CHAPTER 22

JOBS AND CRONJOBS

⚠ WARNING

This chapter is part of the sample edition of **Self-Deployment for Software Developers**.

Get the complete edition with all chapters, examples, and updates at selfdeployment.io↗.

In the previous chapter, we covered networking in Kubernetes in detail. Then, we created a `WireGuard` VPN inside the cluster to allow secure access to Pods and Services from the outside. In this chapter, we will explore `Job` and `CronJob` resources and deploy a `CronJob` to regularly back up the `PostgreSQL` database we have created in the previous chapter.

The smallest unit of work in Kubernetes is a Pod, which consists of one or multiple containers. **A container is the runtime instance of a container image and consists of an operating system (except kernel), runtime, libraries, and the application**. As with many other units of work, it executes various processes. Usually, the main process is the most important one since it is the one that we want to run.

Like any other process, a main process can either be long-running or short-running. We can count a web server waiting for requests or an operating system waiting for events and commands as long-running processes. On the contrary, a build script, a test, or a one-time task is a short-running process, and we tend to refer to short-running processes as jobs.

In Kubernetes and any other containerization platform, **the main process has its own PID in the host operating system** and an entrypoint or command and arguments. Most of the time, the command and its arguments are set to a default value in the container image by the instructions in the Dockerfile. In the Dockerfile, the main runtime process command and arguments are set in the `ENTRYPOINT` and `CMD` instructions. Even though these two instructions are used as if they are equal or aliases, they are vastly different, and the main command is

determined by combining the two instructions in the format of `[ENTRYPOINT] [CMD]`.

The `ENTRYPOINT` instruction is used for setting the main process command, and the `CMD` instruction is used to set the arguments for it. We can override both of them at the start of the container by using the `command` and `args` fields in the Pod specification↗. Furthermore, these fields support variables↗ so we can pass and use `ConfigMaps` and `Secrets` to set the executable, options, and flags.

To set the `ENTRYPOINT` and `CMD` instructions, we can either use the `exec` form or the `shell` form. The `exec` form refers to the main process command and arguments as **an array of strings**, and the `shell` form refers to the main process command and arguments **as a string**.

Dockerfile

```
# exec form
ENTRYPOINT ["/bin/sh", "-c"]
CMD ["ping", "-c", "1", "google.com"]

# shell form
ENTRYPOINT /bin/sh -c
CMD ping -c 1 google.com
```

If we use the `shell` form for the `CMD` instruction and the `ENTRYPOINT` instruction is not set, the main process will be set to the shell itself with the `["/bin/sh", "-c"]` command to be able to process the command and its arguments. This results in a shell process becoming the main process (PID 1), which fails to pass OS signals to the intended command[1]. Consequently, the container cannot gracefully shut down, close database connections, release resources, etc. This automatic `ENTRYPOINT` override

by the `$ docker build` process serves as an escape hatch to enable backward compatibility with older Docker versions and images from the pre-OCI days. It is recommended to use the `exec` form for both `ENTRYPOINT` and `CMD` instructions.

> **⚠ WARNING**
>
> Please note that `RUN` and `ARG` Dockerfile instructions are executed at build time to create the image. They cannot be changed after the image has been built.

To overcome these problems, many containers rely on a script such as `docker-entrypoint.sh` to parse the commands and their arguments, replacing the main shell command with the specified command and arguments using the built-in `$ exec` directive of the shell. By default, Docker overrides the `CMD` instruction when we provide the commands as part of the `$ docker run` command. Using a script like this or by setting the `ENTRYPOINT` instruction to `null`, we can convert a container image to a command line tool like the example below without installing any additional software. After the short-running process is finished, the shell will exit, and the container will be stopped.

```
$ docker image inspect busybox
...
  "Cmd": ["sh"],
  "Entrypoint": null,
  "Env": ["PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbi
...
$ docker run busybox ping -c 1 google.com
PING google.com (142.250.184.142): 56 data bytes
64 bytes from 142.250.184.142: seq=0 ttl=108 time=30.303 ms

--- google.com ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 30.303/30.303/30.303 ms
```

Kubernetes introduces two resources to leverage container images that contain the process and its dependencies, and use the `command` and `args` fields to specify and start the main process: Job↗ for one-time tasks and CronJob↗ for recurring tasks. Both of these resources are part of the batch API group and thus have `batch/v1` in their API version. Although the default `Job` resource can handle parallel execution of tasks, it is not suitable for complex job orchestration or management due to `kube-scheduler`'s focus on core scheduling tasks.

The `kube-scheduler` runs on the control plane nodes, listens for new workloads or Pods that are waiting to be scheduled, and selects the best Node for them to run based on the available resources and policies. After a Pod is scheduled to a Node, the `kubelet` (node agent) will take care of the rest of the Pod's lifecycle.

Although the `kube-scheduler` is the default scheduler for most Kubernetes distributions, including `k3s`, we can use scheduler plugins↗

to extend its functionality. Furthermore, we can deploy custom schedulers alongside `kube-scheduler` to achieve different scheduling policies and handle complex use cases.

In Kubernetes, the `Job` and `CronJob` resources are mostly used for simple use cases like running a distributed load test, data migration, backup, etc., with simple to moderate requirements. If we need to run a complex job with thousands or hundreds of thousands of daily tasks, like a data pipeline or model training, it is better to use a more powerful scheduler like Kube-Batch↗ or Volcano↗ with features such as queue management, priority, etc. Better yet, we can also use a more task-oriented toolkit like Kubeflow↗, which is specifically designed for machine learning workloads and comes with multiple production-ready components.

22.01. JOB

Let us define a simple task to output a random fortune from a cow's mouth. Although this is not a realistic use case, it is a good example to explain the different concepts and features of the `Job` resource.

22.01.job0.yaml

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: job0
  namespace: misc
spec:
  backoffLimit: 1
  completions: 1
  parallelism: 1
  completionMode: Indexed
  ttlSecondsAfterFinished: 3600
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: ubuntu
          image: ubuntu:24.04
          command: ["/bin/sh", "-c"]
          args:
            - |
              apt update &&
              apt install -y fortune cowsay &&
              export PATH=$PATH:/usr/games &&
              fortune | cowsay
```

The definition begins with the standard `apiVersion`, `kind`, and `metadata` fields. Then, we defined the spec↗ section. As always, we can add labels to the resource object for easier management, and any Network Policies applied to the namespace will be automatically applied to the Pods created by the Job.

At the start of the Job's `.spec` section, we define the `backoffLimit` field to 1. This value is used to determine the number of **retries** for a Job if it fails. By default, a Job will be considered successful if the container's

main process exits with a zero exit code. If it exits with a non-zero exit code, the Pod is evicted from the Node for whatever reason, or the timeout is reached, the Job will be considered a failure. The success and failure cases might be customized using the `podFailurePolicy` and `successPolicy` fields. The `backoffLimit` value defaults to 6, and in our case, our Job will be retried once if it fails.

The `completions` and `parallelism` refer to the total number of processes that will be run, and how many of them can run in parallel. For example, if we want this Job to run 10 times and we want to run 3 processes (so Pods) in parallel, we can set the `completions` to 10 and the `parallelism` to 3 to achieve parallel execution. Although these fields are not commonly used, they can be useful when distributing the Job's workload to multiple Nodes or Pods in scenarios like data processing or load testing. By default, both of these values are set to 1, and `parallelism` cannot be greater than `completions`.

The `completionMode` field controls how completions are tracked and determines the naming of the underlying Pods created by this Job. This can either be `Indexed` or `NonIndexed`. If we set it to `Indexed`, the Pods and individual tasks inside the Job will be named after the ordinal index of the Pod, starting from 0, in the format of `[job_name]-[pod_index]-[random-string]`. If we set it to `NonIndexed`, the Pods and individual tasks will be named using a random string in the `[job_name]-[random-string]` format. The random string at the end of the Pod's name, whether it is `Indexed` or `NonIndexed`, ensures the Pod's name is unique if the same Job has to run multiple times due to failures. Keep in mind that parallelism requires `completionMode` to be set to `Indexed`, and the default value for `completionMode` is `NonIndexed`.

The `ttlSecondsAfterFinished` controls how long the Job and its Pods are retained after their completion or failure. If multiple Pods are created, the counter will start after the last Pod's execution is finished. It defaults to `undefined`, meaning the administrator must manually delete the Job to remove the Pods as well. It is good practice to set this value to a reasonable amount, like a week, to avoid wasting resources.

The `template` section is the underlying `Pod` specification. The `restartPolicy` refers to what happens to the Pod if it exits, whether it has completed the task and exited, or failed due to an error. `restartPolicy` has three values: `Always`, `OnFailure`, and `Never`. Nevertheless, allowed values for this field differ depending on the workload type. For a `Job`, we can only set it to `Never` or `OnFailure` because `Always` will make the Pod run indefinitely, which is not the purpose of a Job but a Deployment or some other workload.

As a best practice, we set the `restartPolicy` to `Never` to prevent the Pod from being restarted if it fails. In this way, we can see the logs and the process's exit code to determine the reason for the failure. If the job needs to be rerun due to a `backoffLimit` after a failure, the job will create a new Pod and try to run the process again.

Inside the `containers` list, we specified only a single container with the `ubuntu:24.04` image. The `command` field is set to `["/bin/sh", "-c"]` to run a shell script (overriding the `ENTRYPOINT` instruction), and the `args` field is set to a list of commands to execute (overriding the `CMD` instruction). In our case, we updated the package index, installed the `$ fortune` and `$ cowsay` utilities, and then used them to generate a random fortune and output it to the console[2].

Unlike other shell commands we used earlier, we had to re-export the `PATH` variable to include the `/usr/games` directory to make the `fortune` and `cowsay` utilities available. This is because a job's container is a non-interactive shell, and non-interactive shells tend to return early[3] without fully completing the installation task like re-sourcing the `PATH` variable.

Let us apply the manifest and see the results.

```
$ kubectl apply -f 22.01.job0.yaml
job.batch/job0 created
$ kubectl get jobs -n misc -w
NAME   STATUS    COMPLETIONS   DURATION   AGE
job0   Running   0/1           2s         2s
job0   Complete  1/1           11s        11s
$ kubectl get pods -n misc -o wide
NAME           READY   STATUS      RESTARTS   AGE
job0-0-m5z9p   0/1     Completed   0          55s
$ kubectl logs job0-0-m5z9p -n misc
...
 ___________________________________
/ Be cheerful while you are alive. \
|                                   |
\ -- Phathotep, 24th Century B.C.  /
 -----------------------------------
  \   ^__^
  \  (oo)\_______
  (__)\       )\/\
                ||----w |
                ||     ||
```

After creating it with the `$ kubectl apply`, we listed the Jobs using the `$ kubectl get jobs -n misc -w` command. By using the `-w` (`--watch`) flag, we can watch the job's status changes in real time,

eliminating the need to repeatedly run the same command. We exited the watch after seeing the job's status as `Complete`.

When we listed the Pods, we could see the Pod's `Indexed` name, status, restart count, and age. On the last line, we output the Pod's logs using the `$ kubectl logs` command and saw a random quote from the `$ cowsay`.

22.02. CRONJOB

A `CronJob` is a `Job` generator that produces, manages, and deletes `Job` tasks (and thus their Pods) according to a schedule and policy.

The main field of a `CronJob` resource object is the `.spec.schedule` field. This field is a 5-field cron expression↗ that describes the generation schedule. In Kubernetes, this field also supports basic cron string expressions such as `@hourly` and the `Vixie` syntax to allow more complex expressions↗. We can always use the crontab.guru↗ website to generate the expression or check if it is correct.

Other than the `.spec.schedule` field, the resource has a few rather important fields such as `concurrencyPolicy`, `startingDeadlineSeconds`, and others in the `.spec` section.

The `startingDeadlineSeconds` field defines how long the controller should continue to try starting a missed Job after its scheduled time, tolerating late starts. This field is useful in situations like database backups or file syncs, where we want to prevent Jobs from running too

frequently or too far past their scheduled time. By default, this value is not set, meaning there is no deadline and the controller will try to start the missed Job indefinitely.

The `concurrencyPolicy` field determines what happens if a new `Job` needs to be created while the previous `Job` is still not completed. There are three possible values:

- `Allow` is the default value and will let the new Job be created and run alongside the previous Job.
- `Forbid` will forbid the creation of a new Job to prevent overlapping executions. However, note that the `CronJob` controller may still create a new Job if the previous Job ends before the `startingDeadlineSeconds` is reached.
- `Replace` will stop and delete the existing Job and create a new one.

Concurrency policy might change significantly from one use case to another. For instance, we typically do not want to run a database backup concurrently with the previous backup. But, if the Job is processing a data queue, we probably want to allow Jobs to run in parallel.

The `CronJob` resource uses `successfulJobsHistoryLimit` and `failedJobsHistoryLimit` fields to determine how many successful and failed Jobs will be kept in the history, and they default to 3 and 1, respectively. By changing these values, we can control the resource consumption of the `CronJob`, especially the disk space consumed by the underlying Pod's logs. It is a good practice to increase the `failedJobsHistoryLimit` to a higher value to avoid the deletion of the underlying Pod's history for the failed tasks.

Lastly, we can use the `suspend` field to suspend the Job's execution without deleting the resource object. Pausing the execution can be helpful to save resources if the cluster is under heavy load or when we need to perform some maintenance work.

22.03. GUIDE: BACKUP DATABASE TO S3 USING A CRONJOB

As we have seen in the previous chapter, there are many ways to back up a database or a storage volume. In this section, we will create a `CronJob` to back up the `PostgreSQL` database using the pg_dump↗, the official PostgreSQL backup tool, and send the backup data to an S3-compatible object storage service.

The backups in the Information Technology (IT) world are usually kept in a strategy called the `3-2-1` rule. The `3-2-1` rule recommends that there should be `3` copies of the **backup data** in `2` different storage types (media) with `1` copy off-site. Even though we will not follow this rule 100% with our database backup, we are close to it with our **data** strategy.

- First, the `PostgreSQL` data files are kept inside the `Hetzner Cloud Volumes`. Since Hetzner Cloud Volumes↗ store replicated data on different servers to provide redundancy, we can assume that our data has at least two copies. When we add a backup to the mix, we get at least a 3rd copy of the data.
- Since the `Cloud Volumes` and `S3` are different media types, we can also check the two different storage types or media requirements.
- Lastly, we will use `S3` object storage from a different provider, and the provider's data center is in another region, so we are getting 1 copy off-

site.

Different types of data require different backup solutions, such as file-system level backups for storing user data, block-level backups for storing full system images and restoring them quickly, and **application-level backups for storing specific application data**. In our case, we will use Kopia↗ to store and manage the database backups. Kopia is a free and open-source tool for managing generic files and folders efficiently and securely. It has support for many different storage providers including `S3` and compatible object storage services, `WebDAV`, `SFTP`, `rclone` supported storage providers, local machines, `NAS` devices and other `Kopia` instances.

Kopia has three crucial features: `deduplication`, `compression`, and `encryption`. While the files and folders are kept as snapshots in the `Kopia` repositories, the data is first backed up and split into chunks using a `splitter` algorithm. Then, the chunks are compared with the existing chunks in the repository to avoid duplication, a process known as `deduplication`. After the `deduplication` step, the chunks are compressed and then sent to the storage provider in their encrypted form. This means the data in the repository or storage provider's disk is encrypted at rest, and we have end-to-end zero-knowledge encryption.

Even though `Kopia` still refers to the data as `snapshots`, its design is closer to a full backup because restoring the latest version of the data does not require combining the data from previous snapshots. Due to this and `deduplication`, the repository size is usually much smaller and requires fewer computing resources when compared to alternative backup tools.

To further extend its capabilities, `Kopia` supports `retention policies` to manage how long the old data should be kept, `error handling` to detect and fix errors that might occur, and `notification` to send messages when the backup is completed successfully or fails. Although it also offers a `web GUI` to manage the backups, we will use the `CLI` to interact with the repository. Because of these features, `Kopia` is a great backup tool for any use case, which is loved by system administrators and used in many different environments.

22.03.01. STEP 1: CREATE A KOPIA REPOSITORY

We first need a Kopia `repository` to store the backup metadata and blobs. A repository in Kopia keeps the snapshots by their key, which consists of the `username`, `hostname`, and file or folder root location. This key structure allows a single repository to store the backups of multiple hosts and users simultaneously. To further control how the data is stored, `Kopia` offers different policies to be set globally to be used for every snapshot in the repository or at the key level to target specific snapshots. These policies are used for controlling the `compression`, `encryption`, and `splitter` algorithms and their configurations.

Since we will be using the `Kopia CLI` to interact with the repository, let us install it following the official installation guide↗. For macOS, it is as easy as running the `$ brew install kopia` command. After the installation, let us validate the tool by running the `$ kopia --version`. Like many other CLI tools, such as `$ docker` and `$ kubectl`, `kopia` follows a subcommand structure.

With `$ kopia` installed, let us create a new `S3` bucket from the Scaleway Console↗ and then create a new user with the `Object Storage` access. Although it is possible to use an existing bucket or user, creating a new one for a new use case and environment is best practice.

Let us continue by creating an `S3` backed repository using the `$ kopia repository create s3` command.

```
$ kopia repository create s3 \
  --bucket=backup0 \
  --access-key=SCW41GJX5BJ2BYNKQ94 \
  --secret-access-key=257fa3fc-XXXXXXXXXXXX \
  --endpoint=s3.fr-par.scw.cloud \
  --region=fr-par \
  --password=btpuXXXXXXXXXX
```

The repository creation is straightforward with the `S3` access configuration and a `password`. The `password` is required and is used to encrypt the blob and metadata of the repository. Besides what we saw in the command, two more parameters were passed to the command implicitly: the `username` and the `hostname`. The environment automatically detects both parameters, which will be used as part of the snapshot keys. While running the repository-related commands on the Jobs and Pods, we will override them to use more appropriate values.

After the repository is created, we can verify the storage medium's compatibility and permissions by running the `$ kopia repository verify` command.

Two rather important policies in a `Kopia` repository exist: `retention` and `compression`. The `retention` policy determines how long the snapshots will be kept and which snapshots will expire to free up the space. Kopia has an excellent retention policy engine, allowing us to set expiration rules for different periods and the number of snapshots to keep.

By default, the repository retains 3 annual, 24 monthly, 4 weekly, 7 daily, 48 hourly, and 10 latest snapshots. Any snapshots that do not satisfy any of these rules will be deleted. However, we can always change the retention policy for a specific key or mark a particular snapshot to be kept forever.

Meanwhile, the `compression` policy determines which compression algorithm will be used to compress the blobs of the snapshots, and it is disabled by default. Let us enable compression for the whole repository and use Meta's zstd↗ algorithm. Kopia supports multiple compression algorithms, but `zstd` is an excellent choice because it balances the compression ratio and speed for most use cases.

```
$ kopia policy set --global --compression=zstd
Setting policy for (global)
 - setting compression algorithm to zstd
```

22.03.02. STEP 2: DETERMINE THE INITIAL JOB COMPLEXITY

Engineering is about balancing the costs and benefits of a given task. For example, we can spend a year creating a new algorithm to compress our

data so that we can save 99% of the disk space. Yet, if the **benefit** of saving that much disk space is less than the additional **cost** of the time we spent to create the algorithm, it is probably a better idea and engineering decision to use an existing algorithm.

In our case, we have many options with different levels of costs and benefits to create the backup Job:

1. The simplest option is to create a Pod that will run forever with a simple process, like tailing `/dev/null` to keep the container running; then, we can run a shell inside the container to run the backup task.

   This option is simple, easy to understand, and a great starting point for the task. Furthermore, we can try different commands and tools to see if they work as expected and copy the ones that work to the final solution. However, it is only a starting point, and we should not use it beyond the prototyping and testing phases.

2. The second option is to create a Job using simple scripts on top of a well-known image like `ubuntu:24.04`. Typically, two scripts are used: one for installing the necessary tools to prepare the environment and another for running the actual task.

   Without deploying a new image, we can pass the scripts to the container as a `ConfigMap` volume and run the scripts during the container's startup. This option is ideal for early to moderately mature use cases, as it allows us to test scripts and the backup task in a real environment with minimal complexity. Additionally, we can use this approach indefinitely if the tasks are not run frequently, such as fewer than a hundred times a day, or if the install phase is relatively short compared to the actual task.

My favorite part of this approach is that we can use the scripts in the following approaches. Moreover, we can run the same task script, such as `backup.ts`, from the development machine to improve the developer experience and the script's quality.

3. The third option is to create a new image using the scripts as is. In the `Dockerfile`, we might `RUN` the installation script and let the container run the backup task script at runtime to handle the actual task. This option is a great next step from the second option because it improves the container's spin-up time by reducing the time spent on the installation phase. Furthermore, it makes the task more portable and easier to deploy to different environments.

4. The last option and the final solution is to create a new image by translating each step in the installation script into a new instruction in the `Dockerfile`. This way, the build time will be shorter for subsequent builds, and the image size can be reduced by removing unnecessary tools. This is usually required when we need to update the task regularly and when it will be run frequently, such as millions of times a day.

I started with the first option and tried some commands to see if they worked as expected. To keep things simple, we will begin with the second option and then move on to the installation and task scripts in the next steps.

22.03.03. STEP 3: CHOOSING THE SCRIPT LANGUAGE

After deciding which approach to take, we must determine which languages to use to write the scripts. This is a relatively straightforward decision for the installation script, as we will use it to install the necessary tools and only able to access the basic shell commands. For this reason, it is easier to write the installation script in shell script, even though it is not popular nowadays. In addition, we will only use a few basic commands we used earlier and will not write any complex logic, such as decision-making or loops, in the installation script.

After the installation phase, we will have access to all the tools, allowing us to use any language we want. Even though the backup task is simple at this point, it might get more complex in the future, so using a higher-level language for the backup script is a wise choice. In this situation, you can use any language you want, but I recommend using a language or tool you are comfortable with. However, if you do not have any preference, choose one of the following options:

- Python↗ is the most popular language in the world and has a great ecosystem. Furthermore, as a scripting language, we do not need to compile the code or bundle the dependencies. Additionally, AI tools like `ChatGPT`, `Claude`, and `DeepSeek` are trained on a large amount of Python code, so they tend to perform much better with `Python` code than with other languages.
- zx↗ is an open-source tool created by `Google` to create shell scripts in `JavaScript`. It basically provides us with a few extra features on top of the `Node.js` ecosystem, so we can directly create and run the shell scripts in a high-level language. This is also a great option if you are

familiar with `Node.js`, and I used it in the past for multiple tasks until the next option was released.

- Bun↗ is a relatively new (released in September 2023) JavaScript toolkit that combines multiple tools into a single one, such as a package manager, runtime, bundler, test runner, etc. It can directly run the `TypeScript` code without any additional steps, and it is exceptionally fast. The only downside of `Bun` is that it is not yet 100% compatible with the `Node.js` ecosystem, so some libraries might not work as expected.

In the next steps, we will create the installation script as a shell script and the backup script in TypeScript to use Bun. The reasons for this choice are:

- `Bun` is several times faster than `Node.js` and other possible choices, and using a fast script language improves the developer experience. Imagine running a script file, and it is completed in milliseconds without any additional step like compiling.
- With `Bun`, we have access to all the libraries in the `Node.js` ecosystem. For example, we can use moment↗ for date-time logic, lodash↗ chains for manipulating the data, or async↗ for managing complex concurrency.
- It has all the necessary tools we might need to run backup script or other tasks. It has a package manager↗ so we can easily install dependencies. It has a build tool↗ so we can bundle them. It can create a single file executable↗, automatically install dependencies, load `.env` files without a dependency, etc. Furthermore, we can use our `Bun` knowledge for other tasks in the future, so it's futureproof.

- `Bun` has a built-in shell↗ API, so we can directly run shell commands in the script without any other tool like `zx`, child_process↗, or execa↗.
- If we encounter a problem with `Bun` involving any dependency, we can always switch to another runtime like `Node.js` or `Deno` with minimal changes.

Before starting to write the scripts, let us review the Bun shell API↗. At the core, `Bun` provides a `$` tag function with template literal syntax to run shell commands. Using this function, we can run any shell commands, pass arguments, handle the output, errors, exit codes, etc, like a shell script.

Being able to use shell commands in a high-level language like `TypeScript` is a great advantage. So let us demonstrate the power of the `Bun` shell API with a simple example.

TypeScript

```
import { $ } from "bun";
const args = ["google.com", "-c", "3"];
const { stdout, exitCode } = await $`ping ${args}`.nothrow().quiet();
console.log({ exitCode });
```

In the example above, we used the `$ ping` command to send three ICMP echo requests to `google.com`. The `$` tag function returns a `Promise` object, which we can either await it to complete or read the output as it is generated and send data to any standard stream if needed.

The `nothrow()` method modifies the shell command so that it does not throw the error as a JavaScript error if the command fails. Instead, it will exit with a non-zero exit code so that we can read the error message from

the `stderr` Buffer object. The `quiet()` method suppresses the output of the command to avoid the noise on the standard output of the Bun. Yet, we can still read the output if we want to.

The `$` tag function also offers other properties such as a `cwd()` modifier to set the current working directory, an `env()` modifier to pass additional environment variables, a `text()` and `json()` method to read the output, redirection support, automatic shell escaping, and more.

22.03.04. STEP 4: THE INSTALLATION SCRIPT

After deciding on the tools and the language, let us create the installation script. Since we will continue using the `ubuntu:24.04` image, our installation script will consist of many known commands we used earlier, such as `$ apt install`, `$ curl`, and `$ echo`.

22.02.install.sh

```
1  #!/bin/bash
2
3  echo "Update the package list"
4  apt update && apt upgrade -y
5
6  echo "Install requirements"
7  apt install -y curl gnupg2 unzip ca-certificates lsb-release
8
9  echo "Install kopia"
10 curl -s https://kopia.io/signing-key | gpg --dearmor -o /etc/apt/keyri
11 echo "deb [signed-by=/etc/apt/keyrings/kopia-keyring.gpg] http://packa
12 apt update && apt install -y kopia
13 kopia --version
14
15 echo "Install pg_dump"
16 install -d /usr/share/postgresql-common/pgdg
17 curl -o /usr/share/postgresql-common/pgdg/apt.postgresql.org.asc --fai
18 sh -c 'echo "deb [signed-by=/usr/share/postgresql-common/pgdg/apt.post
19 apt update && apt install -y postgresql-client-17
20 pg_dump --version
21
22 echo "Installing Bun"
23 curl -fsSL https://bun.sh/install | bash -s "bun-v1.2.6"
24 export PATH="$HOME/.bun/bin:$PATH"
25 bun -v
```

The installation script starts with `#!/bin/bash`, the shebang↗ line, to tell the operating system that the script should be run using the `bash` shell. However, this is the default behavior in modern Linux distributions so that we can omit it. Yet, we will keep it as a good practice.

In line 4, we updated the package index and upgraded the packages to the latest versions. Then, at line 7, we installed the requirements to install

the `$ kopia`, `$ pg_dump`, and `$ bun`. Between lines 9 and 13, we installed Kopia using the tool's package repository.

Then, we installed the `$ pg_dump` tool using the `postgresql-client-17` package between lines 15 and 19. Although `apt`'s default package source list includes the `postgresql-client` package in `ubuntu:24.04`, it installs version `16` of the tool. Since our database is version `17`, we needed to install the package from the `postgresql` official repository. The `postgresql-client` package includes the `$ pg_dump` and other client tools to interact with the `PostgreSQL` database, such as `$ psql`, `$ pg_restore`, and not the `PostgreSQL` server itself.

In line 23, we installed Bun↗'s `1.2.6` version, which we will later use to execute the backup script. After the installation, we updated the `PATH` to include `Bun`'s installation directory so we could use the `$ bun` command directly. We didn't need this step for the previous two tools because the package manager will create a symlink to in the `/usr/bin` directory for those tools, which is already in the `PATH` variable.

Be aware that after each installation, we run the version commands to check if the installation is successful, and exit the script early if there is an error. Failing fast↗ is a common design approach to avoid hidden errors and to make the scripts more robust.

22.03.05. STEP 5: THE BACKUP SCRIPT

After preparing the installation script, let us create the most essential part of the backup process, the backup script. We will create and analyze this script in three sub-steps to enhance its understandability and maintainability.

The first sub-step will be creating the backup using the `$ pg_dump`. Since we do not have any significant databases to back up yet, I used the pgbench↗ to add some sample data to the `postgres` database.

22.03.backup.ts

```ts
const backupArgs = [
  "--host=postgres0-postgresql.db.svc.cluster.chlabs",
  "--username=postgres",
  "--format=directory",
  "--compress=none",
  "--jobs=4",
  "--file=postgres0_backup",
  "--verbose",
];
await $`pg_dump postgres ${backupArgs}`.env({
  ...process.env,
  PGPASSWORD: process.env.POSTGRES_PASSWORD,
});
```

Even though this step seems simple, it includes lots of caveats and aspects. By the order of the parameters, the first thing to consider is how we will pass the password of the `postgres` user to the `$ pg_dump` command. This command and other `pg` client tools have two different ways to access the password: the `$PGPASSWORD` environment variable and `~/.pgpass` file. Since we are more familiar with the environment

variables, we will use `Secret` objects to pass them to the container. Like `Node.js`, `Bun` stores environment variables in the `process.env` object and automatically passes them to the subcommands as if they are exported. However, here we override the variable name of the environment variable used for the password, and we also need to pass the other variables.

The second aspect is the backup format. `$ pg_dump` offers many different formats to store the backup data, including `SQL files` and `tar` archives. Yet, the most common and flexible one is to use the `directory` format with the `--format=directory` parameter. In this format, the backup data is separated into multiple files representing the tables and large objects. Additionally, this format will later allow us to restore the backup with more granularity and even table by table.

With the next parameter, `--compress=none`, we decided not to compress the backup data to take advantage of `Kopia`. If we attempt to back up compressed data, the splitter algorithm will not function as expected. This is because data pre-compressed by `$ pg_dump` results in entirely different data, with chunks in a mixed order at the end of each backup. We do not want this; instead, we want data to be as close as possible to the original, following a recognizable pattern and order so that the splitter can create meaningful and efficient chunks.

The `--jobs=4` parameter is used to parallelize the backup process. This parameter is optional and defaults to `1`. But, if we are using a parallelizable backup format like `directory`, it is always a good idea to make the backup process faster by using more than a single job. Note that this approach will create more load on the database instance.

With the `--file` parameter, we specified where the backup data should be saved, and using the `--verbose` flag, we can see the progress of the backup process. It is usually a good idea to increase the log verbosity to see the progress, especially with long-running tasks.

After setting the parameters, we run the `$ pg_dump` command with a positional parameter, `postgres`, to specify the database name and the rest of the parameters. After running the command, we can expect to see the backup data in the `postgres0_backup` directory.

The next sub-step will be connecting to the `Kopia` repository and creating a new snapshot.

22.03.backup.ts

```
21 const repositoryConnectArgs = [
22   "--bucket=backup0",
23   `--access-key=${process.env.REPOSITORY_ACCESS_KEY}`,
24   `--secret-access-key=${process.env.REPOSITORY_SECRET_ACCESS_KEY}`,
25   "--endpoint=s3.fr-par.scw.cloud",
26   "--region=fr-par",
27   `--password=${process.env.REPOSITORY_PASSWORD}`,
28   "--override-username=root",
29   "--override-hostname=host0",
30 ];
31 await $`kopia repository connect s3 ${repositoryConnectArgs}`;
32
33 //
34 console.log("step - create snapshot");
35 const createSnapshotArgs = [
36   "./postgres0_backup/",
37   "--override-source=/backups/postgres0_backup/postgres",
38   "--parallel=4",
39 ];
40 await $`kopia snapshot create ${createSnapshotArgs}`;
```

Before doing anything with `$ kopia` CLI, we need to connect to a repository. In line 30, we connected to the repository we created in the first step. While doing that, we provided three important parameters: the `password` of the repository, the `username`, and the `hostname`. The `username` and `hostname` are extra important because they will be used as part of the snapshot keys.

After connecting to the repository, we created a new snapshot using the `$ kopia snapshot create` command. We provided the path to the backup as a positional argument and then overrode the source path to the backup directory. By default, `$ kopia` will use the absolute path of the backup source in the snapshot key. Yet, we override it with the `--override-source` to a more meaningful path. With the help of the `--parallel` parameter, we parallelize the compression and upload processes.

The last sub-step will be the cleanup process. In this step, we will expire the old snapshots using their key and then run the maintenance tasks to keep the repository clean and efficient by removing the old blobs and metadata. In this sub-step, we also had to set the CLI runner as the owner of the repository to run the maintenance tasks.

22.03.backup.ts

```
43  const expireSnapshotArgs = ["/backups/postgres0_backup/postgres/"];
44  await $`kopia snapshot expire ${expireSnapshotArgs}`;
45  await $`kopia maintenance set --owner=me`;
46  await $`kopia maintenance run --full`;
```

And that was our simple backup script. Although it is not a complete solution, it is a great starting point, and we can further improve this script

by:

- Tweaking parameters of different commands to make the process faster and more suitable for our environment, such as enabling paralellization for the database backup subprocess.
- Instead of using some hardcoded values, like the database or bucket name, we can further parameterize the script with environment variables and use the script for different databases and repositories.
- We can override the snapshot's start and end time to refer to the backup's actual start and end time for more accurate metadata.
- We can add a notification system to be informed when the backup is done. This might either be an `$ kopia notification` subcommand or a webhook to a service such as `Slack` or `Discord`.
- At the end of the backup process, we can remove the backup data from the disk so the next time the backup script is run on our local machine, the `$ pg_dump` will not throw a `file already exists` error.
- We can create another Job that downloads the snapshots and restores them (at least partially) to make sure the backup is working as expected and the data is intact.
- We can use the argv↗ or another library to pass the CLI parameters to the script to increase the flexibility. For example, we can enable table-based backup with a flag and increase the backup period for specific, more critical tables.

These are all great ideas for improving the script, and we can continue with them in the future. Yet, the key advantage is that the script creation phase is a natural waypoint in the development process which allows us to stop here (without setting up a CronJob) and manually trigger the

backup process. Furthermore, by running the script locally using the VPN connection to the cluster, we can develop and debug it more conveniently. Here is how we can run the script locally:

```
$ bun --env-file=22.04.backup.env run 22.03.backup.ts
```

22.03.06. STEP 6: PREPARE THE RESOURCES FOR THE CRONJOB

Now that we have created the scripts, let us pass the `.env` file and the scripts to the cluster. For the `backup.env` file, we will use a `Secret` object since it includes sensitive information, and for the scripts, we will use a `ConfigMap` object.

```
$ cat 22.04.backup.env
POSTGRES_PASSWORD=LT7jIZanGC
REPOSITORY_ACCESS_KEY=SCW41GJFZY4XXXXXXXXXX
...
$ kubectl create secret generic backup \
  -n db \
  --from-env-file=22.04.backup.env
secret/backup created
$ kubectl create configmap backup \
  -n db \
  --append-hash \
  --from-file=install.sh=22.02.install.sh \
  --from-file=backup.ts=22.03.backup.ts
configmap/backup-f2t9f89856 created
```

With the first command, we created a generic `Secret` object named `backup` to keep the sensitive information. While doing that, we used the `--from-env-file` parameter so that `$ kubectl` can read the environment variables from the file and create a key-value pair for each line.

For the scripts, we created a `ConfigMap` object named `backup` to keep the installation and backup scripts as string data. We also added the `--append-hash` parameter to append the content's hash to the ConfigMap's name so we can avoid potential collisions in future script revisions.

22.03.07. STEP 7: CREATE THE CRONJOB

Now that we have everything ready, let us create the CronJob to run the backup process. Since this `CronJob` definition is a little long, we will examine it in two different parts.

22.05.backup.yaml

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: backup
  namespace: db
spec:
  schedule: "0 */4 * * *"
  concurrencyPolicy: Forbid
  failedJobsHistoryLimit: 10
  jobTemplate:
    spec:
      backoffLimit: 1
      ...
```

We began defining the object with standard fields like `.apiVersion`, `.kind`, and `.metadata`. In the `.spec` section, we first defined the schedule to run the Job every 4 hours. By default, this `schedule` cron expression will use the default timezone of the control plane Node, but the `timezone` field can override it. With this configuration, the CronJob will generate a new **Job** at 4:00 AM, 8:00 AM, and every 4 hours thereafter.

After that, we defined the `concurrencyPolicy` to prevent concurrent backup runs and increased the `failedJobsHistoryLimit` from the default value of 1 to `10`. In this way, we can retain the history of failed Jobs longer for later inspection. At the start of the `jobTemplate` section, we defined the `backoffLimit` to `1` so that each Job will be retried at most once if it fails.

22.05.backup.yaml

```yaml
template:
  spec:
    restartPolicy: Never
    containers:
      - name: ubuntu
        image: ubuntu:24.04
        volumeMounts:
          - name: scripts
            mountPath: "/etc/scripts"
        envFrom:
          - secretRef:
              name: backup
        command: ["/bin/bash", "-c"]
        args:
          - |
            cd /tmp &&
            cp -v /etc/scripts/* . &&
            set -e &&
            source install.sh &&
            bun run backup.ts
    volumes:
      - name: scripts
        configMap:
          name: backup
```

In the Pod template for the CronJob, we set `restartPolicy` to `Never` to prevent container restarts, ensuring that failed Pods are not restarted and new Pods are created instead. The rest of the container configuration is relatively simple. We passed the environment variables from the `Secret` object to the container using the `envFrom` field. Then, we specified a volume named `scripts` generated from the `ConfigMap` object and mounted it to the `/etc/scripts` directory of the container.

Using the `args` field, we override the `CMD` instruction. First, we changed the current working directory to the `/tmp` directory because the directory that stores the scripts, `/etc/scripts`, is read-only, and this can become a problem when we try to run the scripts[4]. Then, we copied the scripts and sourced the installation script to install the dependencies.

The difference between running and sourcing the installation script is that running the script will create a new shell and the environment variables, but sourcing the script will use the current shell to run the commands, so we can use the variables if we set them outside the script, such as overriding the `PATH` variable. However, this approach has a problem. While sourcing the script, the errors will not be reported immediately and will be skipped if they are not handled in the script. To avoid this problem, we set the `set -e` command before sourcing the script so that the script will exit immediately if there is a non-zero exit code. Then, we run the backup script using the `bun run backup.ts` command.

Let us create the CronJob using the `$ kubectl apply` command.

```
$ kubectl apply -f 22.04.backup.yaml
cronjob.batch/backup created
$ kubectl get cronjob -n db
NAME     SCHEDULE      TIMEZONE   SUSPEND   ACTIVE   LAST SCHEDULE   A
backup   0 */4 * * *   <none>     False     0        <none>          1
```

22.03.08. STEP 8: RUN THE JOB AND TEST IT

Since the `CronJob` is only a `Job` generator, we do not need to wait until the next scheduled time to test it. We can create a new Job manually, and so the cluster will run it.

```
1  $ kubectl create job --from=cronjob/backup backup-now -n db
2  job.batch/backup-now created
3  $ kubectl get jobs -n db
4  NAME         STATUS     COMPLETIONS   DURATION   AGE
5  backup-now   Complete   1/1           76s        2m12s
6  $ kubectl get pods -n db
7  NAME                          READY   STATUS      RESTARTS   AGE
8  backup-now-9n45d              0/1     Completed   0          5m29s
9  $ kubectl logs backup-now-9n45d -n db
10 ...
11 Cleaned up 0 logs.
12 Finished full maintenance.
13 step - cleanup
```

Using the first command, we created a new Job named `backup-now` from the template of the `CronJob`. After creating the Job, we waited briefly and then checked the status of the Job and the Pods. Since the Job was completed, we listed the Pods in line 6 and then checked the logs as the last instruction.

As one last step, let us use the `$ kopia` CLI to check if the snapshots are created.

```
$ kopia repository list
root@host0:/backups/postgres0_backup/postgres
 2025-03-29 20:29:58 -07 k0278682c5af2afe5fbba6a6540079b87 96 MB drwx-
```

As we can see from the output of the `list` command, a `96 MB` snapshot is created, which includes 6 files and 1 directory. The snapshot is saved with a key that contains the username and the hostname that we overrode in the `repository connect` command. As a best practice, we should download the backup data and restore it to another database to verify that it works as expected.

In this chapter, we explored the `Job` and `CronJob` resources in Kubernetes. Then, we created a simple backup script using TypeScript and Bun to back up the PostgreSQL database we created in the previous chapter. After that, we stored the backup data in a `Kopia` repository. In the next chapter, we will learn about advanced deployment strategies and see how we can create a `GitHub` action to automate the deployment of the `chlabs.io` application.

FOOTNOTES

1. Please refer to Docker documentation↗ for more information and workarounds if you need to use the `shell` form. ↵

2. Even though we used the shell form of the `CMD` instruction, it will not be a problem in this case because PID 1 is a short-running `bash` shell command. ↵

3. In many environments, the shell starts with sourcing a base file like `~/.bashrc` so that we have the environment variables and other configurations. Yet, this base file is usually configured to work in an interactive shell that is connected to a terminal and tends to return early for non-interactive shells such as scripts by checking the primary prompt string variable, such as `[ -z "$PS1" ] && return` at the start. Since most of the configurations are added to the end of the initialization script file, they will not be applied to the non-interactive shells. ↵

4. Sometimes, package managers may require a writable directory to install the packages. Furthermore, if `$ bun` needs to auto-install the packages, it will create a `node_modules` directory in the current working directory, which can't be created if the directory is read-only. ↵