Getting Started

This guide will walk you through the following steps:

- Installing the CDK8s CLI.
- Creating a new CDK8s project in one of the supported programming languages.
- Define & deploy your first CDK8s application.
- Define a custom CDK8s construct.

Install the CLI

CDK8s has a cute little CLI that has a few useful commands. Let's start by installing the CDK8s CLI globally. We have two options for this.

npm

```
npm install -g cdk8s-cli
```

yarn

```
yarn global add cdk8s-cli
```

Prerequisites

TypeScript

- Node.js >= 12.13.0
- Your favorite editor/IDE
- yarn (optional)

Python

- Python >= 3.7.7
- pipenv version 2018.11.26 or above.
- Your favorite editor/IDE

Java

- Maven >= 3.6.3
- Your favorite editor/IDE

Go

- Go >= 1.16
- Your favorite editor/IDE

> **Info**
>
> This Getting Started guide will help you create a Kubernetes manifest using your preferred programming language. **You do not need access to a Kubernetes cluster in order to produce a manifests using CDK8s.**

New Project

Now, we'll use the `cdk8s init` command to create a new CDK8s app:

TypeScript

```
$ mkdir hello
$ cd hello
$ cdk8s init typescript-app
Initializing a project from the typescript-app template
...
```

Python

```
$ mkdir hello
$ cd hello
$ cdk8s init python-app
Initializing a project from the python-app template
...
```

Java

```
$ mkdir hello
$ cd hello
$ cdk8s init java-app
Initializing a project from the java-app template
...
```

Go

```
$ mkdir hello
$ cd hello
$ cdk8s init go-app
Initializing a project from the go-app template
```

This will perform the following:

1. Create a new project directory
2. Install CDK8s as a dependency
3. Import all Kubernetes API objects

Apps & Charts

Apps are structured as a tree of **constructs**, which are composable units of abstraction. We will learn more about constructs soon.

This initial code created by `cdk8s init` defines an app with a single, empty, chart.

When you synthesize the app, a Kubernetes manifest YAML will be produced for each `Chart` in your app and will write it to the `dist` directory. At this point, the YAML file should be empty since we haven't defined any resources yet.

Let have a look at the code:

TypeScript

`main.ts`

```
import { Construct } from 'constructs';
import { App, Chart, ChartProps } from 'cdk8s';

export class MyChart extends Chart {
  constructor(scope: Construct, id: string, props: ChartProps = { }) {
    super(scope, id, props);

    // define resources here

  }
}

const app = new App();
new MyChart(app, 'hello');
app.synth();
```

To produce and inspect the generated manifest, you can run:

```
npm run compile && cdk8s synth
dist/hello.k8s.yaml

cat dist/hello.k8s.yaml
<EMPTY>
```

> Note that since TypeScript is a compiled language, we will need to compile `.ts` files to `.js` in order to execute our CDK8s app. To avoid explicitly compiling every time, you can run a watch process in the background by running: `npm run watch`

Python

`main.py`

```python
#!/usr/bin/env python
from constructs import Construct
from cdk8s import App, Chart


class MyChart(Chart):
    def __init__(self, scope: Construct, id: str):
        super().__init__(scope, id)

        # define resources here


app = App()
MyChart(app, "hello")

app.synth()
```

To produce and inspect the generated manifest, you can run:

```
cdk8s synth
dist/hello.k8s.yaml

cat dist/hello.k8s.yaml
<EMPTY>
```

Java

`src/main/java/com/mycompany/app/Main.java`

```java
package com.mycompany.app;

import software.constructs.Construct;

import org.cdk8s.App;
import org.cdk8s.Chart;
```

```
import org.cdk8s.ChartProps;

public class Main extends Chart
{

    public Main(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public Main(final Construct scope, final String id, final ChartProps
options) {
        super(scope, id, options);

        // define resources here
    }

    public static void main(String[] args) {
        final App app = new App();
        new Main(app, "hello");
        app.synth();
    }
}
```

To produce and inspect the generated manifest, you can run:

```
cdk8s synth
dist/hello.k8s.yaml

cat dist/hello.k8s.yaml
<EMPTY>
```

Go

`main.go`

```
package main

import (
  "github.com/aws/constructs-go/constructs/v3"
  "github.com/aws/jsii-runtime-go"
  "github.com/cdk8s-team/cdk8s-core-go/cdk8s"
)

type MyChartProps struct {
  cdk8s.ChartProps
}

func NewMyChart(scope constructs.Construct, id string, props *MyChartProps)
cdk8s.Chart {
  var cprops cdk8s.ChartProps
  if props != nil {
    cprops = props.ChartProps
  }
  chart := cdk8s.NewChart(scope, jsii.String(id), &cprops)
```

```
  // define resources here

  return chart
}

func main() {
  app := cdk8s.NewApp(nil)
  NewMyChart(app, "hello", nil)
  app.Synth()
}
```

To produce and inspect the generated manifest, you can run:

```
cdk8s synth
dist/hello.k8s.yaml

cat dist/hello.k8s.yaml
<EMPTY>
```

Importing Constructs for the Kubernetes API

OK, now let's define some Kubernetes API objects inside our chart.

Similarly to **charts** and **apps**, Kubernetes API Objects are also represented in CDK8s as **constructs**. These constructs are *imported* to your project using the `cdk8s import` command which will add source files to your project that include constructs that represent the Kubernetes API.

> **Info**
>
> When `cdk8s init` created your project it already executed `cdk8s import` for you, so you should see an imports directory already there. You can either commit this directory to source-control or generate it as part of your build process.

Now, let's use these constructs to define a simple Kubernetes application which contains Service and a Deployment resources inspired by paulbouwer's hello-kubernetes project.

TypeScript

```
import { Construct } from 'constructs';
import { App, Chart, ChartProps } from 'cdk8s';

// imported constructs
import { KubeDeployment, KubeService, IntOrString } from './imports/k8s';
```

```
export class MyChart extends Chart {
  constructor(scope: Construct, id: string, props: ChartProps = { }) {
    super(scope, id, props);

    const label = { app: 'hello-k8s' };

    // notice that there is no assigment neccesary when creating resources.
    // simply instantiating the resource is enough because it adds it to the
construct tree via
    // the first argument, which is always the parent construct.
    // its a little confusing at first glance, but this is an inherent aspect
of the constructs
    // programming model, and you will encounter it many times.
    // you can still perform an assignment of course, if you need to access
    // atrtibutes of the resource in other parts of the code.

    new KubeService(this, 'service', {
      spec: {
        type: 'LoadBalancer',
        ports: [ { port: 80, targetPort: IntOrString.fromNumber(8080) } ],
        selector: label
      }
    });

    new KubeDeployment(this, 'deployment', {
      spec: {
        replicas: 2,
        selector: {
          matchLabels: label
        },
        template: {
          metadata: { labels: label },
          spec: {
            containers: [
              {
                name: 'hello-kubernetes',
                image: 'paulbouwer/hello-kubernetes:1.7',
                ports: [ { containerPort: 8080 } ]
              }
            ]
          }
        }
      }
    });
  }
}

const app = new App();
new MyChart(app, 'hello');
app.synth();
```

Now, to synthesize the app, run:

```
npm run compile && cdk8s synth
```

Python

```python
#!/usr/bin/env python
from constructs import Construct
from cdk8s import App, Chart

from imports import k8s

class MyChart(Chart):
    def __init__(self, scope: Construct, id: str):
        super().__init__(scope, id)

        # define resources here
        label = {"app": "hello-k8s"}

        # notice that there is no assigment neccesary when creating resources.
        # simply instantiating the resource is enough because it adds it to
the construct tree via
        # the first argument, which is always the parent construct.
        # its a little confusing at first glance, but this is an inherent
aspect of the constructs
        # programming model, and you will encounter it many times.
        # you can still perform an assignment of course, if you need to access
        # atrtibutes of the resource in other parts of the code.

        k8s.KubeService(self, 'service',
                    spec=k8s.ServiceSpec(
                    type='LoadBalancer',
                    ports=[k8s.ServicePort(port=80,
target_port=k8s.IntOrString.from_number(8080))],
                    selector=label))

        k8s.KubeDeployment(self, 'deployment',
                    spec=k8s.DeploymentSpec(
                        replicas=2,
                        selector=k8s.LabelSelector(match_labels=label),
                        template=k8s.PodTemplateSpec(
                        metadata=k8s.ObjectMeta(labels=label),
                        spec=k8s.PodSpec(containers=[
                            k8s.Container(
                            name='hello-kubernetes',
                            image='paulbouwer/hello-kubernetes:1.7',
                            ports=
[k8s.ContainerPort(container_port=8080)])])))))


app = App()
MyChart(app, "hello")

app.synth()
```

Now, to synthesize the app, run:

```
cdk8s synth
```

Java

```
package com.mycompany.app;

import software.constructs.Construct;

import java.util.ArrayList;
import java.util.List;
import java.util.HashMap;
import java.util.Map;

import org.cdk8s.App;
import org.cdk8s.Chart;
import org.cdk8s.ChartProps;

import imports.k8s.IntOrString;
import imports.k8s.LabelSelector;
import imports.k8s.ObjectMeta;
import imports.k8s.PodTemplateSpec;
import imports.k8s.KubeService;
import imports.k8s.KubeServiceProps;
import imports.k8s.ServicePort;
import imports.k8s.ServiceSpec;
import imports.k8s.DeploymentSpec;
import imports.k8s.PodSpec;
import imports.k8s.Container;
import imports.k8s.ContainerPort;
import imports.k8s.KubeDeployment;
import imports.k8s.KubeDeploymentProps;

public class Main extends Chart
{

    public Main(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public Main(final Construct scope, final String id, final ChartProps
options) {
        super(scope, id, options);

        // Defining a LoadBalancer Service
        final String serviceType = "LoadBalancer";
        final Map<String, String> selector = new HashMap<>();
        selector.put("app", "hello-k8s");
        final List<ServicePort> servicePorts = new ArrayList<>();
        final ServicePort servicePort = new ServicePort.Builder()
            .port(80)
            .targetPort(IntOrString.fromNumber(8080))
            .build();
        servicePorts.add(servicePort);
        final ServiceSpec serviceSpec = new ServiceSpec.Builder()

             type(serviceType)
```

```
            .type(serviceType)
            .selector(selector)
            .ports(servicePorts)
            .build();
        final KubeServiceProps serviceProps = new KubeServiceProps.Builder()
            .spec(serviceSpec)
            .build();

        // notice that there is no assigment neccesary when creating
resources.
        // simply instantiating the resource is enough because it adds it to
the construct tree via
        // the first argument, which is always the parent construct.
        // its a little confusing at first glance, but this is an inherent
aspect of the constructs
        // programming model, and you will encounter it many times.
        // you can still perform an assignment of course, if you need to
access
        // atrtibutes of the resource in other parts of the code.

        new KubeService(this, "service", serviceProps);

        // Defining a Deployment
        final LabelSelector labelSelector = new
LabelSelector.Builder().matchLabels(selector).build();
        final ObjectMeta objectMeta = new
ObjectMeta.Builder().labels(selector).build();
        final List<ContainerPort> containerPorts = new ArrayList<>();
        final ContainerPort containerPort = new ContainerPort.Builder()
            .containerPort(8080)
            .build();
        containerPorts.add(containerPort);
        final List<Container> containers = new ArrayList<>();
        final Container container = new Container.Builder()
            .name("hello-kubernetes")
            .image("paulbouwer/hello-kubernetes:1.7")
            .ports(containerPorts)
            .build();
        containers.add(container);
        final PodSpec podSpec = new PodSpec.Builder()
            .containers(containers)
            .build();
        final PodTemplateSpec podTemplateSpec = new PodTemplateSpec.Builder()
            .metadata(objectMeta)
            .spec(podSpec)
            .build();
        final DeploymentSpec deploymentSpec = new DeploymentSpec.Builder()
            .replicas(1)
            .selector(labelSelector)
            .template(podTemplateSpec)
            .build();
        final KubeDeploymentProps deploymentProps = new
KubeDeploymentProps.Builder()
            .spec(deploymentSpec)
            .build();

        new KubeDeployment(this, "deployment", deploymentProps);
```

```
    }

    public static void main(String[] args) {
        final App app = new App();
        new Main(app, "hello");
        app.synth();
    }
}
```

Now, to synthesize the app, run:

```
cdk8s synth
```

Go

```
package main

import (
  "example.com/hello-k8s/imports/k8s"
  "github.com/aws/constructs-go/constructs/v3"
  "github.com/aws/jsii-runtime-go"
  "github.com/cdk8s-team/cdk8s-core-go/cdk8s"
)

type MyChartProps struct {
  cdk8s.ChartProps
}

func NewMyChart(scope constructs.Construct, id string, props *MyChartProps)
cdk8s.Chart {
  var cprops cdk8s.ChartProps
  if props != nil {
    cprops = props.ChartProps
  }
  chart := cdk8s.NewChart(scope, jsii.String(id), &cprops)

  label := map[string]*string{"app": jsii.String("hello-k8s")}

  k8s.NewKubeService(chart, jsii.String("service"), &k8s.KubeServiceProps{
    Spec: &k8s.ServiceSpec{
      Type: jsii.String("LoadBalancer"),
      Ports: &[]*k8s.ServicePort{{
        Port:       jsii.Number(80),
        TargetPort: k8s.IntOrString_FromNumber(jsii.Number(8000)),
      }},
      Selector: &label,
    },
  })

  // notice that there is no assigment neccesary when creating resources.
  // simply instantiating the resource is enough because it adds it to the
construct tree via
  // the first argument, which is always the parent construct.
```

```
  // its a little confusing at first glance, but this is an inherent aspect of
the constructs
  // programming model, and you will encounter it many times.
  // you can still perform an assignment of course, if you need to access
  // atrtibutes of the resource in other parts of the code.

  k8s.NewKubeDeployment(chart, jsii.String("deployment"),
&k8s.KubeDeploymentProps{
    Spec: &k8s.DeploymentSpec{
      Replicas: jsii.Number(2),
      Selector: &k8s.LabelSelector{
        MatchLabels: &label,
      },
      Template: &k8s.PodTemplateSpec{
        Metadata: &k8s.ObjectMeta{
          Labels: &label,
        },
        Spec: &k8s.PodSpec{
          Containers: &[]*k8s.Container{{
            Name:  jsii.String("hello-kubernetes"),
            Image: jsii.String("paulbouwer/hello-kubernetes:1.7"),
            Ports: &[]*k8s.ContainerPort{{ContainerPort: jsii.Number(8080)}},
          }},
        },
      },
    },
  })

  return chart
}

func main() {
  app := cdk8s.NewApp(nil)
  NewMyChart(app, "hello", nil)
  app.Synth()
}
```

Now, to synthesize the app, run:

```
cdk8s synth
```

This will be contents of `hello.k8s.yaml`:

```
apiVersion: "v1"
kind: "Service"
metadata:
  name: "hello-service-c8c17160"
spec:
  ports:
    - port: 80
      targetPort: 8080
  selector:
    app: "hello-k8s"
  type: "LoadBalancer"
```

```yaml
---
apiVersion: "apps/v1"
kind: "Deployment"
metadata:
  name: "hello-deployment-c8c7fda7"
spec:
  replicas: 2
  selector:
    matchLabels:
      app: "hello-k8s"
  template:
    metadata:
      labels:
        app: "hello-k8s"
    spec:
      containers:
        - image: "paulbouwer/hello-kubernetes:1.7"
          name: "hello-kubernetes"
          ports:
            - containerPort: 8080
```

The manifest synthesized by your app is ready to be applied to any Kubernetes cluster using standard tools like `kubectl apply`:

```
kubectl apply -f dist/hello.k8s.yaml
```

Abstraction through Constructs

Constructs are the basic building block of **CDK8s**. They are the instrument that enables composition and creation of higher-level abstractions through normal object-oriented classes.

If you come from the Kubernetes world, you can think of constructs as programmatically defined Helm Charts. The nice thing about constructs being "programmatically defined" is that we can use them to leverage the full power of object-oriented programming. For example:

- You can express the abstraction's API using strong-typed data types
- You can express rich interactions with methods and properties
- You can create polymorphic programming models through interfaces and base classes
- Share them through regular package managers
- Test them using our familiar testing tools and techniques
- Version them
- …and do all that stuff that we've been doing with software in the past 20 years.

So let's create our first Kubernetes construct. We'll call it `WebService` and it will basically be a generalization of the **hello world** program. It's actually quite useful.

For example, this one line will add a hello world service to our chart:

TypeScript

```
new WebService(this, 'hello-k8s', {
  image: 'paulbouwer/hello-kubernetes:1.7'
});
```

It can also be customized through an API:

```
new WebService(this, 'hello-k8s', {
  image: 'paulbouwer/hello-kubernetes:1.7',
  containerPort: 8080,
  replicas: 10
});
```

Python

```
WebService(self, 'hello',
           image='paulbouwer/hello-kubernetes:1.7')
```

It can also be customized through an API:

```
WebService(self, 'hello-k8s',
           image='paulbouwer/hello-kubernetes:1.7',
           container_port=8080,
           replicas=10)
```

Java

```
new WebService(this, "hello-k8s", new WebServiceProps.Builder()
  .image("paulbouwer/hello-kubernetes:1.7")
  .build());
```

It can also be customized through an API:

```
new WebService(this, "hello-k8s", new WebServiceProps.Builder()
  .image("paulbouwer/hello-kubernetes:1.7")
  .replicas(2)
  .build());
```

Go

```
NewWebService(chart, jsii.String("hello"), &WebServiceProps{
  Image:    jsii.String("paulbouwer/hello-kubernetes:1.7"),
})
```

It can also be customized through an API:

```
NewWebService(chart, jsii.String("hello"), &WebServiceProps{
  Image:         jsii.String("paulbouwer/hello-kubernetes:1.7"),
  ContainerPort: jsii.Number(8080),
  Replicas:      jsii.Number(10),
})
```

Here's how to implement `WebService`:

TypeScript

Create a file `lib/web-service.ts` (the convention is to use `lib` for reusable components):

```
import { Construct } from 'constructs';
import { Names } from 'cdk8s';
import { KubeDeployment, KubeService, IntOrString } from '../imports/k8s';

export interface WebServiceProps {
  /**
   * The Docker image to use for this service.
   */
  readonly image: string;

  /**
   * Number of replicas.
   *
   * @default 1
   */
  readonly replicas?: number;

  /**
   * External port.
   *
   * @default 80
   */
  readonly port?: number;

  /**
   * Internal port.
   *
   * @default 8080
   */
  readonly containerPort?: number;
}

export class WebService extends Construct {
  constructor(scope: Construct, id: string, props: WebServiceProps) {
```

```
    super(scope, id);

    const port = props.port || 80;
    const containerPort = props.containerPort || 8080;
    const label = { app: Names.toDnsLabel(this) };
    const replicas = props.replicas ?? 1;

    new KubeService(this, 'service', {
      spec: {
        type: 'LoadBalancer',
        ports: [ { port, targetPort: IntOrString.fromNumber(containerPort) }
],
        selector: label
      }
    });

    new KubeDeployment(this, 'deployment', {
      spec: {
        replicas,
        selector: {
          matchLabels: label
        },
        template: {
          metadata: { labels: label },
          spec: {
            containers: [
              {
                name: 'app',
                image: props.image,
                ports: [ { containerPort } ]
              }
            ]
          }
        }
      }
    });
  }
}
```

Now, let's edit `main.ts` and use our new construct:

```
import { Construct } from 'constructs';
import { App, Chart, ChartProps } from 'cdk8s';
import { WebService } from './lib/web-service';


export class MyChart extends Chart {
  constructor(scope: Construct, id: string, props: ChartProps = { }) {
    super(scope, id, props);

    new WebService(this, 'hello', { image: 'paulbouwer/hello-kubernetes:1.7',
replicas: 2 });
    new WebService(this, 'ghost', { image: 'ghost', containerPort: 2368 });

  }
}
```

```
const app = new App();
new MyChart(app, 'hello');
app.synth();
```

Python

Create a file `webservice.py` with the following content:

```
from constructs import Construct, Node

from imports import k8s


class WebService(Construct):
    def __init__(self, scope: Construct, id: str, *,
                 image: str,
                 replicas: int = 1,
                 port: int = 80,
                 container_port: int = 8080):
        super().__init__(scope, id)

        label = {'app': Node.of(self).unique_id}

        k8s.KubeService(self, 'service',
                        spec=k8s.ServiceSpec(
                          type='LoadBalancer',
                          ports=[k8s.ServicePort(port=port,
target_port=k8s.IntOrString.from_number(container_port))],
                          selector=label))

        k8s.KubeDeployment(self, 'deployment',
                           spec=k8s.DeploymentSpec(
                               replicas=replicas,
                               selector=k8s.LabelSelector(match_labels=label),
                               template=k8s.PodTemplateSpec(
                               metadata=k8s.ObjectMeta(labels=label),
                               spec=k8s.PodSpec(
                                 containers=[
                                     k8s.Container(
                                         name='app',
                                         image=image,
                                         ports=
[k8s.ContainerPort(container_port=container_port)])])))))
```

Now, let's edit `main.py` and use our new construct:

```
#!/usr/bin/env python
from constructs import Construct
from cdk8s import App, Chart

from webservice import WebService

class MyChart(Chart):
```

```
    def __init__(self, scope: Construct, id: str):
        super().__init__(scope, id)

        WebService(self, 'hello', image='paulbouwer/hello-kubernetes:1.7',
replicas=2)
        WebService(self, 'ghost', image='ghost', container_port=2368)


app = App()
MyChart(app, "hello")

app.synth()
```

Java

`src/main/java/com/mycompany/app/WebService.java`

```
package com.mycompany.app;

import software.constructs.Construct;

import java.util.ArrayList;
import java.util.List;
import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

import imports.k8s.IntOrString;
import imports.k8s.LabelSelector;
import imports.k8s.ObjectMeta;
import imports.k8s.PodTemplateSpec;
import imports.k8s.KubeService;
import imports.k8s.KubeServiceProps;
import imports.k8s.ServicePort;
import imports.k8s.ServiceSpec;
import imports.k8s.DeploymentSpec;
import imports.k8s.PodSpec;
import imports.k8s.Container;
import imports.k8s.ContainerPort;
import imports.k8s.KubeDeployment;
import imports.k8s.KubeDeploymentProps;

public class WebService extends Construct
{

    public WebService(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public WebService(final Construct scope, final String id, final
WebServiceProps options) {
        super(scope, id);

        final int portNumber = Optional.of(options.getPort()).orElse(80);
```

```
        final int containerPortNumber =
Optional.of(options.getContainerPort()).orElse(8080);
        final int replicas = Optional.of(options.getReplicas()).orElse(1);
        final String image = options.getImage();

        // Defining a LoadBalancer Service
        final String serviceType = "LoadBalancer";
        final Map<String, String> selector = new HashMap<>();
        selector.put("app", "hello-k8s");
        final List<ServicePort> servicePorts = new ArrayList<>();
        final ServicePort servicePort = new ServicePort.Builder()
            .port(portNumber)
            .targetPort(IntOrString.fromNumber(containerPortNumber))
            .build();
        servicePorts.add(servicePort);
        final ServiceSpec serviceSpec = new ServiceSpec.Builder()
            .type(serviceType)
            .selector(selector)
            .ports(servicePorts)
            .build();
        final KubeServiceProps serviceProps = new KubeServiceProps.Builder()
            .spec(serviceSpec)
            .build();

        new KubeService(this, "service", serviceProps);

        // Defining a Deployment
        final LabelSelector labelSelector = new
LabelSelector.Builder().matchLabels(selector).build();
        final ObjectMeta objectMeta = new
ObjectMeta.Builder().labels(selector).build();
        final List<ContainerPort> containerPorts = new ArrayList<>();
        final ContainerPort containerPort = new ContainerPort.Builder()
            .containerPort(containerPortNumber)
            .build();
        containerPorts.add(containerPort);
        final List<Container> containers = new ArrayList<>();
        final Container container = new Container.Builder()
            .name("web")
            .image(image)
            .ports(containerPorts)
            .build();
        containers.add(container);
        final PodSpec podSpec = new PodSpec.Builder()
            .containers(containers)
            .build();
        final PodTemplateSpec podTemplateSpec = new PodTemplateSpec.Builder()
            .metadata(objectMeta)
            .spec(podSpec)
            .build();
        final DeploymentSpec deploymentSpec = new DeploymentSpec.Builder()
            .replicas(replicas)
            .selector(labelSelector)
            .template(podTemplateSpec)
            .build();
        final KubeDeploymentProps deploymentProps = new
KubeDeploymentProps.Builder()
```

```
            .spec(deploymentSpec)
            .build();

        new KubeDeployment(this, "deployment", deploymentProps);
    }
}
```

src/main/java/com/mycompany/app/WebServiceProps.java

```
package com.mycompany.app;

public class WebServiceProps
{
    private String image;
    private int replicas;
    private int port;
    private int containerPort;

    public WebServiceProps(final String image, final int replicas, final int
port, final int containerPort) {
        this.image = image;
        this.replicas = replicas;
        this.port = port;
        this.containerPort = containerPort;
    }

    public String getImage() {
        return this.image;
    }

    public int getReplicas() {
        return this.replicas;
    }

    public int getPort() {
        return this.port;
    }

    public int getContainerPort() {
        return this.containerPort;
    }

    public static final class Builder {
        private String image;
        private int replicas = 1;
        private int port = 80;
        private int containerPort = 8080;

        public Builder image(String image) {
            this.image = image;
            return this;
        }

        public Builder replicas(int replicas) {
            this.replicas = replicas;
            return this;
```

```
        }

        public Builder port(int port) {
            this.port = port;
            return this;
        }

        public Builder containerPort(int containerPort) {
            this.containerPort = containerPort;
            return this;
        }

        public WebServiceProps build() {
            return new WebServiceProps(image, replicas, port, containerPort);
        }
    }
}
```

`src/main/java/com/mycompany/app/Main.java`

```
package com.mycompany.app;

import software.constructs.Construct;

import org.cdk8s.App;
import org.cdk8s.Chart;
import org.cdk8s.ChartProps;

public class Main extends Chart
{

    public Main(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public Main(final Construct scope, final String id, final ChartProps
options) {
        super(scope, id, options);

        new WebService(this, "hello", new WebServiceProps.Builder()
          .image("paulbouwer/hello-kubernetes:1.7")
          .replicas(2)
          .build());

        new WebService(this, "ghost", new WebServiceProps.Builder()
          .image("ghost")
          .containerPort(2368)
          .build());
    }

    public static void main(String[] args) {
        final App app = new App();
        new Main(app, "web-service-java");
        app.synth();
    }
}
```

Go

Create a file `webservice.go` with the following content:

```
package main

import (
  "example.com/hello/imports/k8s"
  "github.com/aws/constructs-go/constructs/v3"
  "github.com/aws/jsii-runtime-go"
)

type WebServiceProps struct {
  constructs.ConstructOptions
  Image         *string
  Replicas      *float64
  Port          *float64
  ContainerPort *float64
}

func NewWebService(scope constructs.Construct, id *string, props
*WebServiceProps) constructs.Construct {
  var cprops constructs.ConstructOptions
  if props != nil {
    cprops = props.ConstructOptions
  }
  construct := constructs.NewConstruct(scope, id, &cprops)

  replicas := props.Replicas
  if replicas == nil {
    replicas = jsii.Number(1)
  }

  port := props.Port
  if port == nil {
    port = jsii.Number(80)
  }

  containerPort := props.ContainerPort
  if containerPort == nil {
    containerPort = jsii.Number(8080)
  }

  label := map[string]*string{
    "app": constructs.Node_Of(construct).Id(),
  }

  k8s.NewKubeService(construct, jsii.String("service"), &k8s.KubeServiceProps{
    Spec: &k8s.ServiceSpec{
      Type: jsii.String("LoadBalancer"),
      Ports: &[]*k8s.ServicePort{{
        Port:       port,
        TargetPort: k8s.IntOrString_FromNumber(containerPort),
```

```
      }},
      Selector: &label,
    },
  })

  k8s.NewKubeDeployment(construct, jsii.String("deployment"),
&k8s.KubeDeploymentProps{
    Spec: &k8s.DeploymentSpec{
      Replicas: replicas,
      Selector: &k8s.LabelSelector{MatchLabels: &label},
      Template: &k8s.PodTemplateSpec{
        Metadata: &k8s.ObjectMeta{Labels: &label},
        Spec: &k8s.PodSpec{
          Containers: &[]*k8s.Container{{
            Name:  jsii.String("web"),
            Image: props.Image,
            Ports: &[]*k8s.ContainerPort{{ContainerPort: containerPort}},
          }},
        },
      },
    },
  })

  return construct
}
```

Now, let's edit `main.go` and use our new construct:

```
package main

import (
  "github.com/aws/constructs-go/constructs/v3"
  "github.com/aws/jsii-runtime-go"
  "github.com/cdk8s-team/cdk8s-core-go/cdk8s"
)

type MyChartProps struct {
  cdk8s.ChartProps
}

func NewMyChart(scope constructs.Construct, id string, props *MyChartProps)
cdk8s.Chart {
  var cprops cdk8s.ChartProps
  if props != nil {
    cprops = props.ChartProps
  }
  chart := cdk8s.NewChart(scope, jsii.String(id), &cprops)

  NewWebService(chart, jsii.String("hello"), &WebServiceProps{
    Image:    jsii.String("paulbouwer/hello-kubernetes:1.7"),
    Replicas: jsii.Number(2),
  })

  NewWebService(chart, jsii.String("ghost"), &WebServiceProps{
    Image:         jsii.String("ghost"),
    ContainerPort: jsii.Number(2368),
  })
```

```
  })

  return chart
}


func main() {
  app := cdk8s.NewApp(nil)
  NewMyChart(app, "hello", nil)
  app.Synth()
}
```

As you can see, we now add `WebService` constructs inside our chart: one that runs the `paulbouwer/hello-kubernetes` image and one with an installation of ghost.