Considerations for Multi-Container Applications

Objectives

After completing this section, students should be able to:

- Describe considerations for containerizing applications with multiple container images.
- Leverage networking concepts in containers.
- Create a multi-container application with Podman.
- Describe the architecture of the To Do List application.

Leveraging Multi-Container Applications

The examples shown so far throughout this course have worked fine with a single container. A more complex application, however, can get the benefits of deploying different components into different containers. Consider an application composed of a front-end web application, a REST back end, and a database server. Those components may have different dependencies, requirements and life cycles.

Although it is possible to orchestrate multi-container applications' containers manually, Kubernetes and OpenShift provide tools to facilitate orchestration. Attempting to manually manage dozens or hundreds of containers quickly becomes complicated. In this section, we are going to return to using Podman to create a simple multi-container application to demonstrate the underlying manual steps for container orchestration. In later sections, you will use Kubernetes and OpenShift to orchestrate these same application containers.

Discovering Services in a Multi-Container Application

Podman uses *Container Network Interface (CNI)* to create a *software-defined network (SDN)* between all containers in the host. Unless stated otherwise, CNI assigns a new IP address to a container when it starts.

Each container exposes all ports to other containers in the same SDN. As such, services are readily accessible within the same network. The containers expose ports to external networks only by explicit configuration.

Due to the dynamic nature of container IP addresses, applications cannot rely on either fixed IP addresses or fixed DNS host names to communicate with middleware services and other application services. Containers with dynamic IP addresses can become a problem when working with multi-container applications because each container must be able to communicate with others to use services upon which it depends.

For example, consider an application composed of a front-end container, a back-end container, and a database. The front-end container needs to retrieve the IP address of the back-end container. Similarly, the back-end container needs to retrieve the IP address of the database container. Additionally, the IP address could change if a container restarts, so a process is needed to ensure any change in IP triggers an update to existing containers.

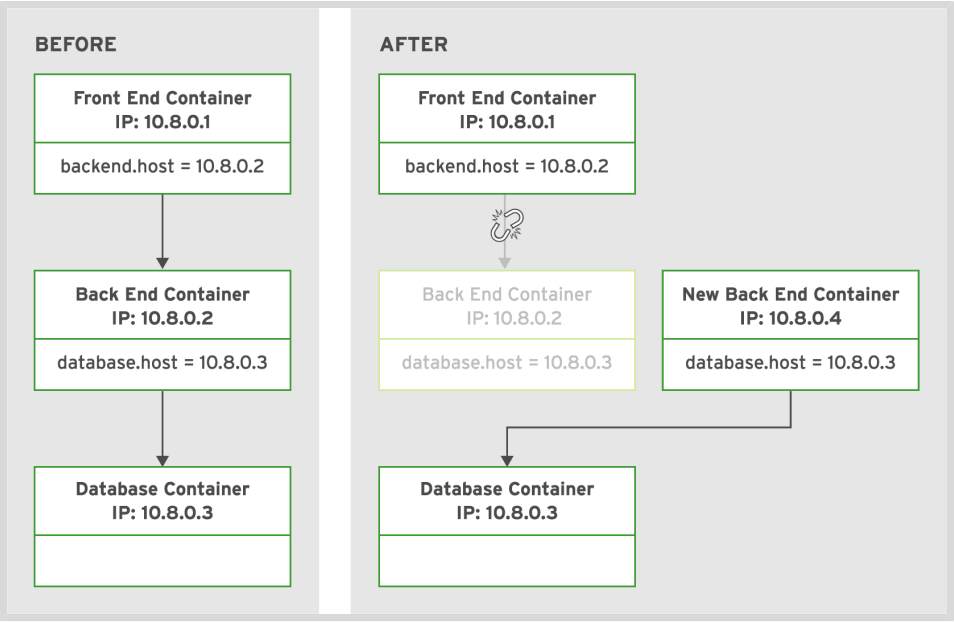

Figure 7.1: A restart breaks three-tiered application links

Both Kubernetes and OpenShift provide potential solutions to the issue of service discoverability and the dynamic nature of container networking. Some of these solutions are covered later in the chapter.

Comparing Podman and Kubernetes

Using environment variables allows you to share information between containers with Podman. However, there are still some limitations and some manual work involved in ensuring that all environment variables stay in sync, especially when working with many containers. Kubernetes provides an approach to solve this problem by creating services for your containers, as covered in previous chapters.

Pods are attached to a Kubernetes namespace, which OpenShift calls a *project*. When a pod starts, Kubernetes automatically adds a set of environment variables for each service defined on the same namespace.

Any service defined on Kubernetes generates environment variables for the IP address and port number where the service is available. Kubernetes automatically injects these environment variables into the containers from pods in the same namespace. These environment variables usually follow a convention:

- *Uppercase*: All environment variables are set using uppercase names.
- *Snakecase*: Any environment variable created by a service is usually composed of multiple words separated with an underscore (_).
- *Service name first*: The first word for an environment variable created by a service is the service name.
- *Protocol type*: Most network environment variables include the protocol type (TCP or UDP).

These are the environment variables generated by Kubernetes for a service:

- **`<SERVICE_NAME>_SERVICE_HOST`**: Represents the IP address enabled by a service to access a pod.

- **<SERVICE_NAME>_SERVICE_PORT**: Represents the port where the server port is listed.
- **<SERVICE_NAME>_PORT**: Represents the address, port, and protocol provided by the service for external access.
- **<SERVICE_NAME>_PORT_<PORT_NUMBER>_<PROTOCOL>**: Defines an alias for the **<SERVICE_NAME>_PORT**.
- **<SERVICE_NAME>_PORT_<PORT_NUMBER>_<PROTOCOL>_PROTO**: Identifies the protocol type (TCP or UDP).
- **<SERVICE_NAME>_PORT_<PORT_NUMBER>_<PROTOCOL>_PORT**: Defines an alias for **<SERVICE_NAME>_SERVICE_PORT**.
- **<SERVICE_NAME>_PORT_<PORT_NUMBER>_<PROTOCOL>_ADDR**: Defines an alias for **<SERVICE_NAME>_SERVICE_HOST**.

For instance, given the following service:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: mysql
  name: mysql
spec:
  ports:
    - protocol: TCP
    - port: 3306
  selector:
    name: mysql
```

The following environment variables are available for each pod created after the service, on the same namespace:

```
MYSQL_SERVICE_HOST=10.0.0.11
MYSQL_SERVICE_PORT=3306
MYSQL_PORT=tcp://10.0.0.11:3306
MYSQL_PORT_3306_TCP=tcp://10.0.0.11:3306
MYSQL_PORT_3306_TCP_PROTO=tcp
MYSQL_PORT_3306_TCP_PORT=3306
MYSQL_PORT_3306_TCP_ADDR=10.0.0.11
```

Note

Other relevant **<SERVICE_NAME>_PORT_*** environment variable names are set on the basis of the protocol. IP address and port number are set in the **<SERVICE_NAME>_PORT** environment variable. For example, **MYSQL_PORT=tcp://10.0.0.11:3306** entry leads to the creation of environment variables with names such as **MYSQL_PORT_3306_TCP**, **MYSQL_PORT_3306_TCP_PROTO**, **MYSQL_PORT_3306_TCP_PORT**, and **MYSQL_PORT_3306_TCP_ADDR**. If the protocol component of an environment variable is undefined, Kubernetes uses the TCP protocol and assigns the variable names accordingly.

Describing the To Do List Application

Many labs from this course make use of a To Do List application. This application is divided into three tiers, as illustrated by the following figure:

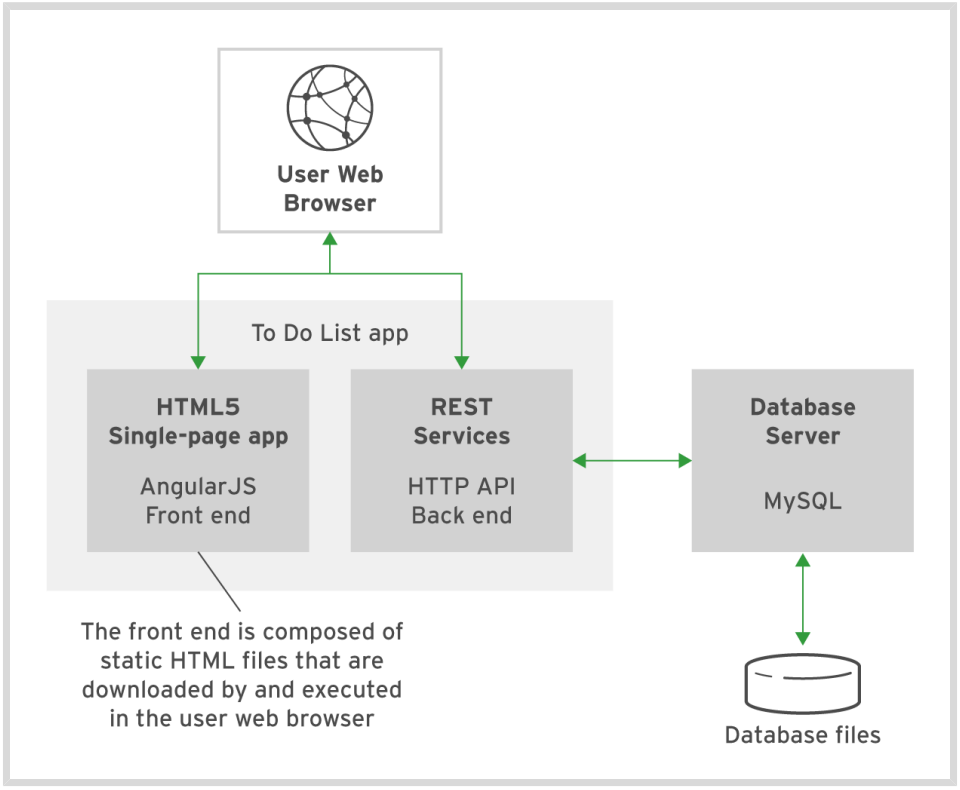

Figure 7.2: To Do List application logical architecture

- The presentation tier is built as a single-page HTML5 front-end using AngularJS.
- The business tier is composed by an HTTP API back-end, with Node.js.
- The persistence tier based on a MySQL database server.

The following figure is a screen capture of the application web interface:

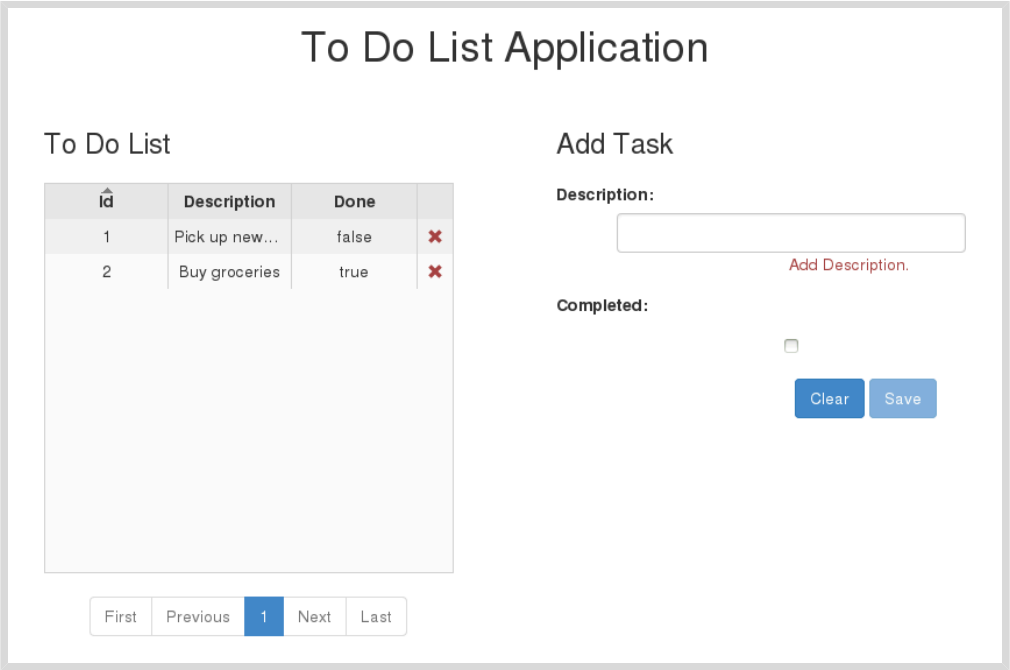

Figure 7.3: The To Do List application

On the left is a table with items to complete, and on the right is a form to add a new item.

The classroom private registry server, **`services.lab.example.com`**, provides the application in two versions:

nodejs
: Represents the way a typical developer would create the application as a single unit, without caring to break it into tiers or services.

nodejs_api
: Shows the changes needed to break the application into presentation and business tiers. Each tier corresponds to an isolated container image.

The sources of both of these application versions are available from the **`todoapp/nodejs`** folder in the Git repository at: `https://github.com/RedHatTraining/DO180-apps.git`.