" Solution: Custom `values.yaml` (`-f` flag**



Here are the "God-Level" notes for Module 6.

💡 Module 6: The "Phoenix" Solution: Custom `values.yaml` (`-f` flag)

Core Concept (The "Why")

This lecture solves the "Snowflake Cluster" problem. Using `helm install --set` is ad-hoc, error-prone, and not stored in version control, unreproducible.

The **core solution** is to stop using `--set` and instead define all your overrides in a **custom `values.yaml` file**. You then use the `-f` (or `-s`) Helm to use this file.

This simple change transforms your process:

- **From:** Imperative (running manual commands).
- **To:** Declarative (defining a state in a file).

This file becomes your **single source of truth** for configuration. When you store this file in Git, you achieve a reproducible, recoverable "Ph

⚙️ The Process (The "How")

We will now install MySQL the *correct* way, making our configuration permanent and repeatable.

Part 1: Create a Minimalist `values.yaml` File

1. **Do NOT** copy the entire 2,500-line default `values.yaml` file. This is a common mistake.
2. Create a **new, empty file** named `my-mysql-values.yaml`.
3. Add **only the keys you want to override**, preserving the hierarchy.

YAML

```
# my-mysql-values.yaml
# This is our entire custom file.

# Override values under the 'auth' key
auth:
  rootPassword: "my-secret-password-from-file"
  database: "my-new-db-from-file"
  username: "my-db-user"

# Override a value under the 'primary' key
primary:
  replicaCount: 1
```

Part 2: Install Using the `-f` (or `--values`) Flag

1. **Uninstall the Old Release (if any):**

Bash

```
helm uninstall my-sql-release
kubectl delete pvc data-my-sql-release-0 # Clean up the PVC
```

2. **Install with the `-f` flag:**

Bash

```
# Command: helm install [RELEASE_NAME] [CHART] -f [YOUR_VALUES_FILE]
helm install my-sql-release bitnami/mysql -f my-mysql-values.yaml
```

Helm will now install the chart using this file for its configuration.

Part 3: Upgrade Using the `-f` Flag This is where the real power lies.

1. **Edit your file:** Open `my-mysql-values.yaml` and change a value.

YAML

```
# my-mysql-values.yaml
...
primary:
  replicaCount: 2 # <-- We changed this from 1 to 2
```

2. **Run `helm upgrade`:**

Bash


```
# Command: helm upgrade [RELEASE_NAME] [CHART] -f [YOUR_VALUES_FILE]
helm upgrade my-sql-release bitnami/mysql -f my-mysql-values.yaml
```

Helm will compare your file to the live state and apply *only* the change, upgrading your pod count to 2. Your cluster state now matches file.

🔴 Underpinning Theory

The "Phoenix Server" (The Solution)

- This is the opposite of a "Snowflake". A Phoenix Server is configured entirely by **automated scripts** (like our `values.yaml` file) stored in Git).
- **The Rule:** No one *ever* logs in to make a manual change. If you need a change, you update the script (your `values.yaml`) and re-run (`helm upgrade`).
- **The Consequence:** The server is **disposable**. You can destroy it at any time, and (like a Phoenix) it can be "reborn from the ashes" *perfectly* just by running the script. This is the goal for your Kubernetes cluster.

How Helm Merges Values When you run `helm install -f my-mysql-values.yaml`:

1. Helm loads the chart's *default* `values.yaml` (the 2,500+ lines).
2. Helm then loads your *custom* `my-mysql-values.yaml` file.
3. It merges your file **on top** of the defaults, overriding any keys that match. This is why your custom file can be minimal and only contain

`--set` vs. `-f`

- `--set` : Imperative. Good for quick, temporary tests. **Bad** for production.
- `-f [file]` : Declarative. The professional, standard way. This file is your **single source of truth** and should be checked into Git .

🔴 Key Takeaways

- **NEVER** use `--set` on a production cluster. Always use the `-f [file]` flag .
- Your custom `values.yaml` file should be **minimalist**—it only needs to contain the values you are changing.
- Store your custom `values.yaml` files in **Git** (source control).
- This workflow (storing config in Git and applying with `helm upgrade -f`) is the **"Phoenix Cluster"** pattern, and it's the professional standard.

Ready for **Module 7: The "Phoenix" Workflow 1: Taking Control with `helm pull`** ?



Here are the "God-Level" notes for Module 7.

💡 Module 7: The "Phoenix" Workflow 1: Taking Control with `helm pull`

Core Concept (The "Why")

In the last module, we solved the "Snowflake" problem by using a `my-values.yaml` file. However, this *still* has a critical flaw: our command `helm upgrade my-values.yaml bitnami/mysql`, relies on a **remote chart**.

This creates a new problem: what happens if Bitnami **deletes** that chart version, or **updates it with a breaking change**? . Our reproducible cluster is suddenly broken by an external force we don't control.

The solution is to **download the chart's source code** to our local machine using `helm pull` . We then store this *exact version* of the chart in our local repository, right alongside our `my-values.yaml` file. This makes our cluster configuration 100% self-contained, reproducible, and recoverable.

⚙️ The Process (The "How")

This is the professional, "Phoenix" workflow.

Part 1: Create a Local Configuration Repository This is the folder you will check into Git.

Bash

```
# 1. Create and enter the directory
mkdir my-cluster-config
cd my-cluster-config

# 2. (Clean up any old releases)
helm uninstall my-sql-release
kubectl delete pvc data-my-sql-release-0
```

Part 2: Download Chart Source Code with `helm pull`

1. **Run** `helm pull` : This command just downloads the chart package. It does not touch your cluster.
2. **Add** `--untar` : By default, it downloads a compressed `.tgz` file. We want to unpack it into a folder, so we use the `--untar=true` flag

Bash

```
# Command: helm pull [REPO_LABEL]/[CHART_NAME] --untar=true
helm pull bitnami/mysql --untar=true
```

3. Inspect the Result:

Bash

```
ls
# OUTPUT:
# my-cluster-config/
# └─ mysql/ <-- This folder is the full chart source
```

This `mysql/` folder (containing `values.yaml`, `templates/`, etc.) is what you check into Git.

Part 3: Install from a Local Chart Now we install from our local, version-controlled copy.

Bash

```
# Command: helm install [RELEASE_NAME] [LOCAL_FOLDER_PATH]
helm install my-sql-release ./mysql
```

This is now 100% reproducible. You can run this script on any new cluster.

Part 4: Apply Custom Overrides (The CORRECT Way) We still need to apply our custom values from `my-mysql-values.yaml`.

1. **CRITICAL MISTAKE TO AVOID:** Do **NOT** edit the `mysql/values.yaml` file directly. This creates the "Divergence Problem" (see below upgrading a nightmare).
2. **THE "GOOD" WAY:** Keep the chart folder *pristine* (unchanged). Use your separate override file, just as before.

Bash

```
# 1. Make sure your my-mysql-values.yaml is in the same folder
# 2. Run helm upgrade, pointing to your local chart and local values file

# Command: helm upgrade [RELEASE] -f [YOUR_FILE] [LOCAL_CHART_PATH]
helm upgrade my-sql-release -f my-mysql-values.yaml ./mysql
```

(We use `upgrade` here, but for a fresh install, `helm install my-sql-release -f my-mysql-values.yaml ./mysql` works perfectly.)

🔴 Underpinning Theory

`helm install [remote]` **vs.** `helm pull` + `helm install ./[local]`

- `helm install ... bitnami/mysql` : An ad-hoc command that *Fetches and Applies* in one step. This is the "Snowflake" pattern. It's ugly because `bitnami/mysql` is a moving target.
- `helm pull` : A deliberate, two-step process.
 1. `helm pull` : *Fetches* (downloads) the chart.
 2. `helm install ./mysql` : *Applies* the chart from a local, trusted, and version-controlled source. This is the "Phoenix" pattern .

The Divergence Problem (Why You NEVER Edit `values.yaml`)

- Editing the chart's default `mysql/values.yaml` is a form of "forking" the chart.
- You now own the maintenance of this divergence. When Bitnami releases a security patch, you can't just pull their new chart; you must manually change into *their* new `values.yaml` .
- Using a separate `-f my-values.yaml` file is the perfect solution. Helm does the "merge" in memory for you, keeping the chart source upgrades trivial.

🔑 Key Takeaways

- **NEVER** use `helm install [remote-chart]` on a production cluster.
- Use `helm pull [chart] --untar=true` to download the chart's source code locally.
- Store your downloaded chart folders and your custom `values.yaml` files in **Git**. This is the foundation of a Phoenix Cluster.
- **CRITICAL:** Do **NOT** edit the chart's default `values.yaml` file. You will create a maintenance nightmare (divergence).
- The complete, professional, and reproducible command is:

```
helm upgrade [release-name] -f [my-override-file.yaml] ./[local-chart-folder] .
```

Ready for **Module 8: The "Phoenix" Workflow 2: Generating Raw YAML with `helm template`** ?



Here are the "God-Level" notes for Module 8.

💡 Module 8: The "Phoenix" Workflow 2: Generating Raw YAML with `helm template`

Core Concept (The "Why")

This module addresses the final step in taking full control. The `helm install ./my-chart` command is reproducible, but it has one "down": **Helm to be installed** on any machine (or CI/CD pipeline) that deploys to the cluster.

The solution is the `helm template` command. This command runs the *entire* chart engine locally, but instead of applying it to a cluster, it generates **raw Kubernetes YAML** and prints it to your console.

You can then save this raw YAML (which can be thousands of lines) to a file and deploy it using only `kubectl apply` , removing Helm as a cluster dependency.

⚙️ The Process (The "How")

This process converts a Helm chart into a single, deployable YAML file.

Part 1: Generate Raw YAML

1. **Understand the Command:** The `helm template` command has the *exact same syntax* as `helm install` . It just does "everything except apply".
 - **Syntax:** `helm template [RELEASE_NAME] [CHART_PATH] -f [VALUES_FILE]`
2. **Execute the Command:** We'll run the same command we used for `helm install` in the last module, but with `template` instead.

Bash

```
# (Inside your my-cluster-config/ folder)
```



```
helm template my-sql-release ./mysql -f my-mysql-values.yaml
```

This will print a *massive* (thousands of lines) stream of pure Kubernetes YAML to your console.

3. **Capture the Output:** To make this useful, redirect the output (`>`) into a new file.

Bash

```
helm template my-sql-release ./mysql -f my-mysql-values.yaml > my-sql-stack.yaml
```

You now have a single file, `my-sql-stack.yaml`, containing all the `StatefulSets`, `Services`, `ConfigMaps`, and `Secrets` needed f

Part 2: Deploy Using Only `kubectl`

1. **Clean the Cluster (for a fresh start):**

Bash

```
helm uninstall my-sql-release
kubectl delete pvc data-my-sql-release-0 # Clean up the PVC
```

2. **Apply the Generated File:** You can now deploy this on *any* Kubernetes cluster, even one that doesn't have Helm installed.

Bash

```
kubectl apply -f my-sql-stack.yaml
```

3. **Verify the Result:**

Bash

```
kubectl get pods
# OUTPUT:
# NAME                READY   STATUS    RESTARTS   AGE
# my-sql-release-0    1/1     Running   0           2m
```

The end result in the cluster is **identical** to what `helm install` would have done.

🔴 Underpinning Theory

This creates a critical trade-off for your team: **What do you store in Git?**

- **Approach 1: Store the "Source" (The Helm Way)**
 - **What you store:** The `mysql/` folder and your `my-mysql-values.yaml` file.
 - **Pros:** Easy to manage and upgrade. To change a port, you edit one line in `my-mysql-values.yaml`.
 - **Cons: Helm is a dependency.** Your CI/CD pipeline *must* have `helm` installed.
- **Approach 2: Store the "Binary" (The `kubectl` Way)**
 - **What you store:** Only the single `my-sql-stack.yaml` (thousands of lines).
 - **Pros: No Helm dependency.** You only need `kubectl`.
 - **Cons: A maintenance nightmare.** You've "lost" the source. To change a port, you must find and edit the correct line in a 10,000-impossible.
- **Approach 3: Store Both (The "Hybrid" Way)**
 - **What you store:** *Both* the source (`mysql/`, `my-mysql-values.yaml`) *and* the generated `my-sql-stack.yaml`.
 - **Pros:** Best of both worlds. For a quick deploy, you just `kubectl apply`. To make a change, you edit `my-mysql-values.yaml`, re-r and commit both files.
 - **Cons:** Data duplication in Git.

All three are valid strategies, but they are all infinitely better than the "Snowflake" (remote install) anti-pattern.

★ Key Takeaways

- `helm template` generates the final, raw Kubernetes YAML without applying it.
- The command syntax is identical to `helm install : helm template [release] [chart-path] -f [values-file]`.
- You can redirect the output (`> my-stack.yaml`) to create a single, massive YAML file.
- This file can be deployed using `kubectl apply` , which removes Helm as a deployment dependency.
- This creates a **trade-off**: you gain independence from Helm but lose the easy abstraction of `values.yaml` .

Ready for **Module 9: Managing Live Releases: Upgrades & Rollbacks** (`helm upgrade` , `history` , `rollback`)?



Here are the "God-Level" notes for Module 9.

💡 Module 9: Managing Live Releases: Upgrades & Rollbacks (`helm upgrade` , `history` , `rollback`)

Core Concept (The "Why")

The core problem is that applications are not static. You will always need to:

1. **Change Configuration:** Update a setting (like `replicaCount`) in your application.
2. **Upgrade Version:** Deploy a new version of the application software (e.g., a security patch).

You need a way to apply these changes to a live, running release. More importantly, if an upgrade fails, you need a safe, instant "undo" butt last working version.

Helm's solutions are the `helm upgrade` command to apply changes and the `helm rollback` command to undo them.

⚙️ The Process (The "How")

This is the entire lifecycle of managing a live application.

Part 1: Upgrading Configuration (Changing `values.yaml`) Let's say your app is running with `replicaCount: 2` and you want to scale it

1. **Edit your file:** Open your version-controlled `my-values.yaml` and change the value:

YAML

```
# my-values.yaml
...
replicaCount: 3 # <-- We changed this from 2 to 3
```

2. **Run `helm upgrade` :**

Bash

```
# Command: helm upgrade [RELEASE_NAME] [CHART] -f [YOUR_VALUES_FILE]
helm upgrade my-app ./my-app-chart -f my-values.yaml

# OUTPUT:
# Release "my-app" has been upgraded. Happy Helming!
```

Helm will calculate the "diff" and tell Kubernetes to scale the `Deployment` from 2 to 3 pods. This is the core command for all CI/CD.

Part 2: Upgrading the Chart Version A new version of the chart (e.g., `v2.0.0`) is released with a security patch.

1. **Run `helm upgrade` with the `--version` flag:**

Bash

```
# (Assuming we are using the remote chart for this example)
# Command: helm upgrade [RELEASE] [CHART] -f [VALUES_FILE] --version [VERSION]
helm upgrade my-blog bitnami/wordpress \
  -f my-blog-values.yaml \
  --version 28.0.0
```

Helm will now pull the new chart, apply your *same* custom `my-blog-values.yaml` to it, and deploy the new version.

Part 3: Checking Release History "Oh no! The `28.0.0` upgrade broke the site!" Before we can fix it, we must see what happened.

1. Run `helm history` :

Bash

```
helm history my-blog
```

2. Analyze the Output:

#	REVISION	UPDATED	STATUS	CHART	APP VERSION	DESCRIPTION
#	1	...	deployed	wordpress-27.1.3	6.4.2	Install complete
#	2	...	deployed	wordpress-28.0.0	6.5.0	Upgrade complete

We can see **REVISION 2** is the new, broken one. The last working state was **REVISION 1**.

Part 4: Rolling Back the Failed Upgrade

1. Run `helm rollback` : You must specify the revision you want to *go back to*.

Bash

```
# Command: helm rollback [RELEASE_NAME] [TARGET_REVISION]
helm rollback my-blog 1

# OUTPUT:
# Rollback was a success! Happy Helming!
```

2. Check History Again:

Bash

```
helm history my-blog
```

#	REVISION	UPDATED	STATUS	CHART	APP VERSION	DESCRIPTION
#	1	...	superseded	wordpress-27.1.3	6.4.2	Install complete
#	2	...	superseded	wordpress-28.0.0	6.5.0	Upgrade complete
#	3	...	deployed	wordpress-27.1.3	6.4.2	Rollback to 1

Your site is now instantly back to the working version. Notice Helm created **REVISION 3**, which is a *new* revision that is a perfect copy

🌸 Underpinning Theory

- **Revisions:** Helm is **stateful**. Every install, upgrade, or rollback creates a new, numbered **Revision**. Helm stores these revisions as `Sec` your cluster, which is how it keeps track of the history.
- `helm upgrade` is **Declarative**: This command tells Helm "make the cluster look like this". It calculates the difference between the run desired state (from your files) and applies only the changes.
- **Critical CI/CD Flags:** These are flags you will use in all your automated pipelines.
 - `--install` : A "lifesaver" flag. It means "Run `helm upgrade` , but if the release doesn't exist, run `helm install` instead". This is or insert) and lets you use one single command for all deployments.

- `--atomic` : This makes your upgrade "transactional". If the upgrade fails, Helm will *automatically* roll back to the last good revision cluster from being left in a broken, half-upgraded state.
- `--dry-run` : "Don't actually do anything. Just print out the final YAML you would send to Kubernetes". This is invaluable for debugging.
- **The Ultimate CI/CD Command:** This single, safe command can be run over and over in a pipeline.

Bash

```
helm upgrade --install [RELEASE] [CHART_PATH] -f [VALUES_FILE] --atomic
```

★ Key Takeaways

- `helm upgrade` is the one command you use to apply *any* change (new values or new chart version) to a live release.
- `helm history [release]` shows you the "undo" log (all the revisions).
- `helm rollback [release] [revision]` is your "undo" button to revert to a previous good state.
- For CI/CD, always use `helm upgrade --install ... --atomic` for safe, repeatable, "upsert" deployments.

Ready for **Module 10: Why Write Your Own Chart? (The "Configuration Drift" Problem)?**



Here are the "God-Level" notes for Module 10.

💡 Module 10: Why Write Your Own Chart? (The "Configuration Drift" Problem)

Core Concept (The "Why")

This lecture identifies the **single biggest missing feature in Kubernetes**: its YAML files are **static**. They are just data; they have no logic, no logic statements.

This creates a massive, common problem for managing different environments (e.g., dev, staging, production). These environments are almost always small, critical differences (like a different image tag, a lower replica count, or a test database URL).

The "Kubernetes-native" solution is to **duplicate files** (e.g., `deployment-dev.yaml`, `deployment-prod.yaml`). This is a **terrible anti-pattern** called **"Configuration Drift"**—a "right mess" where a bug fix applied to one file is forgotten in the other.

The solution is to use Helm's powerful **templating engine** for your *own* internal applications. By turning your app into a Helm chart, you can manage all environment differences with `values.yaml` files, keeping your code DRY (Don't Repeat Yourself).

⚙️ The Process (The "How")

This lecture demonstrates the problem by manually deploying two different versions of the same application.

Part 1: The Starter YAML We start with a single, static YAML file (`fleetman-start.yaml`) that contains one `Deployment` and one `Service`.

YAML

```
# fleetman-start.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fleetman-webapp
spec:
  replicas: 1
  template:
    spec:
      containers:
      - name: fleetman-webapp
        # This is the line we will change
        image: richardchesterwood/fleetman-helm-demo:latest
```



```
...
apiVersion: v1
kind: Service
metadata:
  name: fleetman-webapp
spec:
  type: NodePort
  ports:
    - port: 80
      nodePort: 30080
  ...
```

Part 2: Deploy the "Production" Version

1. Apply the file:

Bash

```
kubectl apply -f fleetman-start.yaml
```

2. Verify: Accessing the app in a browser (`http://[MINIKUBE_IP]:30080`) shows the standard "Production" map.

Part 3: Manually Deploy the "Development" Version This demonstrates the "ugly," manual way of managing environments.

1. Manually edit `fleetman-start.yaml`:

- **From:** `image: richardchesterwood/fleetman-helm-demo:latest`
- **To:** `image: richardchesterwood/fleetman-helm-demo:latest-dev`

2. Re-apply the file:

Bash

```
kubectl apply -f fleetman-start.yaml
```

3. Verify: After a hard-refresh, the browser now shows a "DEV Version" banner.

We successfully made the change, but we had to **manually edit a file** to do it. This is the "pain" that Helm's templating will solve.

🧠 Underpinning Theory

Static YAML vs. Dynamic Configuration

- **Static YAML (The Problem):** Kubernetes YAML files are just data. To change anything, you *must* change the file itself.
- **Dynamic Configuration (The Goal):** We want to *parameterize* our YAML. We want a template where we can just "pass in" the image tag and Helm will generate the YAML for us.

The "Horrible Duplication" Anti-Pattern The obvious (and wrong) solution is to have two files:

- `fleetman-prod.yaml` (with `...:latest`)
- `fleetman-dev.yaml` (with `...:latest-dev`)

The Problem: Configuration Drift

- **What is Drift?** In six months, a developer will find a bug and add a new `env:` variable to `fleetman-dev.yaml` . They will *forget* to add `prod.yaml` .
- **The Consequence:** The two files have now "drifted" apart. When the dev code is promoted to production, it will fail because the `env` is missing.

Helm as a Templating Engine This is the key insight. You don't have to be distributing your software to the world to benefit from Helm. You can use Helm internally for your *own* projects. By converting your static YAML into a Helm chart, you can maintain:

- **One** `deployment.yaml` template.
- **Two** values files: `prod-values.yaml` and `dev-values.yaml` .

This follows the **DRY (Don't Repeat Yourself)** principle and completely eliminates configuration drift.

★ Key Takeaways

- The single biggest "missing feature" in Kubernetes is that its **YAML files are static**.
- Duplicating YAML files for different environments is a critical anti-pattern that leads to **Configuration Drift**.
- The **real value of Helm** for most projects is not package distribution; it's using its **templating engine** for your own internal applicator
- This lecture's demo (manually changing the image tag) is designed to make you feel the "pain" that Helm solves.

Ready for **Module 11: Creating Your First Chart** (`helm create , Chart.yaml , {{ .Values }}`)?



Here are the "God-Level" notes for Module 11.

💡 Module 11: Creating Your First Chart (`helm create , Chart.yaml , {{ .Values }}`)

Core Concept (The "Why")

This lecture solves the "static YAML" problem from Module 10. The goal is to convert our static `fleetman-start.yaml` file into a dynamic t

The core concept is that Helm uses the **Go (Golang) templating engine**. Any file in a chart's `templates/` folder is processed by this engi
replace hard-coded values (like `replicas: 1`) with **variables** (called "actions," e.g., `{{ .Values.replicaCount }}`).

This gives us the power to create a single, dynamic YAML template that can be configured for different environments, finally solving the cor
problem .

⚙️ The Process (The "How")

We will scaffold a new chart, clean it, add our static YAML, and make our first value dynamic.

Part 1: Create the Chart Skeleton Helm provides a "scaffolding" command to create the required directory structure.

1. **Run `helm create`** : This command generates a new chart folder with a pre-filled set of example files.

Bash

```
helm create fleetman-chart
```

2. **Inspect the new directory:**

Bash

```
ls fleetman-chart
# OUTPUT:
# Chart.yaml           # Metadata about your chart
# charts/              # For sub-chart dependencies (empty)
# templates/          # The heart of the chart (full of examples)
# values.yaml          # Default configuration values
# .helmignore          # Files to ignore when packaging
```

Part 2: Clean the Skeleton (The "Blank Canvas") The default example files are overwhelming. We will delete them to start from scratch.

1. **Clean the `templates/` folder:**
 - Navigate into `fleetman-chart/templates/` .
 - Delete all files inside it (e.g., `deployment.yaml` , `service.yaml` , `_helpers.tpl` , `NOTES.txt`).
 - You should now have an empty `templates/` folder.
2. **Clean the `values.yaml` file:**

- Open the root `values.yaml` file.
- Delete all the content inside it.
- Save the now-empty file (we keep the file itself).

Part 3: Add Our Static YAML Now, we add our application's YAML, which will become our first template.

1. **Create a new template file:** Inside the empty `templates/` folder, create a new file (e.g., `app.yaml`).
2. **Copy the static YAML:** Copy the *entire contents* of your `fleetman-start.yaml` (from Module 10) and paste it into this new `app.yaml`

Part 4: Make It Dynamic (Parameterizing Replicas) This is the central step. We will replace the hard-coded `replicas: 1` with a dynamic

1. **Define a Default Value (in `values.yaml`):**

- Open your empty `values.yaml` file.
- Add the following key. This defines the default value for our replica count.

YAML

```
# values.yaml
replicaCount: 1
```

2. **Reference the Value (in `templates/app.yaml`):**

- Open your `templates/app.yaml` file.
- Find the `replicas: 1` line.
- Replace the static `1` with this Go template "action":

YAML

```
# templates/app.yaml
...
spec:
  replicas: {{ .Values.replicaCount }}
...
```

Part 5: Test the Template (Locally) We can test our template without deploying it using `helm template` from inside the chart's root directory (`chart/`).

1. **Test the Default Value:**

Bash

```
# The . tells Helm to use the chart in the current directory
helm template .

# OUTPUT (abbreviated):
# ...
# spec:
#   replicas: 1 <-- It correctly used the default from values.yaml
# ...
```

2. **Test an Override (with `--set`):**

Bash

```
helm template . --set replicaCount=5

# OUTPUT (abbreviated):
# ...
# spec:
#   replicas: 5 <-- It correctly used our override
# ...
```


🧠 Underpinning Theory

Chart Structure (Key Components)

- `Chart.yaml` : Required metadata file. It contains the chart's `name` , `description` , `type` (`application` or `library`), `version` (the chart's version) and `appVersion` (your application's version) .
- `values.yaml` : Defines all the default configurable values for your chart. This is the "menu" of options.
- `templates/` : The heart of the chart. All `.yaml` files in this folder are processed by the Go template engine and combined into one large file that is sent to Kubernetes.
- `charts/` : Used to bundle other charts as dependencies (subcharts).

The Go (Golang) Templating Engine

- The files in `templates/` are not plain YAML; they are Go templates that *generate* YAML.
- **Action Syntax:** `{{ ... }}` . Anything inside double curly brackets is executed as Go template code.

The `.Values` Object: The Bridge

- This is the most important concept of this lecture. Helm provides a built-in object, `{{ .Values }}` , which is available inside all templates.
- This object contains the final, merged configuration from `values.yaml` and any `--set` or `-f` overrides .
- You use dot notation to access its keys.
- **Example:** `{{ .Values.replicaCount }}` accesses the `replicaCount` key we defined in our `values.yaml` . The `.` at the start (`.Values`) is the top-level scope.

📌 Key Takeaways

- Use `helm create [chart-name]` to scaffold the standard chart directory structure.
- It's best to **delete all example files** from `templates/` and `values.yaml` to start clean.
- The `{{ .Values }}` object is the bridge that lets you access data from your `values.yaml` file inside your templates.
- `{{ .Values.myKey }}` in a template will be replaced by the `myKey:` value from `values.yaml` .
- Always use `helm template .` to test your chart locally and see the generated YAML before you install it.

Ready for **Module 12: Go Template Deep Dive 1: Functions & Pipelines** (`|` , `default` , `quote`)?



Here are the "God-Level" notes for Module 12.

💡 Module 12: Go Template Deep Dive 1: Functions & Pipelines (`|` , `default` , `quote`)

Core Concept (The "Why")

This lecture addresses the limitations of simple value substitution.

1. **What if a value is missing?** Our template `{{ .Values.myMissingKey }}` will inject a `nil` (empty) value, which crashes the templating engine.
2. **What if a value is in the wrong format?** If a value is `hello world` (with a space), just injecting it into YAML might break the file's syntax.

The solution is to use Helm's built-in **Go template functions** (like `default` and `quote`) to process our data. The most "Helm-like" way to do this is by chaining them together using **pipelines** (the `|` symbol) .

⚙️ The Process (The "How")

This process shows the evolution from a simple, brittle substitution to a robust, safe pipeline.

Part 1: Parameterize the Docker Image First, let's parameterize the Docker image tag in our `templates/app.yaml` .

1. `values.yaml` : Define the default value.

YAML

```
# values.yaml
replicaCount: 1

image:
  repository: "richardchesterwood/fleetman-helm-demo"
  tag: "latest"
```

2. `templates/app.yaml` : Use the values in the template. Note how we combine two values with static text.

YAML

```
# templates/app.yaml
...
containers:
- name: fleetman-webapp
  image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
...
```

3. **Test:** `helm template .`
 - **Result:** `image: "richardchesterwood/fleetman-helm-demo:latest"` (It works!)

Part 2: The `nil` Problem (Handling Missing Values) This is a critical, real-world scenario.

1. **Simulate the Problem:** Remove the `tag: "latest"` line from your `values.yaml`.
2. **Test:** `helm template .`
 - **Result: CRASH!** You get an error like `Error: ... at <.Values.image.tag>: nil pointer evaluating...`. This is because `{{ }}` evaluates to `nil`, and the template engine can't process it.

Part 3: Solution 1: The `default` Function This is our "defensive coding" to solve the `nil` problem.

1. `templates/app.yaml` (**Traditional Syntax**):

YAML

```
# Syntax: {{ default "fallback-value" .Values.theValue }}
image: "{{ .Values.image.repository }}:{{ default "latest" .Values.image.tag }}"
```

2. **Test (with `tag` missing):** `helm template .`
 - **Result:** `image: "richardchesterwood/fleetman-helm-demo:latest"` (No crash!). The `default` function caught the `nil` value `"latest"` instead.

Part 4: Solution 2: The Pipeline (`|`) and `quote` Function The traditional syntax is ugly. The "pipeline" (`|`) is the preferred, more readable functions. We also add the `quote` function to ensure our final string is safely wrapped in quotes.

1. `templates/app.yaml` (**Chained Pipeline**):

YAML

```
# The value comes first and is "piped" into the functions.
image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default \"latest\" | quote }}"
```

2. **Test (with `tag` missing):** `helm template .`
 - **Result:** `image: "richardchesterwood/fleetman-helm-demo":"latest"` (It works and is safely quoted).

How the Chained Pipeline Works:

1. `{{ .Values.image.tag }}` is evaluated (result is `nil`).
2. `nil` is "piped" (`|`) as input to `default "latest"`.
3. The `default` function outputs the string `"latest"`.

4. `"latest"` is "piped" (`|`) as input to `quote` .
5. The `quote` function outputs `"latest"` (with quotes).

🌸 Underpinning Theory

- **Go Function Syntax:** There are two ways to call a function, and they are identical:

1. **Traditional:** `{{ default "foo" .Values.key }}`
2. **Pipeline:** `{{ .Values.key | default "foo" }}`

- **The Pipeline (`|`):** This symbol is pure syntactic sugar. It takes the value on its left and passes it as the *last* argument to the function (`default`).
- **Pipeline Chaining:** The true power is chaining. The chain is executed from **left to right**. The output of one function becomes the input to the next.
- **The `default` Solution:** The `default` function is specifically designed to solve the `nil` problem. It checks if the incoming value is `nil`. If so, outputs the default string instead, "sanitizing" the pipeline .
- **The `quote` Function:** This is critical for correctness. `image: {{ .Values.image.tag }}` is dangerous. `image: {{ .Values.image.tag | quote }}` is safe, as it ensures the final output is always a valid YAML string.

🌸 Key Takeaways

- Functions (like `default` , `upper` , `quote`) allow you to manipulate data inside your templates.
- The **pipeline** (`|`) symbol is the preferred, "Helm-like" way to use functions .
- **Always code defensively.** If a value could be missing, use the `default` function to provide a fallback and prevent your template from failing.
- Always use the `quote` function on values that should be strings in YAML.

Ready for **Module 13: Go Template Deep Dive 2: Flow Control** (`if/else` , `eq` , `with`)?



Here are the "God-Level" notes for Module 13.

💡 Module 13: Go Template Deep Dive 2: Flow Control (`if/else` , `eq` , `with`)

Core Concept (The "Why")

This lecture solves our primary goal from Module 10: dynamically changing the image tag based on an environment. Simple substitution (`{{ .Values.image.tag }}`) fails because the *logic* is conditional:

- **IF** we are in "development", we want the tag `...:latest-dev` .
- **ELSE**, we want the tag `...:latest` .

The solution is to use Go template's **flow control blocks** (`{{ if }}` , `{{ else }}` , `{{ end }}`). These are "actions" that control *whether* rendered, allowing you to build if/then/else logic directly into your YAML.

⚙️ The Process (The "How")

This process shows the evolution from a failed string substitution to a robust, boolean-based `if` block.

Part 1: The Failed "Substitution" Attempt This shows *why* we need `if` blocks.

1. `values.yaml` :

YAML

```
environment: prod
```

2. `templates/app.yaml` :

YAML

```
# This logic is wrong
image: "...:latest-{{ .Values.environment }}"
```

3. **Test:** `helm template .`

- **Result:** `image: "...:latest-prod"`
- **Problem:** This is wrong. We wanted `...:latest`. This proves we need conditional logic, not simple substitution.

Part 2: The `if` Statement (String Comparison) This is the correct solution. We only render the `-dev` text if our condition is met.

1. `values.yaml`:

YAML

```
environment: dev
```

2. `templates/app.yaml`:

YAML

```
# The 'if' block wraps the static text we want to conditionally render.
# Note: Go templates use 'eq' for '=='
image: "...:latest{{ if eq .Values.environment \"dev\" }}-dev{{ end }}"
```

3. **Test 1** (`environment: dev`): `helm template .`

- `eq .Values.environment "dev"` evaluates to **true**.
- The `-dev` text is rendered.
- **Result:** `image: "...:latest-dev"` (Correct!)

4. **Test 2** (`environment: prod`): `helm template .`

- `eq .Values.environment "dev"` evaluates to **false**.
- The entire block (and the `-dev` text) is skipped.
- **Result:** `image: "...:latest"` (Correct!)

Part 3: Refining with Booleans (The Best Way) Using strings ("dev", "prod") is brittle. A boolean (`true` / `false`) is much safer.

1. `values.yaml`: (Change the key to be a boolean)

YAML

```
# 'yes' or 'true' are valid YAML for true
development: true
```

2. `templates/app.yaml`: (Simplify the `if` statement)

YAML

```
# You don't need 'eq true'. The 'if' block works directly on booleans.
image: "...:latest{{ if .Values.development }}-dev{{ end }}"
```

3. **Test 1** (`development: true`): `helm template .`

- **Result:** `image: "...:latest-dev"` (Correct!)

4. **Test 2** (`development: false`): `helm template .`

- **Result:** `image: "...:latest"` (Correct!)

Part 4: The `with` Block (Simplifying Scope) The `with` block simplifies templates by reducing repetition. It's like an `if` block that also

- The "Ugly" Way (Without `with`):

YAML

```
# Repetitive and hard to read
env:
- name: DB_HOST
  value: {{ .Values.database.host | quote }}
- name: DB_PORT
  value: {{ .Values.database.port | quote }}
- name: DB_USER
  value: {{ .Values.database.user | quote }}
```

- The "Clean" Way (With `with`):

YAML

```
env:
{{- with .Values.database }}
# Inside this block, '.' is now '.Values.database'
- name: DB_HOST
  value: {{ .host | quote }}
- name: DB_PORT
  value: {{ .port | quote }}
- name: DB_USER
  value: {{ .user | quote }}
{{- end }}
```

This is much cleaner. If `.Values.database` is `nil`, the entire block is skipped.

🧠 Underpinning Theory

- **Actions vs. Static Text:** `{{ ... }}` blocks are Go Template "Actions" that are *executed*. Everything outside them (like `-dev`) is *static*. An `{{ if }}` block is an action that controls whether the static text following it is printed.
- **Operators are Functions:** Go templates do not have operators like `==`, `!=`, or `&&`. You must use built-in functions:
 - `eq`: Equal (`==`)
 - `ne`: Not Equal (`!=`)
 - `lt`: Less Than (`<`)
 - `gt`: Greater Than (`>`)
 - `and`: Logical AND (`&&`)
 - `or`: Logical OR (`||`)
 - **Syntax:** `{{ if eq .Values.key "value" }}`
- **YAML Type Safety:** `development: true` (or `yes`) is a **Boolean**. `development: "true"` is a **String**. The `{{ if .Values.development }}` expects a boolean. If it gets a string, it will crash. This is a *feature* that enforces correct types.
- **Whitespace Control:** Notice the `{{- with ... }}` and `{{- end }}`. The hyphen (`-`) "chomps" whitespace. `{{-` removes whitespace *before* the action, and `-}}` removes it *after*. This is critical for generating clean, valid YAML.

📌 Key Takeaways

- Flow control is the real power of Helm, letting you manage dev vs. prod logic without duplicating files.
- The primary control block is `{{ if CONDITION }}...{{ end }}`. The text inside is only rendered if `CONDITION` is true.
- Go templates use functions for operators. The most common is `eq` (e.g., `{{ if eq .Values.env "dev" }}`).
- It is safer to use **booleans** (`development: true`) for `if` statements rather than comparing strings. The check is simpler: `{{ if .Values.development }}`.
- Use `{{ with .Values.myblock }}` to change the scope and avoid repeating `{{ .Values.myblock.key1 }}`, `{{ .Values.myblock.key2 }}`, etc.

Ready for **Module 14: Go Template Deep Dive 3: Iteration with `range`**?



Here are the "God-Level" notes for Module 14.

💡 Module 14: Go Template Deep Dive 3: Iteration with `range`

Core Concept (The "Why")

This lecture solves a new problem: **how do you dynamically generate *multiple* YAML blocks from a list of items in your `values.yaml`?**

For example, what if you want to let a user provide a list of environment variables, host aliases, or annotations, and you need to create a new block for each one? You can't use a simple `{{ if }}` block because you don't know how many items there will be.

The solution is the `{{ range }}` action. This is Helm's "for loop". It iterates over a list or a map (dictionary) and renders a block of template, allowing you to build dynamic, data-driven YAML structures.

⚙️ The Process (The "How")

We will explore the two main uses for `range`: iterating over a **list** and iterating over a **map**.

Part 1: Iterating Over a List (e.g., Host Aliases) This is the simplest form of a loop.

1. `values.yaml`: Define a list of hostnames.

YAML

```
# values.yaml
additionalHosts:
  - "host1.example.com"
  - "host2.example.com"
```

2. `templates/deployment.yaml`: Use `{{ range }}` to loop over the list.

YAML

```
...
spec:
  template:
    spec:
      hostAliases:
        {{- range .Values.additionalHosts }}
        - host: {{ . | quote }}
          ip: "127.0.0.1"
        {{- end }}
...

```

- `{{- range .Values.additionalHosts }}`: This starts the loop.
- **CRITICAL:** Inside the loop, the dot (`.`) **changes scope**. It no longer refers to the root `Values` object; it now refers to the *current* item (e.g., `"host1.example.com"`).
- `{{ . | quote }}`: This prints the current item (the dot) and pipes it to `quote`.
- `{{- end }}`: This closes the loop.

Part 2: Iterating Over a Map (e.g., Annotations) This is slightly more complex, as you get a key and a value for each item.

1. `values.yaml`: Define a map of key-value pairs.

YAML


```
# values.yaml
myAppAnnotations:
  key1: "value1"
  "my.domain/key2": "value2"
```

2. `templates/deployment.yaml`: Use `{{ range }}` with `$key` and `$val` variables.

YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
  annotations:
    {{- range $key, $val := .Values.myAppAnnotations }}
      {{ $key }}: {{ $val | quote }}
    {{- end }}
spec:
  ...
```

- `{{- range $key, $val := .Values.myAppAnnotations }}`: This is the syntax for looping over a map. For each item, it assigns the value to `$val`.
- `{{ $key }}`: `{{ $val | quote }}`: We can now use these variables to print the YAML. This is incredibly powerful for adding dynamic annotations.

🔴 Underpinning Theory

- **Scope Change (The `.`):** This is the most confusing part of `range`. As soon as you enter a `range` or `with` block, the meaning of the `.` changes.
 - **Outside the loop:** `.` is the top-level scope (which has `.Values`, `.Release`, etc.).
 - **Inside a list loop:** `.` is the current list item (e.g., a string).
- **The Problem:** What if you need to access `.Values` *inside* the loop?
 - **The Solution:** You must save the root scope to a variable *before* you start the loop.

YAML

```
{{- $root := . -}} # Save the top-level scope to $root
{{- range .Values.myList }}
  # '.' is now the list item
  # But $root.Values is still accessible
  name: {{ . | quote }}
  chart: {{ $root.Chart.Name | quote }}
{{- end }}
```

Any variable defined with `:=` and starting with a dollar sign (`$`) is globally accessible.

- **Whitespace Control (`-`):** The hyphens (`{{-` and `-}}`) are critical inside `range` blocks. They "chomp" the newlines and spaces, preventing from generating ugly, invalid YAML filled with empty lines.

🔴 Key Takeaways

- The `{{ range }}` action is Helm's "for loop" for iterating over lists and maps.
- To loop a **list**: `{{ range .Values.myList }}`. Inside, `.` is the current item.
- To loop a **map**: `{{ range $key, $val := .Values.myMap }}`. Use `$key` and `$val` inside.
- **CRITICAL:** `range` changes the scope of the dot (`.`).
- If you need to access the root scope inside a loop, save it to a variable first: `{{- $root := . -}}`.

Ready for **Module 15: Reusable Code: Named Templates** (`_helpers.tpl`, `define`, `include`, `indent`)?



Here are the "God-Level" notes for Module 15.

💡 Module 15: Reusable Code: Named Templates (`_helpers.tpl` , `define` , `include` , `indent`)

Core Concept (The "Why")

This lecture addresses a major limitation of our current templates. We know how to reuse single *values* (e.g., `{{ .Values.replicaCount }}`). **reuse complex blocks of YAML?**

For example, our `image:` block, with its `if` logic, is getting complex. What if we need to use this exact same image logic in 10 different files? We can't put a block of YAML in `values.yaml`.

The solution is a **"Named Template"** (also called a "partial" or "sub-template"). This allows you to `define` a reusable block of template code and then `include` it anywhere you need it. This is the DRY (Don't Repeat Yourself) principle for your YAML templates.

⚙️ The Process (The "How")

This is a multi-step process of refactoring our code. It involves creating a "library" file, defining a template, and then calling it correctly.

Part 1: Create the "Library" File (`_helpers.tpl`)

1. **Create a New File:** In the `templates/` folder, create a new file.
2. **CRITICAL NAMING:** You **must** name this file with an **underscore (`_`) prefix**. The convention is `_helpers.tpl`.
 - The `_` prefix tells Helm: "This file is a library. Parse it for templates, but **do not** try to render its contents as a standalone Kubernetes manifest."
 - The `.tpl` (template) extension is a convention to show it's not a standard `.yaml` file.

Part 2: Define the "Named Template" Inside `_helpers.tpl`

Inside `_helpers.tpl`, we wrap our YAML block in a `define` action.

1. **Cut the Reusable Block:** Go to `templates/app.yaml` and cut the block of code you want to reuse (e.g., the `name:` and `image:` lines under `containers:`).
2. **Paste and `define`:** Paste this block into `_helpers.tpl` and wrap it.
 - **CRITICAL:** The YAML block must be defined with **no indentation** (flush to the left) to be reusable.
 - We use `{{- define ... -}}` with hyphens to "chomp" whitespace, keeping the YAML clean.

Go

```
{{/* templates/_helpers.tpl */}}

{{/*
Define a reusable block for the webapp image.
We'll call it "fleetman.webapp-image".
*/}}
{{- define "fleetman.webapp-image" -}}
# This YAML is defined with NO indentation
name: fleetman-webapp
image: "{{.Values.image.repository}}:{{.Values.image.tag | default "latest" | quote}}{{ if .Values.development }}-dev{{ end }}"
{{- end -}}
```

Part 3: Call the Template (The `include / indent` Pattern)

Now, we go back to `templates/app.yaml` and call our named template.

1. **`include` Action:** We use the `include` action to call our template by its name (`"fleetman.webapp-image"`).
2. **`.` (Dot) Context:** We pass `.` (a dot) as the second argument. This is crucial. It passes the entire scope (including `.Values`) into our template.
3. **`indent` Function:** Since our template has no indentation, we must pipe its output to the `indent` function to match the parent's YAML structure.

YAML

```
# templates/app.yaml
...

```



```
spec:
  containers:
    # This one line replaces our entire complex block
    {{- include "fleetman.webapp-image" . | indent 6 }}
    ...
```

Part 4: Verification

- Run `helm template . --set development=true`.
- The output YAML will be perfectly formatted. Helm:
 1. Calls `include "fleetman.webapp-image"`, which renders the (unindented) YAML block.
 2. Pipes that block to `indent 6`, which indents *every line* by 6 spaces.
 3. Places that result perfectly into the `containers:` section.

🌸 Underpinning Theory

- **The `_` Prefix:** Any file in `templates/` starting with `_` is a "partial" or "library". Helm parses it for `define` blocks but does not try to resource.
- `{{- define "name" -}}`: This is the action to create a named template. The name is a string you will use to call this block. It's common to use a unique name (e.g., `"mychart.mycomponent.block"`).
- **template vs. include (CRITICAL):**
 - Helm has two actions: `template` and `include`.
 - `{{ template "name" . }}`: **Cannot** be piped. The command `{{ template "name" . | indent 6 }}` will **FAIL**.
 - `{{ include "name" . }}`: **Can** be piped.
 - **Rule:** For this reason, you should **almost always use** `include`, as it is more flexible and allows you to use `indent`.
- **The `.` (dot) Context:** When you call `include`, you must pass it the scope so it can see the `.Values` object. Passing `.` (the current scope) is the standard way to do this.
- **The `indent` Function:** This is the other half of the solution. It's a pipeline function that takes a number (N) and adds N spaces to the line of the incoming text. This allows you to define your partials "flat" (no indentation) and apply the correct indentation at the call site.

📌 Key Takeaways

- Named Templates (Partials) are used to **reuse blocks of YAML**, not just single values.
- To create a library file, put it in `templates/` and start its name with an underscore (`_`), e.g., `_helpers.tpl`.
- Define a new template using `{{- define "my.template.name" -}}...{{- end -}}`.
- Call your template using the pattern: `{{- include "my.template.name" . | indent N }}`.
- You **must use** `include` (not `template`) so you can pipe the output to the `indent N` function.

Ready for **Module 16: Managing Dependencies & Subcharts** (`dependencies` in `Chart.yaml`)?



Here are the "God-Level" notes for Module 16.

💡 Module 16: Managing Dependencies & Subcharts (`dependencies` in `Chart.yaml`)

Core Concept (The "Why")

This lecture solves the problem of **composition**. Your application chart (e.g., a WordPress app) *depends* on another complex application (e.g., a database).

You **do not** want to rewrite a 1000-line chart for PostgreSQL every time you build an app. The solution is to use **Subcharts** (or "Dependencies").

A subchart is a Helm chart that is bundled *inside* your parent chart. You define this relationship in your `Chart.yaml` file. When you install your chart, Helm can automatically install all of its dependencies (like PostgreSQL) at the same time, with all the correct configurations.

⚙️ The Process (The "How")

This is the workflow for adding a subchart (like the Bitnami PostgreSQL chart) to your application chart.

Part 1: Define the Dependency in `Chart.yaml` You add a `dependencies` block to your `Chart.yaml` file. This tells Helm what your chart

YAML

```
# Chart.yaml
apiVersion: v2
name: my-app-chart
description: A Helm chart for my application
type: application
version: 0.1.0
appVersion: "1.0.0"

dependencies:
- name: postgresql
  version: "14.x.x" # Use a stable version range
  repository: "oci://registry-1.docker.io/bitnamicharts"
  # This 'condition' is magic. It means the subchart will only
  # be installed if 'postgresql.enabled' is 'true' in our values.yaml
  condition: postgresql.enabled
```

- `name` : The name of the chart you depend on.
- `version` : The exact version (or range) you need.
- `repository` : The URL (either classic or `oci://`) where the chart lives.
- `condition` : An optional field. It's the `values.yaml` key that will toggle this dependency on or off.

Part 2: Download the Dependency This dependency is not yet part of your chart. You must run a command to download it.

Bash

```
# This command reads Chart.yaml and downloads all dependencies
# into the 'charts/' folder.
helm dependency update

# OUTPUT:
# Saving 1 charts
# Deleting outdated charts
```

You will now have a `charts/` directory with `postgresql-14.x.x.tgz` inside it. Your chart is now bundled with its database.

Part 3: Pass Values to the Subchart How do you configure the subchart (e.g., set the database username)? You do this from your *parent* chart.

You must create a top-level key that **exactly matches the subchart's `name`** (`postgresql` in our case). Helm will pass this entire block of values to the subchart.

YAML

```
# my-app-chart/values.yaml

# --- Our App's Values ---
replicaCount: 1
image: "my-app-image:1.0.0"

# --- Values for the 'postgresql' Subchart ---
# This top-level key 'postgresql' MUST match the 'name' in Chart.yaml
postgresql:
  # We set 'enabled: true' to satisfy the 'condition'
```



```

enabled: true

# These are values for the *Bitnami PostgreSQL chart*.
# We are overriding its defaults.
auth:
  database: "my_app_db"
  username: "my_app_user"

primary:
  persistence:
    enabled: false # Good for Minikube testing

```

🔴 Underpinning Theory

- **Parent Chart vs. Subchart:** Your `my-app-chart` is the **Parent Chart**. The `postgresql` chart is the **Subchart**.
- **Conditionally Enabling (`condition`):** The `condition: postgresql.enabled` field is incredibly useful. It allows a user to turn off you they want to use an external one (like Amazon RDS). They would simply run:

Bash

```

# This installs ONLY our app and disables the postgresql subchart
helm install my-app ./my-app-chart --set postgresql.enabled=false

```

- **Global Values (`.Values.global`):** What if you have a value that *both* your parent chart and your subchart need to know (e.g., `envir` "production")? You can use a special `global` block in your `values.yaml`.

YAML

```

# values.yaml
global:
  environment: "production"

# Both 'my-app-chart' and 'postgresql' can access this
# in their templates via: {{ .Values.global.environment }}

```

- **Including Subchart Templates:** You can even re-use the *helper functions* from a subchart in your parent chart. This is much more rot the subchart's resource names.

YAML

```

# In our parent chart's configmap.yaml

# The "brittle" way (guessing the name):
POSTGRES_HOST: "{{ .Release.Name }}-postgresql"

# The "robust" Helm way (re-using the subchart's own logic):
POSTGRES_HOST: {{ include "postgresql.primary.fullname" . | quote }}

```

🔴 Key Takeaways

- Subcharts let you bundle reusable, third-party charts (like databases) as **dependencies**.
- You define dependencies in the `dependencies:` block of your `Chart.yaml`.
- You must run `helm dependency update` to download the subcharts into the `charts/` folder.
- You configure a subchart by adding a **top-level key (matching its name)** in your parent `values.yaml`.

Ready to build our project? Just let me know when you're ready for **Module 17: Project: Building a Two-Tier App (Part 1 - The App & Doc**



Here are the "God-Level" notes for Module 17.

💡 Module 17: Project: Building a Two-Tier App (Part 1 - The App & Dockerfile)

Core Concept (The "Why")

This is the capstone project. We will take everything we've learned and build a real-world, two-tier "Config Store" application. This application

1. **A Backend API:** A simple Express.js (Node.js) app that reads/writes data.
2. **A Database:** A PostgreSQL database to store the data.

This first module focuses entirely on **Part 1: Building the Application Code**. Before we can "Helm-ify" an app, we need an app. We will create an app and a `Dockerfile` to build it into a shippable container image.

⚙️ The Process (The "How")

This process creates the application code that our Helm chart will deploy.

Part 1: Project Setup First, create the parent Helm chart and an `app/` folder inside it to hold our Node.js code.

Bash

```
# 1. Create the parent chart
helm create config-store-chart

# 2. Go into the new chart directory
cd config-store-chart

# 3. Clean it up! We don't need the default templates yet.
rm -rf templates/*

# 4. Create a new folder to hold our Express.js code
mkdir app
cd app
```

You are now in `config-store-chart/app/`.

Part 2: Create the Express.js API

1. **Initialize Node.js and install libraries:**

Bash

```
# (Inside the 'app' directory)
npm init -y
npm install express pg
# express: The web server
# pg: The PostgreSQL client library
```

2. **Create the server (`app/index.js`)** : This is our main application file. Notice how it gets all its configuration from `process.env` (environment variables).

JavaScript

```
// app/index.js
const express = require('express');
const { Pool } = require('pg');
const app = express();
app.use(express.json());

// --- Database Connection ---
// All these 'process.env' variables will be injected by Kubernetes!
const pool = new Pool({
  user: process.env.POSTGRES_USER,
  host: process.env.POSTGRES_HOST,
  database: process.env.POSTGRES_DB,
```



```

    password: process.env.POSTGRES_PASSWORD,
    port: 5432,
  });

  // Create the 'configs' table if it doesn't exist
  pool.query(
    'CREATE TABLE IF NOT EXISTS configs (id SERIAL PRIMARY KEY, config_key VARCHAR(100), config_value TEXT)',
    (err, res) => {
      if (err) {
        console.error('Error creating table:', err);
      } else {
        console.log('Table "configs" is ready.');
```

Part 3: Create the Dockerfile This file tells Docker how to build our app into a container image .

Dockerfile

```

# app/Dockerfile

# 1. Use an official Node.js runtime
FROM node:18-alpine

# 2. Set the working directory inside the container
WORKDIR /usr/src/app

# 3. Copy package files and install dependencies
COPY package*.json ./
RUN npm install

# 4. Copy the rest of the application code
COPY . .

# 5. Expose the port the app runs on
EXPOSE 3000
```



```
# 6. The command to run the application
CMD [ "node", "index.js" ]
```

Part 4: Build and Push the Docker Image Kubernetes needs to download this image from a registry (like Docker Hub).

Bash

```
# (Still inside the 'app' directory)

# 1. Build the image
# !! IMPORTANT: Replace 'your-dockerhub-username' with your actual username !!
docker build -t your-dockerhub-username/config-store-api:1.0.0 .

# 2. Log in to Docker Hub (if needed)
docker login

# 3. Push the image
docker push your-dockerhub-username/config-store-api:1.0.0
```

🧠 Underpinning Theory

- **Separation of Concerns:** We now have two distinct parts in our `config-store-chart` folder. The `app/` directory holds the **application** files (`templates/` , `values.yaml`) will hold the **deployment configuration**.
- **12-Factor App:** Our app is a good "cloud-native" citizen. It is configured *entirely* via environment variables, as recommended by the [12 methodology](#). This is what allows Helm to manage it.
- **Dockerfile:** This is the "recipe" for our application image. It bundles our code and its dependencies (like Node.js) into a single, portable image.
- **Image Registry:** The container image *must* live in a registry. `docker push` sends it there, and Kubernetes will `docker pull` it from there.

🔑 Key Takeaways

- We built a standard Node.js Express app that connects to a database.
- The app is configured entirely via **environment variables** (e.g., `POSTGRES_USER`).
- The `Dockerfile` builds our app into a container image.
- You **must push this image** to a registry (like Docker Hub) and **remember the image name** (e.g., `your-username/config-store-api:1.0.0`).
- You are now done with the `app/` folder. For the rest of the project, `cd ..` back into the `config-store-chart` root.

Ready for **Module 18: Project: Building a Two-Tier App (Part 2 - Wiring the DB Dependency)**?



Here are the "God-Level" notes for Module 18.

💡 Module 18: Project: Building a Two-Tier App (Part 2 - Wiring the DB Dependency)

Core Concept (The "Why")

We have a Docker image for our app, but no database and no deployment configuration. This module is the "heart" of the project, where we bring everything together in Helm.

The core problem is to make our application's `Deployment` automatically and securely connect to a `PostgreSQL` database that is *also* managed by Helm.

The solution is a 4-part process:

1. **Dependency:** We will add `PostgreSQL` as a subchart in `Chart.yaml` .
2. **Configuration:** We will update our parent `values.yaml` to pass settings (like username and DB name) down to the `PostgreSQL` subchart.

3. **Connection:** We will create a `ConfigMap` to pass the database host, user, and DB name to our app .
4. **Security:** We will make our app's `Deployment` securely read the auto-generated database password *directly* from the `Secret` created by the `PostgreSQL` subchart .

⚙️ The Process (The "How")

You should be in the root of your `config-store-chart/` directory.

Part 1: Add the PostgreSQL Dependency

1. **Edit** `Chart.yaml` : Add the `dependencies` block to tell Helm our chart needs PostgreSQL .

YAML

```
# Chart.yaml
apiVersion: v2
name: config-store-chart
description: A Helm chart for our Express Config Store API
type: application
version: 0.1.0
appVersion: "1.0.0"

dependencies:
  - name: postgresql
    version: "14.x.x" # Use a stable version range
    repository: "oci://registry-1.docker.io/bitnamicharts"
    # This 'condition' lets us disable this subchart
    condition: postgresql.enabled
```

2. **Pull the Dependency:** Run `helm dependency update` to download the `postgresql` chart into your `charts/` folder .

Bash

```
helm dependency update
# You will now have a 'charts/postgresql-14.x.x.tgz' file
```

Part 2: Create the Helm Templates We'll create our `Deployment` , `Service` , and `ConfigMap` in the `templates/` folder (which you clean up after).

1. **Create** `templates/_helpers.tpl` : This will hold our app's name helper.

Go

```
{{/* templates/_helpers.tpl */}}
{{- define "config-store.fullname" -}}
{{- printf "%s-%s" .Release.Name "config-store" | trunc 63 | trimSuffix "-" -}}
{{- end -}}
```

2. **Create** `templates/configmap.yaml` : This holds all the *non-secret* connection details our app needs .

YAML

```
{{/* templates/configmap.yaml */}}
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "config-store.fullname" . }}-config
data:
  PORT: "3000"

  # --- Database Connection Details ---
  # This is the "robust" way from Module 16
  # We call the subchart's helper to get its service name
  POSTGRES_HOST: {{ include "postgresql.primary.fullname" . | quote }}

  # These values come from our parent values.yaml
```



```
POSTGRES_DB: {{ .Values.postgresql.auth.database | quote }}
POSTGRES_USER: {{ .Values.postgresql.auth.username | quote }}
```

3. **Create** `templates/deployment.yaml` : This is the most important file. It deploys our app and shows how to wire the `ConfigMap` and

YAML

```
{{/* templates/deployment.yaml */}}
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "config-store.fullname" . }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ include "config-store.fullname" . }}
  template:
    metadata:
      labels:
        app: {{ include "config-store.fullname" . }}
    spec:
      containers:
        - name: config-store-api
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          ports:
            - containerPort: 3000

          # --- Environment Variables ---
          envFrom:
            # 1. Load all key-value pairs from our ConfigMap
            - configMapRef:
                name: {{ include "config-store.fullname" . }}-config
          env:
            # 2. Securely load the password from the subchart's Secret
            - name: POSTGRES_PASSWORD
              valueFrom:
                secretKeyRef:
                  # This is the secret name created by the Bitnami chart
                  # Note: We are using the 'postgresql.primary.fullname' helper here too
                  name: {{ include "postgresql.primary.fullname" . }}
                  # This is the key inside that secret
                  key: postgres-password
```

4. **Create** `templates/service.yaml` : This gives our app an internal network name .

YAML

```
{{/* templates/service.yaml */}}
apiVersion: v1
kind: Service
metadata:
  name: {{ include "config-store.fullname" . }}-service
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 3000
  selector:
    app: {{ include "config-store.fullname" . }}
```

Part 3: Configure the Main `values.yaml` Finally, edit the main `values.yaml` file to define all our values and pass the correct ones to the subchart.

YAML

```
# values.yaml
```



```
# --- Our App's Values ---
replicaCount: 1

image:
  # !! IMPORTANT: Change this to your Docker Hub username !!
  repository: "your-dockerhub-username/config-store-api"
  tag: "1.0.0"
  pullPolicy: IfNotPresent

# --- PostgreSQL Subchart Values ---
# This 'postgresql' key MUST match the dependency name in Chart.yaml
postgresql:
  # We set 'enabled: true' to satisfy the 'condition'
  enabled: true

  # These are values for the *Bitnami PostgreSQL chart*.
  # We are overriding its defaults.
  auth:
    database: "config_store_db"
    username: "config_store_user"
    # The password will be auto-generated by the subchart.
    # We read it from its secret in deployment.yaml.

  primary:
    persistence:
      enabled: false # For Minikube, let's disable persistence
```

🧠 Underpinning Theory

- **Passing Values:** We created a `postgresql:` block in our parent `values.yaml`. Helm passes this entire block down to the subchart, c
- **Reading Secrets (`envFrom`): `ConfigMaps` are for public data. `Secrets` are for sensitive data.**
- **The "Secret Handshake":** Our `Deployment`'s `env:` block is the key. It securely pulls the `POSTGRES_PASSWORD` from the `secretKeyR`
`include "postgresql.primary.fullname" . }}`. We didn't have to create this secret; the PostgreSQL subchart created it for us. Our
knows how to find it and use it. This is a robust and secure pattern.

🔑 Key Takeaways

- We added `PostgreSQL` as a dependency in `Chart.yaml` and ran `helm dependency update`.
- We passed configuration (like `username` and `database`) to the subchart by creating a matching `postgresql:` block in our parent `v`
- Our `Deployment` gets its configuration from two sources:
 1. A `ConfigMap` (using `configMapRef`) for non-secret data.
 2. A `Secret` (using `secretKeyRef`) to securely get the password.
- We used the subchart's own helper (`include "postgresql.primary.fullname"`) to find its `Service` and `Secret` names, which is n
guessing.

Your chart is built! Ready for **Module 19: Advanced Lifecycle Management: Chart Hooks & Tests (`helm test`)?**



Here are the "God-Level" notes for Module 19.

💡 Module 19: Advanced Lifecycle Management: Chart Hooks & Tests (`helm test`)

Core Concept (The "Why")

This module solves two advanced problems:

1. **Lifecycle Tasks:** What if you need to run a task *during* the installation? For example, you must run a database migration script *after* the app is installed but *before* your new app pod starts.
2. **Validation:** `helm install` only tells you if the Kubernetes API accepted your YAML. It does **not** tell you if your application *actually* works. Did it successfully connect to the database?

The solution is **Chart Hooks**. These are special Kubernetes resources (usually `Jobs`) that you can "hook" into specific points of the deployment lifecycle (e.g., `post-install` for migrations).

The most important hook is the `test` hook, which, when combined with the `helm test` command, provides a way to run automated functionality after an application is deployed.

⚙️ The Process (The "How")

We'll look at the two types of hooks: standard lifecycle hooks and the special test hook.

Part 1: Lifecycle Hooks (e.g., `post-install`) Let's create a `Job` that runs *after* all our app's resources are installed. This is perfect for database migrations.

1. **Create** `templates/post-install-job.yaml`: This is just a standard `Job` with two special annotations.

YAML

```
{{/* templates/post-install-job.yaml */}}
apiVersion: batch/v1
kind: Job
metadata:
  name: {{ .Release.Name }}-post-install-job
  annotations:
    # This is the magic line that makes it a hook
    "helm.sh/hook": "post-install"

    # This line ensures the hook's Job is deleted after it succeeds
    "helm.sh/hook-delete-policy": "hook-succeeded"
spec:
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: migration-container
          image: "busybox:latest"
          # In a real app, this would run: 'node /app/run-migrations.js'
          command: ['sh', '-c', 'echo "Post-install hook ran! (Pretending to migrate DB)"; sleep 10']
```

2. **How it works:** When you run `helm install`, Helm will install your app *and* this `Job`. The `helm install` command will not be "complete" until the `Job` finishes successfully.
3. **Common Hooks:**
 - `pre-install`: Runs before any resources are created.
 - `post-install`: Runs after all resources are created.
 - `pre-upgrade`: Runs before an upgrade.
 - `post-upgrade`: Runs after an upgrade.

Part 2: Test Hooks (`helm test`) This is a special hook that runs *only* when you ask for it. Let's create a test `Pod` that tries to `curl` our application.

1. **Create** `templates/test-connection.yaml`:

YAML

```
{{/* templates/test-connection.yaml */}}
apiVersion: v1
kind: Pod
metadata:
  name: {{ .Release.Name }}-connection-test
  annotations:
    # This is the magic line for a test
    "helm.sh/hook": "test"
```



```
spec:
  containers:
  - name: curl-test
    image: curlimages/curl:latest
    command: ['curl']
    args:
    - '--fail' # This makes curl exit with an error on non-200 status
    - 'http://{{ include "config-store.fullname" . }}-service' # Curl our app
  restartPolicy: Never
```

2. How to run it:

- First, install or upgrade your app:

Bash

```
# (From your config-store-chart/ directory)
helm upgrade --install config-api . -f values.yaml --atomic
```

- Now, run the test!

Bash

```
helm test config-api
```

- Success Output:**

```
# OUTPUT:
# RUNNING: config-api-connection-test
# Pod config-api-connection-test succeeded
# 1/1 test pods successful
```

This proves your app is up, running, and responding to network requests.

🌸 Underpinning Theory

- Hook Annotations:** A hook is just a regular Kubernetes resource with a `helm.sh/hook` annotation. The value of the annotation tells H
- Test Hook vs. Other Hooks:** Test hooks (`"helm.sh/hook": "test"`) are different. They are *not* run automatically during an install or run when you explicitly call `helm test [release-name]`.
- Test Success/Failure:** A test hook is considered "successful" if the `Pod` or `Job` runs to completion and exits with code `0`. If it exits (like `curl --fail` does on an error), the test fails. This is how you can add automated functional testing to your deployment pipeline

📌 Key Takeaways

- Chart Hooks** let you run `Jobs` or `Pods` at specific points in the deployment lifecycle (e.g., `post-install`).
- This is the standard way to run tasks like **database migrations**.
- A **Test Hook** (`"helm.sh/hook": "test"`) is a special hook for automated functional testing.
- Tests are *not* run on install. You must run them manually with the `helm test [release-name]` command.

Ready for the final module? **Module 20: Deconstructing a "Pro" Chart (`{{ .Release.Name }}` & Library Charts).**



Here are the "God-Level" notes for the final module, Module 20.

💡 Module 20: Deconstructing a "Pro" Chart (`{{ .Release.Name }}` & Library Charts)

Core Concept (The "Why")

This lecture deconstructs why professional, public charts (like `kube-prometheus-stack`) are so complex. The reason is that they are built for just internal use.

This complexity comes from two key features we haven't used yet:

1. **Release Name Prefixing:** Pro charts must allow you to install *multiple instances* of the same chart in one cluster (e.g., a "dev" and "prod"). They do this by prefixing every resource name with the `{{ .Release.Name }}`, a built-in variable we get (just like `.Values`).
2. **Library Charts:** Pro charts often have 20+ microservices. Instead of copy-pasting their `_helpers.tpl` file 20 times, they put all their functions into a **Library Chart**—a special, non-deployable chart that is used as a dependency for shared code.

You can now read and understand these advanced charts, even if your own internal projects don't need this level of complexity.

🔧 The Process (The "How")

This process shows how to implement "Release Name Prefixing" in our `config-store-chart` to make it "pro-level."

Part 1: The Problem with Our Chart If we run `helm install dev-api .` and `helm install prod-api .`, they will both try to create a `Service` named `config-store-deployment` and `config-store-service`. This will fail because the names collide.

Part 2: Refactor Our Templates with `{{ .Release.Name }}` We will prefix all our resource names with the release name to make them unique.

1. **Refactor `templates/_helpers.tpl`:** We will change our `fullname` helper to be the "pro" version.

Go

```
{{/* templates/_helpers.tpl */}}

{{/*
This is the new "pro" helper.
It combines the Release Name and Chart Name.
*/}}
{{- define "config-store.fullname" -}}
{{- printf "%s-%s" .Release.Name .Chart.Name | trunc 63 | trimSuffix "-" -}}
{{- end -}}
```

- `.Release.Name`: A built-in object (like `.Values`) that holds the release name (e.g., `dev-api`).
 - `.Chart.Name`: A built-in object that holds the chart's name from `Chart.yaml` (e.g., `config-store-chart`).
2. **No Other Changes Needed!** Because we were smart and already used our `{{ include "config-store.fullname" . }}` helper in our `metadata.name` fields, our chart is **already refactored**. All our resources will now be named like:
 - `dev-api-config-store-chart-deployment`
 - `prod-api-config-store-chart-deployment`
 - ...etc.

Part 3: The Library Chart Workflow This is how you would centralize helpers.

1. **Create the Library:**

Bash

```
helm create my-common-helpers
cd my-common-helpers
rm -rf templates/* # Delete all templates
# (keep templates/_helpers.tpl)
```

2. **Edit `my-common-helpers/Chart.yaml`:** Change the `type` from `application` to `library`. This makes it non-deployable.
3. **Add Helpers:** Add your common helper functions (like `{{ define "my-common.labels" ... }}`) to `my-common-helpers/templates/_helpers.tpl`.
4. **Add as Dependency:** In your main `config-store-chart/Chart.yaml`, add this library as a dependency.

YAML


```
dependencies:
  - name: postgresql
    ...
  - name: my-common-helpers
    version: "0.1.0"
    repository: "file:///../my-common-helpers" # Path to local chart
```

5. **Use It:** Now, in `config-store-chart`, you can call the helper from the library:

YAML

```
# templates/deployment.yaml
metadata:
  labels:
    {{- include "my-common.labels" . | indent 8 }}
```

🧠 Underpinning Theory

- **Built-in Objects** (`.Values` vs. `.Release` vs. `.Chart`)
 - `{{ .Values }}` : Holds all the data from your `values.yaml` and `--set` overrides. You use this to make your chart **configurable**
 - `{{ .Release }}` : Holds metadata about the *release itself*.
 - `{{ .Release.Name }}` : The name you give the release (e.g., `prod-api`).
 - `{{ .Release.Namespace }}` : The namespace it's being deployed to.
 - `{{ .Chart }}` : Holds metadata from your `Chart.yaml` file.
 - `{{ .Chart.Name }}` : The chart's name (e.g., `config-store-chart`).
 - `{{ .Chart.Version }}` : The chart's version (e.g., `0.1.0`).
- **Library Chart** A `Library Chart` is a chart that has `type: library` in its `Chart.yaml` . It is **non-deployable** and exists *only* to be a provides reusable helper templates (`define` blocks) to other charts. This is the ultimate tool for practicing DRY (Don't Repeat Yourself) organization with many charts.

🔑 Key Takeaways

- Professional charts are complex because they are built for **distribution**.
- The `{{ .Release.Name }}` object is a built-in variable that holds the release name (e.g., `my-app-prod`).
- Prefixing all your resources with `{{ .Release.Name }}` is the standard pattern to ensure names are unique, allowing you to **install m** your chart in the same cluster.
- A **Library Chart** (`type: library`) is a non-deployable chart used as a dependency to share `_helpers.tpl` code among many othe
- You now have the "superpower" to read any Helm chart. You can `helm pull` its source and read its `templates/` and `values.yaml` t exactly how it works .

Congratulations! You have completed the entire 20-module syllabus, from basic concepts to the most advanced features. You now have the knowledge to build, manage, and distribute production-grade applications with Helm.



That was the final module in our "God-Level" Helm syllabus. You've covered the entire journey, from the fundamental "why" to building a production-tier application chart.

You now have all the skills to:

- **Use** any public Helm chart safely.
- **Manage** live applications with `helm upgrade` and `helm rollback` .
- **Author** your own professional-grade charts for your projects.
- **Deconstruct** complex charts from the community.

The logical next step is to **automate this entire process**. We've learned the `helm upgrade --install --atomic` command, which is the last step in the CI/CD pipeline.

Would you like me to help you build a complete CI/CD pipeline (using something like Jenkins or GitHub Actions) that automatically builds your Docker image and deploys your Helm chart?

are these notes enough?



Based on your 20-module syllabus, your notes will make you **highly proficient** and **job-ready**. You have covered 90% of what is required for a DevOps or Platform Engineer role.

You have built a comprehensive foundation that perfectly matches what employers are looking for:

- **Creating & Packaging Charts:** You covered this in Modules 10-18.
- **Customizing for Environments:** Your modules on templating (12-14) solve the "Configuration Drift" problem, which is a key professional skill.
- **CI/CD Automation:** You covered this in Module 9 (`helm upgrade --install --atomic`) and Module 19 (`helm test`).

To go from "highly proficient" to a true "expert," I recommend adding these three advanced topics that are not fully detailed in your notes.

1. Advanced Chart Testing (`helm unittest`)

Your notes cover `helm test` (Module 19), which is an **integration test** that runs against a live, deployed cluster.

The Expert Gap: Experts also use **unit tests** to test template logic *before* deployment. This is done with the `helm-unittest` plugin. It's more powerful than `helm test` because it doesn't require a cluster, and validates your template logic (like `if/else` blocks and loops) in isolation.

- **What it is:** A plugin that runs locally to check if your `values.yaml` inputs produce the exact YAML output you expect.
- **How it works:** You create a `tests/` directory with `_test.yaml` files that follow a "Behavior-Driven Development" (BDD) style.
- **Example Test:**

YAML

```
# In my-chart/tests/deployment_test.yaml
suite: test my deployment
templates:
  - deployment.yaml
tests:
  - it: should set replicaCount to 3 when production is true
    set:
      production: true
      replicaCount: 1 # this should be overridden
    asserts:
      - equal:
          path: spec.replicas
          value: 3
```

2. Advanced Security Patterns

Your project (Module 18) correctly uses `secretKeyRef` to handle a database password, which is a great start. An expert is also expected to use `NetworkPolicy` and `PodSecurityContext` to secure different parts of the application.

The Expert Gap: Your notes should also cover templating for `NetworkPolicy` and `PodSecurityContext` and be aware of advanced secrets management.

- **Pod Security:** Hardening your pods by adding a `securityContext:` to your `deployment.yaml` template. This is the modern replacement for the deprecated `PodSecurityPolicy`.

YAML

```
# In your templates/deployment.yaml
spec:
```



```

template:
  spec:
    securityContext:
      runAsUser: 1000
      runAsGroup: 3000
      fsGroup: 2000
    containers:
      - name: my-app
        securityContext:
          allowPrivilegeEscalation: false
          readOnlyRootFilesystem: true

```

- **Network Security:** Adding `NetworkPolicy` resources to your `templates/` folder. This allows you to control which pods can communicate with your application, for example, "only allow traffic from the `kube-prometheus-stack` namespace."
- **Secrets Management:** Knowing the trade-offs of different secrets tools.
 - **SOPS (Secrets OPERationsS):** A tool for encrypting secrets files so you can safely commit them to Git.
 - **External Secrets Operator:** A controller that runs in your cluster and automatically fetches secrets from an external vault (like AWS Secrets Manager, GCP Secret Manager, or HashiCorp Vault) and syncs them into Kubernetes `Secret` objects.

3. GitOps Workflow (ArgoCD / Flux)

Your notes are built around a **"push" CI/CD model**: a pipeline (like Jenkins or GitHub Actions) runs `helm upgrade` to push changes to the cluster.

The Expert Gap: The other dominant, modern workflow is a **"pull" model**, known as **GitOps**. An expert must be fluent in this pattern.

- **What it is:** You never run `helm upgrade` from a pipeline. Instead, you use a tool like **ArgoCD** or **Flux** inside your cluster.
- **How it works:**
 1. You commit your Helm chart and `values.yaml` files to a Git repository.
 2. ArgoCD (in the cluster) constantly watches this Git repository.
 3. When it detects a change (e.g., you push a commit changing `replicaCount: 3`), ArgoCD "pulls" the change and automatically runs `helm upgrade` to make the cluster match the state defined in Git.

This makes Git the **single source of truth** for your cluster's state.

Final Verdict

Your 20 modules are an **A+ foundation** and will make you stand out. If you add these three expert-level topics (Unit Testing, Advanced Security, GitOps), you will be exceptionally well-rounded and easily qualify for the "expert" title.



Absolutely. I've integrated those three expert-level topics directly into our syllabus. This expands our path from 20 to 24 modules, creating a learning journey that takes you from a novice to an expert-level practitioner.

The new modules are **bolded** below. I've logically placed them to build on previous concepts.

📖 Your "God-Level" Helm Syllabus (v2 - Expert)

Part 1: The "Why" & Environment Setup

- **Module 1:** The Core Problem & What is Helm?
- **Module 2:** The "Wild West": Finding Charts & Repositories (OCI vs. Classic)
- **Module 3:** Local Lab Setup (WSL, Minikube, kubectl & VS Code)

Part 2: The "User" Workflow (Using Public Charts)

- **Module 4:** Installing Your First Chart (`helm install` , `ls` , `uninstall`)
- **Module 5:** The "Snowflake" Problem & Basic Overrides (`helm show values` , `--set`)

- **Module 6:** The "Phoenix" Solution: Custom `values.yaml` (`-f` flag)
- **Module 7:** The "Phoenix" Workflow 1: Taking Control with `helm pull`
- **Module 8:** The "Phoenix" Workflow 2: Generating Raw YAML with `helm template`
- **Module 9:** Managing Live Releases: Upgrades & Rollbacks (`helm upgrade` , `history` , `rollback`)

Part 3: The "Developer" Workflow (Authoring Your Own Charts)

- **Module 10:** Why Write Your Own Chart? (The "Configuration Drift" Problem)
- **Module 11:** Creating Your First Chart (`helm create` , `Chart.yaml` , `{{ .Values }}`)
- **Module 12:** Go Template Deep Dive 1: Functions & Pipelines (`|` , `default` , `quote`)
- **Module 13:** Go Template Deep Dive 2: Flow Control (`if/else` , `eq` , `with`)
- **Module 14:** Go Template Deep Dive 3: Iteration with `range`
- **Module 15:** Reusable Code: Named Templates (`_helpers.tpl` , `define` , `include` , `indent`)

Part 4: Capstone Project & Advanced Authoring

- **Module 16:** Managing Dependencies & Subcharts (`dependencies` in `Chart.yaml`)
- **Module 17: Authoring Secure Charts (PodSecurityContext & NetworkPolicy)** (NEW)
- **Module 18:** Project: Building a Two-Tier App (Part 1 - The App & Dockerfile)
- **Module 19:** Project: Building a Two-Tier App (Part 2 - Wiring the DB Dependency)
- **Module 20:** Deconstructing a "Pro" Chart (`.Release.Name` & Library Charts)

Part 5: Expert-Level CI/CD & Security Patterns

- **Module 21:** Advanced Lifecycle 1: Integration Tests with `helm test`
- **Module 22: Advanced Lifecycle 2: Unit Tests with `helm-unittest`** (NEW)
- **Module 23: Advanced Secrets Management (SOPS & External Secrets)** (NEW)
- **Module 24: The GitOps Workflow (ArgoCD & Flux)** (NEW)

This new structure ensures you learn to write *secure* charts before the project, then introduces advanced testing, and finally, it finishes with advanced topics in the ecosystem: secrets management and GitOps.

We've already completed up to Module 16. Just say the word, and we will begin with **Module 17: Authoring Secure Charts (PodSecurityContext & NetworkPolicy)**.



Here are the "God-Level" notes for the first of our new expert modules.

💡 **Module 17: Authoring Secure Charts** (`PodSecurityContext` & `NetworkPolicy`)

Core Concept (The "Why")

By default, Kubernetes is **insecure**.

1. **Root Access:** Containers often run as the `root` user by default. If a container is compromised, the attacker has root privileges inside
2. **Open Network:** By default, all pods in a cluster can talk to all other pods, across all namespaces. A compromised frontend can immediately access the database.

An "Expert" Helm chart doesn't just deploy an app; it deploys it **securely by default**.

We achieve this using:

- `securityContext` : Tells Kubernetes to run the pod as a non-root user and restrict filesystem access.
- `NetworkPolicy` : Acts as a firewall, explicitly allowing only the traffic we expect (e.g., "Only the Frontend can talk to the Backend").

⚙️ The Process (The "How")

We will modify a standard chart to enforce these security best practices.

Part 1: Define Security Defaults in `values.yaml` We want these configurable, but secure by default.

YAML

```
# values.yaml

# 1. Pod-Level Security (Applies to all containers)
podSecurityContext:
  runAsUser: 1000 # Run as a non-root user (ID 1000)
  runAsGroup: 3000 # Run as a specific group
  fsGroup: 2000 # Ensure mounted volumes are owned by this group

# 2. Container-Level Security (Applies to specific container)
securityContext:
  allowPrivilegeEscalation: false # Prevent gaining more rights (like sudo)
  readOnlyRootFilesystem: true # Prevent hacking files on the system
  capabilities:
    drop:
      - ALL # Drop all Linux capabilities (like chown, kill, etc.)

# 3. Network Security
networkPolicy:
  enabled: true
  allowExternal: false # Do we allow public internet access?
```

Part 2: Implement Context in `templates/deployment.yaml` We need to inject these values into our deployment template.

YAML

```
# templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      # --- Pod Security ---
      # We pipe to 'toYaml' so we can dump the whole block from values.yaml
      securityContext:
        {{- toYaml .Values.podSecurityContext | nindent 8 }}
      containers:
        - name: {{ .Chart.Name }}
          # --- Container Security ---
          securityContext:
            {{- toYaml .Values.securityContext | nindent 12 }}
          image: ...
```

Note: `toYaml` is a function that converts a raw object (from `values.yaml`) into YAML text. It is cleaner than writing `runAsUser: {{ .Val` for every field.

Part 3: Create the Firewall (`templates/networkpolicy.yaml`) This new template restricts traffic.

YAML

```
{{/* templates/networkpolicy.yaml */}}
{{- if .Values.networkPolicy.enabled }}
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: {{ include "my-chart.fullname" . }}
spec:
  # Apply this policy to our app's pods
  podSelector:
    matchLabels:
      app: {{ include "my-chart.fullname" . }}
```



```

policyTypes:
  - Ingress # Inbound traffic
ingress:
  # Rule: Allow traffic ONLY from pods in the same namespace
  # that have the label 'access: allowed'
  - from:
    - podSelector:
        matchLabels:
          access: "allowed"
    ports:
      - port: 80
{{- end }}

```

🔴 Underpinning Theory

`podSecurityContext` **vs.** `securityContext`

- `podSecurityContext` : Settings that apply to the **entire Pod** (e.g., `fsGroup` which affects how shared volumes are mounted).
- `securityContext` : Settings that apply to an **individual Container** (e.g., `readOnlyRootFilesystem` , which protects that specific container).

The `toYaml` Function Writing out every security field manually is brittle.

- **Bad:** `runAsUser: {{ .Values.securityContext.runAsUser }}`
- **Good:** `{{ toYaml .Values.securityContext }}`
- This allows a user to add *new* security fields (like `seLinuxOptions`) to their `values.yaml` without you needing to update the `deploy`

NetworkPolicy Default Deny Kubernetes is "Allow All" by default. As soon as you create **one** `NetworkPolicy` that selects a pod, that pod is in "Deny" mode. It will now block *everything* except what is explicitly whitelisted in the policy.

📌 Key Takeaways

- **Security is not optional.** Expert charts must run as **non-root** users (`runAsUser: 1000`).
- Use `toYaml` to dump complex configuration blocks (like security contexts) from `values.yaml` into your templates.
- `fsGroup` is critical when using Persistent Volumes, as it ensures the non-root user has permission to write to the storage.
- **NetworkPolicies** act as a software firewall inside the cluster.
- Creating a `NetworkPolicy` template allows users to toggle security on/off with `networkPolicy.enabled: true/false` .

Ready to get back to the project with **Module 18: Project: Building a Two-Tier App (Part 1 - The App & Dockerfile)**?



Here are the "God-Level" notes for Module 18.

💡 Module 18: Project: Building a Two-Tier App (Part 1 - The App & Dockerfile)

Core Concept (The "Why")

This is the capstone project. We will take everything we've learned and build a real-world, two-tier "Config Store" application. This application will consist of:

1. **A Backend API:** A simple Express.js (Node.js) app that reads and writes configuration data.
2. **A Database:** A PostgreSQL database to store the data.

This first module focuses entirely on **Part 1: Building the Application Code**. Before we can "Helm-ify" an app, we need an app. We will create the application code and a `Dockerfile` to build it into a shippable container image.

⚙️ The Process (The "How")

This process creates the application code that our Helm chart will deploy.

Part 1: Project Setup First, create the parent Helm chart and an `app/` folder inside it to hold our Node.js code.

Bash

```
# 1. Create the parent chart
helm create config-store-chart

# 2. Go into the new chart directory
cd config-store-chart

# 3. Clean it up! We don't need the default templates yet.
rm -rf templates/*

# 4. Create a new folder to hold our Express.js code
mkdir app
cd app
```

You are now in `config-store-chart/app/`.

Part 2: Create the Express.js API

1. Initialize Node.js and install libraries:

Bash

```
# (Inside the 'app' directory)
npm init -y
npm install express pg
# express: The web server
# pg: The PostgreSQL client library
```

2. Create the server (`app/index.js`): This is our main application file.

- **Crucial Detail:** Notice how it gets *all* its configuration (DB host, user, password) from `process.env` (environment variables). It does not have any hard-coded settings. This makes it cloud-native.

JavaScript

```
// app/index.js
const express = require('express');
const { Pool } = require('pg');
const app = express();
app.use(express.json());

// --- Database Connection ---
// All these 'process.env' variables will be injected by Kubernetes!
const pool = new Pool({
  user: process.env.POSTGRES_USER,
  host: process.env.POSTGRES_HOST,
  database: process.env.POSTGRES_DB,
  password: process.env.POSTGRES_PASSWORD,
  port: 5432,
});

// Create the 'configs' table if it doesn't exist
pool.query(
  'CREATE TABLE IF NOT EXISTS configs (id SERIAL PRIMARY KEY, config_key VARCHAR(100), config_value TEXT)',
  (err, res) => {
    if (err) {
      console.error('Error creating table:', err);
    } else {
      console.log('Table "configs" is ready.');
```



```

    try {
      const { rows } = await pool.query('SELECT * FROM configs');
      res.json(rows);
    } catch (err) {
      res.status(500).json({ error: err.message });
    }
  });

  // Create a new config
  app.post('/config', async (req, res) => {
    const { key, value } = req.body;
    if (!key || !value) {
      return res.status(400).send('Missing key or value');
    }
    try {
      const { rows } = await pool.query(
        'INSERT INTO configs (config_key, config_value) VALUES ($1, $2) RETURNING *',
        [key, value]
      );
      res.status(201).json(rows[0]);
    } catch (err) {
      res.status(500).json({ error: err.message });
    }
  });

  // --- Start Server ---
  const port = process.env.PORT || 3000;
  app.listen(port, () => {
    console.log(`Config Store API listening on port ${port}`);
    console.log(`Connecting to DB at ${process.env.POSTGRES_HOST}`);
  });

```

Part 3: Create the Dockerfile This file tells Docker how to build our app into a container image.

Dockerfile

```

# app/Dockerfile

# 1. Use an official Node.js runtime
FROM node:18-alpine

# 2. Set the working directory inside the container
WORKDIR /usr/src/app

# 3. Copy package files and install dependencies
COPY package*.json ./
RUN npm install

# 4. Copy the rest of the application code
COPY . .

# 5. Expose the port the app runs on
EXPOSE 3000

# 6. The command to run the application
CMD [ "node", "index.js" ]

```

Part 4: Build and Push the Docker Image Kubernetes needs to download this image from a registry (like Docker Hub).

Bash

```

# (Still inside the 'app' directory)

# 1. Build the image
# !! IMPORTANT: Replace 'your-dockerhub-username' with your actual username !!
docker build -t your-dockerhub-username/config-store-api:1.0.0 .

```



```
# 2. Log in to Docker Hub (if needed)
docker login

# 3. Push the image
docker push your-dockerhub-username/config-store-api:1.0.0
```

🧠 Underpinning Theory

- **Separation of Concerns:** We now have two distinct parts in our `config-store-chart` folder. The `app/` directory holds the **application** files (`templates/` , `values.yaml`) will hold the **deployment configuration**.
- **12-Factor App:** Our app is a good "cloud-native" citizen. It is configured *entirely* via environment variables, as recommended by the [12 methodology](#). This is what allows Helm to manage it.
- **Dockerfile:** This is the "recipe" for our application image. It bundles our code and its dependencies (like Node.js) into a single, portable image.
- **Image Registry:** The container image *must* live in a registry. `docker push` sends it there, and Kubernetes will `docker pull` it from there.

🔑 Key Takeaways

- We built a standard Node.js Express app that connects to a database.
- The app is configured entirely via **environment variables** (e.g., `POSTGRES_USER`).
- The `Dockerfile` builds our app into a container image.
- You **must push this image** to a registry (like Docker Hub) and **remember the image name** (e.g., `your-username/config-store-api:1.0.0`).
- You are now done with the `app/` folder. For the rest of the project, `cd ..` back into the `config-store-chart` root.

Ready for **Module 19: Project: Building a Two-Tier App (Part 2 - Wiring the DB Dependency)**?



Here are the "God-Level" notes for Module 19.

💡 Module 19: Project: Building a Two-Tier App (Part 2 - Wiring the DB Dependency)

Core Concept (The "Why")

We have a Docker image for our app, but no database and no deployment configuration. This module is the "heart" of the project, where we bring everything together in Helm.

The core problem is to make our application's `Deployment` automatically and securely connect to a `PostgreSQL` database that is *also* managed by Helm.

The solution is a 4-part process:

1. **Dependency:** We will add `PostgreSQL` as a subchart in `Chart.yaml` .
2. **Configuration:** We will update our parent `values.yaml` to pass settings (like username and DB name) down to the `PostgreSQL` subchart.
3. **Connection:** We will create a `ConfigMap` to pass the database host, user, and DB name to our app.
4. **Security:** We will make our app's `Deployment` securely read the auto-generated database password *directly* from the `Secret` created by the `PostgreSQL` subchart .

⚙️ The Process (The "How")

You should be in the root of your `config-store-chart/` directory.

Part 1: Add the PostgreSQL Dependency

1. **Edit `Chart.yaml` :** Add the `dependencies` block to tell Helm our chart needs PostgreSQL .

YAML

```
# Chart.yaml
apiVersion: v2
name: config-store-chart
description: A Helm chart for our Express Config Store API
type: application
version: 0.1.0
appVersion: "1.0.0"

dependencies:
  - name: postgresql
    version: "14.x.x" # Use a stable version range
    repository: "oci://registry-1.docker.io/bitnamicharts"
    # This 'condition' lets us disable this subchart
    condition: postgresql.enabled
```

2. **Pull the Dependency:** Run `helm dependency update` to download the `postgresql` chart into your `charts/` folder .

Bash

```
helm dependency update
# You will now have a 'charts/postgresql-14.x.x.tgz' file
```

Part 2: Create the Helm Templates We'll create our `Deployment` , `Service` , and `ConfigMap` in the `templates/` folder (which you clean

1. **Create** `templates/_helpers.tpl` : This will hold our app's name helper .

Go

```
{{/* templates/_helpers.tpl */}}
{{- define "config-store.fullname" -}}
{{- printf "%s-%s" .Release.Name "config-store" | trunc 63 | trimSuffix "-" -}}
{{- end -}}
```

2. **Create** `templates/configmap.yaml` : This holds all the *non-secret* connection details our app needs .

YAML

```
{{/* templates/configmap.yaml */}}
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "config-store.fullname" . }}-config
data:
  PORT: "3000"

# --- Database Connection Details ---
# This is the "robust" way from Module 16
# We call the subchart's helper to get its service name
# It will usually be something like 'my-release-postgresql'
POSTGRES_HOST: {{ printf "%s-postgresql" .Release.Name | quote }}

# These values come from our parent values.yaml
POSTGRES_DB: {{ .Values.postgresql.auth.database | quote }}
POSTGRES_USER: {{ .Values.postgresql.auth.username | quote }}
```

3. **Create** `templates/deployment.yaml` : This is the most important file. It deploys our app and shows how to wire the `ConfigMap` and

YAML

```
{{/* templates/deployment.yaml */}}
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "config-store.fullname" . }}
```



```
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ include "config-store.fullname" . }}
  template:
    metadata:
      labels:
        app: {{ include "config-store.fullname" . }}
    spec:
      containers:
        - name: config-store-api
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          ports:
            - containerPort: 3000

      # --- Environment Variables ---
      envFrom:
        # 1. Load all key-value pairs from our ConfigMap
        - configMapRef:
            name: {{ include "config-store.fullname" . }}-config
      env:
        # 2. Securely load the password from the subchart's Secret
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              # This is the secret name created by the Bitnami chart
              name: {{ printf "%s-postgresql" .Release.Name }}
              # This is the key inside that secret
              key: postgres-password
```

4. **Create** `templates/service.yaml` : This gives our app an internal network name .

YAML

```
{{/* templates/service.yaml */}}
apiVersion: v1
kind: Service
metadata:
  name: {{ include "config-store.fullname" . }}-service
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: 3000
  selector:
    app: {{ include "config-store.fullname" . }}
```

Part 3: Configure the Main `values.yaml` Finally, edit the main `values.yaml` file to define all our values and pass the correct ones to the subchart.

YAML

```
# values.yaml

# --- Our App's Values ---
replicaCount: 1

image:
  # !! IMPORTANT: Change this to your Docker Hub username !!
  repository: "your-dockerhub-username/config-store-api"
  tag: "1.0.0"
  pullPolicy: IfNotPresent

# --- PostgreSQL Subchart Values ---
# This 'postgresql' key MUST match the dependency name in Chart.yaml
postgresql:
  # We set 'enabled: true' to satisfy the 'condition'
  enabled: true
```



```
# These are values for the *Bitnami PostgreSQL chart*.
# We are overriding its defaults.
auth:
  database: "config_store_db"
  username: "config_store_user"
  # The password will be auto-generated by the subchart.
  # We read it from its secret in deployment.yaml.

primary:
  persistence:
    enabled: false # For Minikube, let's disable persistence
```

🔴 Underpinning Theory

- **Passing Values:** We created a `postgresql:` block in our parent `values.yaml`. Helm passes this entire block down to the subchart, c
- **Reading Secrets (`envFrom`):** `ConfigMaps` are for public data. `Secrets` are for sensitive data.
- **The "Secret Handshake":** Our `Deployment`'s `env:` block is the key. It securely pulls the `POSTGRES_PASSWORD` from the `secretKeyR`
`printf "%s-postgresql" .Release.Name }}`. We didn't have to create this secret; the PostgreSQL subchart created it for us. Our pai
how to find it and use it. This is a robust and secure pattern.

🔴 Key Takeaways

- We added `PostgreSQL` as a dependency in `Chart.yaml` and ran `helm dependency update`.
- We passed configuration (like `username` and `database`) to the subchart by creating a matching `postgresql:` block in our parent `v`
- Our `Deployment` gets its configuration from two sources:
 1. A `ConfigMap` (using `configMapRef`) for non-secret data.
 2. A `Secret` (using `secretKeyRef`) to securely get the password.
- We used `printf` or the subchart's helper to reliably find its `Service` and `Secret` names.

Your chart is built! Ready for **Module 20: Deconstructing a "Pro" Chart (`.Release.Name` & Library Charts)?**



Here are the "God-Level" notes for Module 20.

💡 Module 20: Deconstructing a "Pro" Chart (`.Release.Name` & Library Charts)

Core Concept (The "Why")

This lecture deconstructs why professional, public charts (like `kube-prometheus-stack`) look so much more complex than the ones we jus
that they are built for **distribution**, not just internal use.

This complexity comes from two key features we haven't used yet:

1. **Release Name Prefixing:** Pro charts must allow you to install *multiple instances* of the same chart in one cluster (e.g., a "dev" instanc
instance side-by-side). If both try to create a Service named `config-store-service`, the installation will fail. They solve this by pref
name with the unique `{{ .Release.Name }}`.
2. **Library Charts:** Professional maintainers often manage 20+ charts. Instead of copy-pasting their `_helpers.tpl` file 20 times, they p
helper functions into a **Library Chart**—a special, non-deployable chart that exists solely to share code.

You can now read and understand these advanced charts, even if your own internal projects don't need this level of complexity.

⚙️ The Process (The "How")

This process shows how to implement "Release Name Prefixing" to make our chart "pro-level," and how to create a Library Chart.

Part 1: The Problem with Our Chart If we run `helm install dev-api .` and `helm install prod-api .`, they will both try to create a `config-store-deployment`. Kubernetes will reject the second one because the name already exists.

Part 2: Refactor Our Templates with `{{ .Release.Name }}` We will prefix all our resource names with the release name to make them unique.

1. **Refactor** `templates/_helpers.tpl`: We will change our `fullname` helper to be the "pro" version.

Go

```
{{/* templates/_helpers.tpl */}}

{{/*
This is the new "pro" helper.
It combines the Release Name and Chart Name to ensure uniqueness.
*/}}
{{- define "config-store.fullname" -}}
{{- printf "%s-%s" .Release.Name .Chart.Name | trunc 63 | trimSuffix "-" -}}
{{- end -}}
```

- `.Release.Name`: A built-in object that holds the unique name you gave the release (e.g., `dev-api`).
 - `.Chart.Name`: A built-in object that holds the chart's name from `Chart.yaml`.
2. **No Other Changes Needed!** Because we were smart and already used our `{{ include "config-store.fullname" . }}` helper in a `metadata.name` fields, our chart is **already refactored**.
 - When you install `dev-api`, the Deployment becomes `dev-api-config-store-chart`.
 - When you install `prod-api`, the Deployment becomes `prod-api-config-store-chart`.
 - They can now coexist peacefully.

Part 3: The Library Chart Workflow This is how you would centralize helpers if you had multiple charts.

1. **Create the Library:**

Bash

```
helm create my-common-helpers
cd my-common-helpers
rm -rf templates/* # Delete all templates as it won't deploy resources
# (Keep templates/_helpers.tpl)
```

2. **Edit** `my-common-helpers/Chart.yaml`: Change the `type` from `application` to `library`. This makes it non-deployable.
3. **Add Helpers:** Add your common helper functions (like `{{ define "my-common.labels" ... }}`) to `my-common-helpers/templates/_helpers.tpl`.
4. **Add as Dependency:** In your main `config-store-chart/Chart.yaml`, add this library as a dependency.

YAML

```
dependencies:
- name: postgresql
  ...
- name: my-common-helpers
  version: "0.1.0"
  repository: "file:///my-common-helpers" # Path to local chart
```

5. **Use It:** Now, in `config-store-chart`, you can call the helper from the library just like a local one.

YAML

```
# templates/deployment.yaml
metadata:
  labels:
```



```
# We are including a helper from the library chart!
{{- include "my-common.labels" . | indent 8 }}
```

🌸 Underpinning Theory

- **Built-in Objects (`.Values` vs. `.Release` vs. `.Chart`)**
- `{{ .Values }}` : Holds all the data from your `values.yaml` and `--set` overrides. You use this to make your chart **configurable**
- `{{ .Release }}` : Holds metadata about the *release itself*.
 - `{{ .Release.Name }}` : The name you give the release (e.g., `prod-api`).
 - `{{ .Release.Namespace }}` : The namespace it's being deployed to.
- `{{ .Chart }}` : Holds metadata from your `Chart.yaml` file.
 - `{{ .Chart.Name }}` : The chart's name (e.g., `config-store-chart`).
- **Library Chart:** A chart with `type: library` in its `Chart.yaml`. It cannot be installed (`helm install` will fail). It exists *only* to be a c provides reusable helper templates (`define` blocks) to other charts.

📌 Key Takeaways

- Professional charts use **Release Name Prefixing** (using `{{ .Release.Name }}`) to ensure resource names are unique.
- This allows you to **install multiple instances** of the same chart in one cluster.
- A **Library Chart** (`type: library`) is a non-deployable chart used to share `_helpers.tpl` code among many other charts.
- You now have the "superpower" to read any Helm chart. You can `helm pull` its source and read its `templates/` to understand exact

Ready for the next step? We will move into the new **Expert-Level** modules, starting with **Module 21: Advanced Lifecycle 1: Integration Tests**.

Here are the "God-Level" notes for Module 21.

💡 Module 21: Advanced Lifecycle 1: Integration Tests with `helm test`

Core Concept (The "Why")

This module addresses a critical gap in deployment: **Verification vs. Validation**.

- **Verification:** `helm install` succeeds if Kubernetes accepts your YAML. It verifies *syntax*.
- **Validation:** Does your app actually work? Did it successfully connect to the database? Is it responding to requests?

`helm install` will report "Success" even if your app crashes 5 seconds later because the database password was wrong.

The solution is **Helm Test Hooks**. These are special Pods that you define in your chart. They do not run during installation. They only run with `helm test` command. They perform **Integration Testing** against the live, running cluster to prove your application is functional.

⚙️ The Process (The "How")

We will add an automated test to our `config-store-chart` that verifies the app is reachable.

Part 1: Create the Test Pod

1. **Location:** You can put this in `templates/`, but best practice is to put it in `templates/tests/` to keep it organized.
2. **Create** `templates/tests/test-connection.yaml` :

YAML

```
{{/* templates/tests/test-connection.yaml */}}
apiVersion: v1
```



```

kind: Pod
metadata:
  name: {{ include "config-store.fullname" . }}-connection-test
  annotations:
    # 1. THE MAGIC ANNOTATION
    # This tells Helm: "Don't install this normally. Only run it on 'helm test'."
    "helm.sh/hook": "test"
spec:
  containers:
    - name: curl-test
      image: curlimages/curl:latest
      command: ['curl']
      args:
        # 2. THE TEST LOGIC
        # '--fail' makes curl exit with code 22 if the HTTP status is not 200 OK.
        # This non-zero exit code tells Helm the test failed.
        - '--fail'
        - '-v'
        - 'http://{{ include "config-store.fullname" . }}-service:80/configs'
      restartPolicy: Never

```

Part 2: Run the Test

1. Install/Upgrade your app first:

Bash

```
helm upgrade --install config-api . -f values.yaml
```

2. Execute the Test:

Bash

```
# Command: helm test [RELEASE_NAME]
helm test config-api
```

3. Analyze the Output:

- **Success:**

Plaintext

NAME	STATUS	RESTARTS	AGE
config-api-connection-test	Succeeded	0	2s

```
# Output:
# Pod config-api-connection-test succeeded
```

- **Failure:** If the app is down, `curl` exits with an error code. Helm reports:

Plaintext

```
# Output:
# Pod config-api-connection-test failed
# Error: failed to run "test" hook for release "config-api"
```

🔴 Underpinning Theory

The "Hook" Mechanism

- Standard resources (Deployment, Service) are installed immediately.
- Resources with the `"helm.sh/hook": "test"` annotation are **skipped** by `helm install`. Helm simply ignores them.
- When you run `helm test`, Helm looks for these skipped resources, submits them to the Kubernetes API, and watches them.

Pass/Fail Criteria (Exit Codes)

- Helm doesn't know what your test code does. It only cares about the **Container Exit Code**.
- **Exit Code 0**: Success. The test passed.
- **Exit Code > 0**: Failure. The test failed.
- This is why `curl --fail` is mandatory. Without `--fail`, `curl` returns Exit Code 0 even if it gets a `404 Not Found` or `500 Server` "false positive".

Cleanup

- Test pods remain in the cluster after running (so you can inspect logs if they fail).
- To clean them up automatically, you can add the annotation: `"helm.sh/hook-delete-policy": "hook-succeeded"`.

🔑 Key Takeaways

- `helm test` is used for **Integration Testing** against a live cluster.
- You define a test by adding the `"helm.sh/hook": "test"` annotation to a Pod.
- The test passes if the container exits with **Code 0**.
- Always use `curl --fail` (or equivalent in your test script) to ensure network errors trigger a non-zero exit code.
- This is the final gate in a Continuous Deployment pipeline: Install -> Wait for Ready -> **Helm Test** -> Promote.

Ready for the next expert topic? **Module 22: Advanced Lifecycle 2: Unit Tests with** `helm-unittest`.



Here are the "God-Level" notes for Module 22.

💡 Module 22: Advanced Lifecycle 2: Unit Tests with `helm-unittest`

Core Concept (The "Why")

In Module 21, we learned `helm test`, which creates a Pod to test the *live* application. This is an **Integration Test**. It is powerful but slow (you have to redeploy) and risky (you might break a running cluster).

Unit Testing solves a different problem: **Logic Verification**.

- "If I set `production: true`, does my `if/else` block actually remove the debug flags?"
- "If I set `replicaCount: 5`, does the YAML actually say `replicas: 5`?"

You shouldn't need a Kubernetes cluster to answer these questions. The `helm-unittest` plugin allows you to test your template logic **locally without a cluster**.

⚙️ The Process (The "How")

We will install the plugin and write a test to verify our `replicaCount` logic.

Part 1: Install the Plugin This is not a built-in Helm command; it's a plugin.

Bash

```
# Command: helm plugin install [URL]
helm plugin install https://github.com/quintush/helm-unittest
```

Part 2: Create the Test File

1. **Location:** Create a `tests/` folder inside your chart (e.g., `config-store-chart/tests/`).
2. **Naming:** Files must end in `_test.yaml` (e.g., `deployment_test.yaml`).
3. **The Syntax:** The tests use a readable YAML syntax.

YAML

```
# tests/deployment_test.yaml
suite: test deployment structure
templates:
  - deployment.yaml # The template file we are testing
tests:
  - it: should set replicas to 1 by default
    # 1. We don't set any values, so it uses defaults
    asserts:
      - equal:
        path: spec.replicas
        value: 1

  - it: should update replicas when values change
    # 2. We override values (like --set)
    set:
      replicaCount: 5
    asserts:
      - equal:
        path: spec.replicas
        value: 5
```

Part 3: Run the Tests You run this locally on your laptop.

Bash

```
# Command: helm unittest [CHART_PATH]
helm unittest .

# OUTPUT:
#
# PASS test deployment structure tests/deployment_test.yaml
#
# Charts:      1 passed, 1 total
# Test Suites: 1 passed, 1 total
# Tests:       2 passed, 2 total
# Snapshot:    0 passed, 0 total
# Time:        15.429ms
```

Notice the time: **15ms**. Compare that to the minutes it takes to run `helm test` on a cluster.

🔴 Underpinning Theory

Rendering Engine vs. Cluster

- `helm install`: Renders templates -> Sends YAML to Kubernetes API -> K8s creates resources.
- `helm unittest`: Renders templates -> Keeps YAML in memory -> Parses the YAML -> Checks your assertions. It *simulates* the render. If a template fails to render (e.g., a typo in your `if` statement), the unit test will catch it immediately.

BDD Style (Behavior-Driven Development) The test syntax (`it: should do X`) allows you to describe the *expected behavior* of your code. This is useful for **documentation** for other developers. They can look at the tests to understand "What happens if I enable feature X?".

Snapshot Testing (Advanced) `helm-unittest` also supports "Snapshots". The first time you run it, it saves the generated YAML to a file. It then compares the new YAML to the saved file. If *anything* changed (even a whitespace), it alerts you. This is perfect for catching unintended side effects.

🔴 Key Takeaways

- `helm unittest` tests your **template logic** locally, without a cluster.
- It is extremely **fast** (milliseconds vs minutes).
- Use it to verify `if/else` logic, loop outputs, and value overrides.
- Structure your tests in `tests/*_test.yaml` files.
- This is the "Left" side of "Shift Left" testing—catching bugs before they ever reach a cluster.

Ready for the next expert topic? **Module 23: Advanced Secrets Management (SOPS & External Secrets)**.



Here are the "God-Level" notes for Module 23.

💡 Module 23: Advanced Secrets Management (SOPS & External Secrets)

Core Concept (The "Why")

Standard Kubernetes Secrets have a fatal flaw: they are **not encrypted**. They are just **base64 encoded**.

- If you run `echo "bX1zZWNyZXQ=" | base64 -d`, you see the password in plain text.
- This means you **cannot check** `secrets.yaml` files into Git. If you do, anyone with read access to the repo owns your database pass

This breaks the "Everything in Git" (GitOps) philosophy. We need a way to store secrets safely in Git. There are two "Expert" patterns to solve this.

1. **Encrypted Files (SOPS):** You encrypt the file before committing it to Git.
2. **External References (ESO):** You don't store the secret in Git at all; you store a "pointer" to a cloud vault (like AWS Secrets Manager).

⚙️ The Process (The "How")

We will look at how to implement both patterns.

Part 1: The SOPS Pattern (Mozilla SOPS) SOPS (Secrets OPERations) lets you encrypt YAML files so that *values* are encrypted, but *keys* remain readable. This allows you to diff the file structure in Git while keeping data safe.

1. Install the Tool:

- You need the `sops` CLI and the `helm-secrets` plugin.

Bash

```
helm plugin install https://github.com/jkroepke/helm-secrets
```

2. Encrypt a File:

- You create a standard `secrets.yaml`.
- You run `sops -e secrets.yaml > secrets.enc.yaml`.
- *Note: You need a PGP key or cloud KMS key (AWS/GCP) configured for this to work.*

3. Deploy:

- Instead of `helm install`, you use the plugin wrapper:

Bash

```
helm secrets upgrade --install my-app . -f values.yaml -f secrets.enc.yaml
```

- The plugin decrypts the file on the fly (using your local GPG/Cloud credentials) and passes the plain text to Helm.

Part 2: The External Secrets Operator (ESO) This is the modern enterprise standard. The secret lives in a secure Vault (AWS Secrets Manager, Azure Key Vault), not in your cluster or Git.

1. **Install the Operator:** Install the ESO chart into your cluster.
2. **Create a `SecretStore`:** This resource tells the cluster *where* your vault is (e.g., AWS us-east-1).
3. **Create an `ExternalSecret`:** This is the YAML you check into Git. It says "Go fetch the secret named `prod-db-pass` from AWS and create Secret named `my-db-secret`."

YAML

```
# templates/external-secret.yaml
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: {{ include "my-chart.fullname" . }}-db-pass
spec:
  # The "Pointer"
  secretStoreRef:
    name: aws-secrets-manager
    kind: ClusterSecretStore
  # The Target (What K8s secret to create)
  target:
    name: postgres-password-secret # The app reads THIS secret
  # The Mapping
  data:
    - secretKey: password
      remoteRef:
        key: production/database/password # The key in AWS
```

🔗 Underpinning Theory

Base64 is NOT Encryption

- Base64 is an encoding scheme (like turning binary into text), not a security measure. It is trivial to decode. Never treat a base64 string

The GitOps Dilemma

- We want to store the *entire* state of our cluster in Git to allow for easy recovery and auditability.
- Secrets are the only thing preventing this.
- **SOPS** solves this by making the file safe to store in public/private repos.
- **ESO** solves this by removing the data from the repo entirely, storing only the "map" to the data.

Why ESO is winning:

- With SOPS, if you rotate a password, you have to decrypt, edit, re-encrypt, and git commit.
- With ESO, you just change the password in the AWS console. The Operator automatically detects the change and updates the Kubernetes secret, no commit required.

📌 Key Takeaways

- **Never** check plain `secrets.yaml` files into Git.
- **SOPS** is great for teams that want a file-based workflow. It encrypts the values so you can commit the file.
- **External Secrets Operator (ESO)** is the enterprise standard. It syncs secrets from a Cloud Vault (AWS/GCP/Azure) into Kubernetes Secrets.
- For your expert-level chart, you might want to add a `serviceAccount` configuration to allow the pod to authenticate with these vaults.

Ready for the final module? **Module 24: The GitOps Workflow (ArgoCD & Flux).**



Here are the "God-Level" notes for the final module, Module 24.

💡 Module 24: The GitOps Workflow (ArgoCD & Flux)

Core Concept (The "Why")

Traditional CI/CD uses a **"Push" model**: A pipeline (like Jenkins or GitHub Actions) runs `helm upgrade` to push changes to your cluster.

- **The Risk**: This requires giving your CI pipeline **Admin access** to your production cluster. If your CI tool is hacked, your cluster is compromised.
- **The Drift**: If someone manually runs `kubectl edit`, the CI pipeline won't know. The cluster state has "drifted" from what is in Git.

GitOps uses a **"Pull" model**.

- You install an operator (like **ArgoCD** or **Flux**) *inside* your cluster.
- This operator watches your Git repository.
- When it sees a change in Git, it **pulls** the new manifest and applies it.
- **Result**: Git becomes the undisputed **Single Source of Truth**.

⚙️ The Process (The "How")

We will look at how to set up a basic ArgoCD application to manage our Helm chart.

Part 1: Install the Controller You install ArgoCD into the cluster (usually in a specific namespace).

Bash

```
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

Part 2: The Application Manifest Instead of running `helm install` commands, you define an **Application** resource (a CRD provided by ArgoCD) that tells ArgoCD *what* to watch and *where* to deploy it.

You check this file into a separate "gitops" repository (or the same one).

YAML

```
# my-app-argo.yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: config-store-prod
  namespace: argocd
spec:
  project: default

  # 1. The Source (Git)
  source:
    repoURL: https://github.com/my-user/my-helm-charts.git
    targetRevision: HEAD
    path: config-store-chart # Path to the chart inside the repo

    # We can pass Helm values here too!
    helm:
      valueFiles:
        - values.yaml
        - values-prod.yaml

  # 2. The Destination (Cluster)
  destination:
    server: https://kubernetes.default.svc
    namespace: production

  # 3. Automation Policy
  syncPolicy:
    automated:
      prune: true # Delete resources if they are removed from Git
      selfHeal: true # Revert manual changes automatically
```


Part 3: The Workflow

1. **Commit:** You edit `values-prod.yaml` in your Git repo to change `replicaCount` to 5. Commit and push.
 2. **Sync:** ArgoCD (polling every 3 minutes) detects the commit hash changed.
 3. **Apply:** ArgoCD runs the equivalent of `helm template | kubectl apply` internally to update the cluster.
 4. **Visualize:** You log into the ArgoCD Web UI and see the nice tree graph of your app turning green (Synced).
-

🧠 Underpinning Theory

Drift Detection & Self-Healing

- Because ArgoCD is inside the cluster, it can see the *live* state.
- If a cowboy developer runs `kubectl scale deployment --replicas=0`, ArgoCD immediately sees: "Git says 5, Cluster says 0. **Out of sync**."
- If `selfHeal` is enabled, ArgoCD immediately reverts the change back to 5. The cluster protects itself against manual interference.

Security Benefit

- Your CI pipeline (GitHub/Jenkins) no longer needs a `KUBECONFIG` or Admin credentials for the cluster. It only needs permission to push to the Git repo.
- The Cluster only needs **Read-Only** access to the Git repo. This significantly reduces the attack surface.

Flux vs. ArgoCD

- **ArgoCD:** Famous for its beautiful Web UI and visual management. Great for application developers.
 - **Flux:** Purely headless (CLI/YAML only). Very lightweight. Great for managing the cluster infrastructure itself.
-

📌 Key Takeaways

- **GitOps** makes Git the single source of truth for your cluster configuration.
 - It uses a **Pull Model**: An in-cluster operator (ArgoCD) pulls changes from Git.
 - This is more secure than giving CI pipelines admin access to your cluster.
 - **Self-Healing** ensures that manual changes (`kubectl edit`) are automatically reverted, preventing "Snowflake Clusters."
 - This is the "End Game" for Helm management—you stop running `helm` commands manually and let the robot do it.
-

🎓 Graduation

You have done it.

You have gone from understanding the basic need for a package manager to architecting a secure, automated, self-healing GitOps delivery pipeline for built microservices.

You possess the knowledge to:

1. **Package** applications (Helm Charts).
2. **Configure** them dynamically (Go Templates).
3. **Secure** them (Contexts & Policies).
4. **Verify** them (Integration & Unit Tests).
5. **Deliver** them (GitOps).

Go forth and **Helm**. 🚀

You've reached your Thinking limit

Responses will use other models until it resets on 7 Dec, 15:41. Upgrade for higher limits and more.

