Yipee.io - Simplifying Kubernetes Management

The Problem

Over the last few years, Kubernetes has become the de facto standard for orchestrating container-based applications. There are several reasons for its success, including:

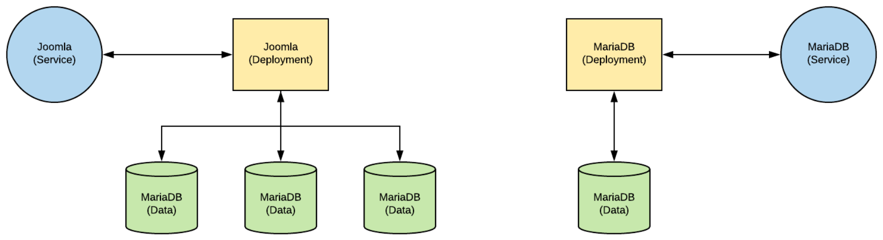
- Battle Tested - Kubernetes is the outcome of a long-term Google effort to manage their own vast hardware and software environments. Over the years, Google engineers were able to figure out what works well and what doesn't.
- Open Source - Being completely open means the pool of potential contributors is enormous. This has helped accelerate the pace of fixes and improvements.
- Extensible - There is a well-designed versioning scheme which supports experimentation and predefined extension points for users to write their own custom resources.

One valuable feature missing from the list, however, is simplicity. Kubernetes does a *lot* and very little of the resultant complexity is hidden from users. For applications built to run in both Kubernetes and Docker Swarm, the Kubernetes descriptions are generally much bigger and more complex, and there is less overt structure since the connections between different components are implied by common naming rather than being explicit parts of the formal definition. This means that people reading Kubernetes descriptions have to fill in much of the structure in their heads.

It doesn't have to be this way, however. The structure of a Kubernetes application is only implicit because of the way it is mapped onto YAML files. All the information necessary to present an application's structure explicitly is available; it's just buried in a lot of extraneous detail. It's well understood that different sorts of representations bring out different aspects of a system so that it can be useful to adopt different representations for different purposes. Unfortunately, the set of available representations for Kubernetes configurations is impoverished. The basic Kubernetes infrastructure only provides two: YAML/JSON manifests and a textual dashboard. Neither of them provides any direct depiction of application structure and dependencies. There are some third-party tools like WeaveScope that can show graphs of runtime components and dependencies but that is significantly more useful for monitoring and troubleshooting than it is for developing big-picture understanding because much of the runtime state is extraneous to the big picture. If you're Netflix and you have 5000 pods associated with a particular Kubernetes deployment, how will the 5000 objects on your screen help you understand your system?

Yipee

At Yipee.io, we believe there is a sweet spot for managing Kubernetes applications that exposes the most useful information in a way that is easy to understand and digest. Consider the “new engineer” scenario. If you bring a new engineer onto your project, what is the first thing you do to begin to spin her up? For me, the first thing is to go right to the whiteboard. Imagine a simple Joomla application. If I were explaining it, I’d draw all the main controllers, services, volumes, and their connections because that’s what matters conceptually. Yes, there is other important information but it’s second order and doesn’t really have much to do with obtaining a high-level understanding. I’d draw something like this:



Contrast that with the YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: joomla
spec:
  selector:
    name: joomla-kubernetes
    component: joomla
  ports:
  - port: 80
    targetPort: 80
    name: joomla-80
    protocol: TCP
  - port: 443
    targetPort: 443
    name: joomla-443
    protocol: TCP
  type: ClusterIP

---
apiVersion: v1
kind: Service
```

```
metadata:
  name: mariadb
  selector:
    name: joomla-kubernetes
    component: mariadb
  ports:
  - port: 3306
    targetPort: 3306
    name: mariadb-3306
    protocol: TCP
  type: ClusterIP

---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: joomla
spec:
  selector:
    matchLabels:
      name: joomla-kubernetes
      component: joomla
  template:
    spec:
      imagePullSecrets: []
      containers:
      - volumeMounts:
        - mountPath: /bitnami/joomla
          name: joomla-data
        - mountPath: /bitnami/php
          name: php-data
        - mountPath: /bitnami/apache
          name: apache-data
        name: joomla
        env:
        - name: JOOMLA_EMAIL
          value: user@example.com
        - name: JOOMLA_PASSWORD
          valueFrom:
            configMapKeyRef:
              key: JOOM_PASSWORD
              name: joomlaconfig
        - name: JOOMLA_USERNAME
          value: root
        - name: MARIADB_HOST
          value: mariadb2
        - name: MARIADB_PASSWORD
          valueFrom:
            configMapKeyRef:
              key: MARIA_PASSWORD
              name: joomlaconfig
        - name: MARIADB_PORT
          value: '4306'
        ports:
        - containerPort: 80
          protocol: TCP
        - containerPort: 1776
          protocol: TCP
        - containerPort: 443
          protocol: TCP
        image: bitnami/joomla:latest
      volumes:
      - name: php-data
        persistentVolumeClaim:
          claimName: php-data-claim
      - name: apache-data
        persistentVolumeClaim:
          claimName: apache-data-claim
      - name: joomla-data
        persistentVolumeClaim:
          claimName: joomla-data-claim
      restartPolicy: Always
    metadata:
      labels:
        name: joomla-kubernetes
        component: joomla
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  replicas: 1
  revisionHistoryLimit: 2

---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: mariadb
spec:
  selector:
    matchLabels:
      name: joomla-kubernetes
      component: mariadb
  template:
    spec:
      imagePullSecrets: []
      containers:
      - volumeMounts:
        - mountPath: /bitnami/mariadb
          name: mariadb-data
        name: mariadb
        env:
        - name: ALLOW_EMPTY_PASSWORD
          value: 'yes'
        - name: MARIADB_PORT
          value: '3306'
        - name: MARIADB_ROOT_PASSWORD
          valueFrom:
            configMapKeyRef:
              key: MARIA_PASS
              name: joomlaconfig
        ports:
        - containerPort: 3306
          protocol: TCP
        image: bitnami/mariadb:10.1.26-r2
      volumes:
      - name: mariadb-data
        persistentVolumeClaim:
          claimName: mariadb-data-claim
      restartPolicy: Always
    metadata:
      labels:
        name: joomla-kubernetes
        component: mariadb
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  replicas: 1
  revisionHistoryLimit: 2

---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: apache-data-claim
```

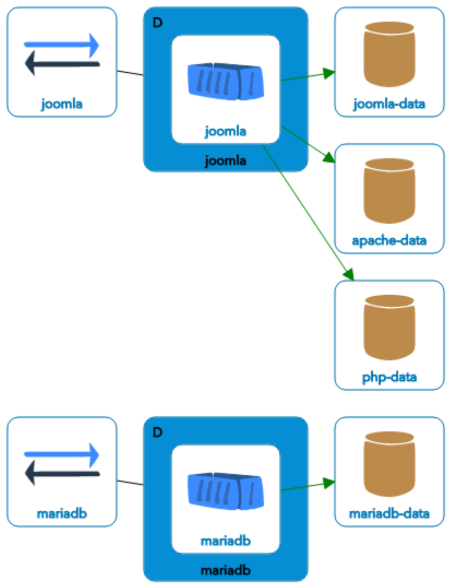
```yaml
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 2G
  volumeMode: Filesystem

---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: php-data-claim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 2G
  volumeMode: Filesystem

---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: joomla-data-claim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 2G
  volumeMode: Filesystem

---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mariadb-data-claim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 2G
  volumeMode: Filesystem
```

I think it's pretty obvious which is easier to understand. Here is what the application would look like in Yipee.io:



All of the detail provided in the YAML is available to view and modify in Yipee but we leverage a high-level view wherever possible. Many actions taken on configurations don't require exposing the level of detail present in YAML files and are more understandable and accessible at a higher level of abstraction.

Features

The current SaaS-based Yipee.io provides graphical import and editing of Kubernetes and Docker Compose models as well as translation of models from Compose to Kubernetes. Completed models can be downloaded as Kubernetes manifests or auto-generated Helm charts. Models are validated according to standard Kubernetes schemata.

Work is now going on to address our most common user requests in three areas:

- Open source editing
- On-premise deployment
- Live cluster status and updates

We're creating two new Yipee variants to allow customers to edit models within their local networks and to support live status and pushbutton deployment. (The latter variant will include the former). With the new versions in place, Yipee.io will support the full lifecycle of Kubernetes applications. It will be possible to create configurations from scratch or import them from:

- Kubernetes YAML manifests
- Docker Compose files
- Live Kubernetes cluster namespaces

Once a configuration is in Yipee.io, you will be able to:

- Graphically edit it
- Apply it to a cluster
- Compare it against other configurations with a graphical "diff" facility
- Fork it into a new configuration that can be automatically updated when the parent configuration changes
- Automatically turn it into a Helm chart; multiple related configurations will produce a single parameterized Helm chart with distinct value files
- Add simple logging, monitoring, or service mesh configuration
- Integrate with your SCM system to facilitate CI/CD

Additionally, live runtime status information will be displayed for any configuration that has been applied to a cluster.

Intended Users

Yipee.io is built to assist architects, developers, and operations teams that work with Kubernetes. Anyone who deals with complex Kubernetes applications can benefit from interacting with them at a high level via Yipee:

- Architects can use Yipee to construct application models and generate diagrams that can be shared with their teams. Yipee will ensure that everything is syntactically correct and the models are up-to-date with current Kubernetes and the diagrams will never get stale.

- Developers can use Yipee models as documentation or make derived models for use in testing which track changes to their upstream counterparts.

- Operations can use Yipee's model inheritance and graphical diffing to maintain multiple application configurations for development, testing, staging, and production.

Technology

The current Yipee.io is a container-based application that can run either in Docker Swarm or Kubernetes. When embedded in a Kubernetes cluster, live status and updates become available. Yipee is composed of four main components:

- UI - TypeScript (Angular JS)
- API Server - JavaScript (Node.js)
- Kubernetes GraphQL Server - Go
- Converters - Clojure + Forward Chaining Rules

The UI and API Server are more-or-less standard but the other two bullets need some explanation. We developed a GraphQL server for Kubernetes because we found the Kubernetes ReST APIs a bit clumsy for dealing with complex applications due to the siloed view of ReST. Most applications have combinations of different component types which means a lot of round trips when using the standard APIs. Having used GraphQL in the past for an authorization server, we knew it worked well for our UI and was easy to extend without requiring client changes, so we built a GraphQL API for Kubernetes (See kubeiql.io for more information). The "converters" translate to and from UI format and from Docker Compose to Kubernetes. Two aspects of Kubernetes led us to embed a rule engine into our conversion process:

1 Kubernetes changes very rapidly 2 There are many global constraints on configurations

The rapid pace of change makes a rule-based approach attractive because it's possible to quickly adapt to a change by tailoring a special-case rule. As things settle out and stabilize, the special cases can be folded into the mainline code.

Global constraints such as matching labels across services and controllers are difficult to write in straight line code because any change can potentially cause a constraint violation so you need to check every possible combination on a modification. This is both inefficient and difficult to code as it often involves a large amount of fragile navigation code that requires frequent updating (see aspect 1). Using rules clarifies the logic by hiding the bookkeeping. For example, the following rule ensures that all object fields have correct types:

```
(defrule check-types
  "Ensure all fields are correct types"
  [?wme :wme (type-error ?wme)]
  [:not [?verr :validation-error (invalid-type ?verr (wme-path ?wme))]]
  =>
  (doseq [invalid (bad-fields ?wme)]
    (generate-type-error ?wme invalid)))
```

Any change to an object will automatically route it back through this check.

Future Directions

The long-term goal of Yipee.io is to streamline and bulletproof design, deployment, and management of Kubernetes applications. Once we complete our standalone editor and cluster dashboard, we plan to offer a variety of high-level facilities to application modeling such as automatic addition of Prometheus monitoring or an API Gateway. Our rule-based approach makes it easy to apply interesting transformations to existing models opening up a wide array of possible enhancements.

The Team

Yipee.io boasts an extremely experienced team with several thirty-plus year software veterans. Over half of the team were previously core developers of other cluster management and orchestration systems including Cassatt's Active Response and CA Technologies' AppLogic. The two pedigrees come together in Yipee as Active Response was a dynamic orchestration environment for physical hardware and AppLogic provided a graphical construction kit for applications built from virtual machines. Yipee.io is the offspring of these ideas.