Creating a New Serverless Application Project

In this lesson, we will learn to create a new serverless application project.

We'll cover the following

Jenkins X does its best to be easy for everyone and not to introduce unnecessary complexity. True to that goal, there is nothing special users need to do to create a new project with serverless deployments. There is no additional command, nor are there any extra arguments. The `jx edit deploy` command already told Jenkins X that we want all new projects to be serverless by default, so all we have to do is create a new quick start.

Creating a new quick start project

```
jx create quickstart \
    --filter golang-http \
    --project-name jx-knative \
    --batch-mode
```

As you can see, that command was no different than any other quick start we created earlier. We needed a project with a unique name, so the only change is that this one is called `jx-knative`.

If you look at the output, there is nothing new there either. If someone else changed the team's deployment kind, you wouldn't even know that a quick start will end with the first release running in the staging environment in the serverless fashion.

Browsing to project directory

There is one difference, though, and we need to enter the project directory to find it.

```
cd jx-knative
```

Inspecting the `values.yaml` file

Now, there is only one value that matters, and it is located in `values.yaml`.

```
cat charts/jx-knative/values.yaml
```

The output, limited to the relevant parts, is as follows.

```
...
# enable this flag to use knative serve to deploy the app
```

```
knativeDeploy: true
...
```

As you can see, the `knativeDeploy` variable is set to `true`. All the past projects, at least those created after May 2019, had that value set to `false`, simply because we did not have the **Gloo** addon installed, and our deployment setting was set to `default` instead of `knative`. But, now that we changed that, `knativeDeploy` will be set to `true` for all the new projects unless we change the deployment setting again.

Now, you might be thinking to yourself that a **Helm** variable does not mean much by itself unless it is used. You are right; it is only a variable, and we have yet to discover the reason for its existence.

Inspecting the `templates` directory

Let's take a look at what we have in the Chart's `templates` directory.

```
ls -1 charts/jx-knative/templates
```

The output is as follows.

```
NOTES.txt
_helpers.tpl
deployment.yaml
ksvc.yaml
service.yaml
```

We are already familiar with `deployment.yaml` and `service.yaml` files, but we might have missed a crucial detail. Let's take a look at what's inside one of them.

```
cat charts/jx-knative/templates/deployment.yaml
```

The output, limited to the top and bottom parts, is as follows.

```
{{- if .Values.knativeDeploy }}
{{- else }}
...
{{- end }}
```

We have the `{{- if .Values.knativeDeploy }}` instruction that immediately continues into `{{- else }}`, while the whole definition of the deployment is between `{{- else }}` and `{{- end }}`. While that might look strange at first, it actually means that the Deployment resource should be created only if `knativeDeploy` is set to `false`

If you take a look at the `service.yaml` file, you'll notice the same pattern. In both cases, the resources are created only if we didn't choose to use **Knative** deployments. That brings us to the `ksvc.yaml` file.

Inspecting the `ksvc.yaml` file.

```
cat charts/jx-knative/templates/ksvc.yaml
```

The output is as follows.

```
{{- if .Values.knativeDeploy }}
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
{{- if .Values.service.name }}
  name: {{ .Values.service.name }}
{{- else }}
  name: {{ template "fullname" . }}
{{- end }}
  labels:
    chart: "{{ .Chart.Name }}-{{ .Chart.Version | replace "+" "_" }}"
spec:
  runLatest:
    configuration:
      revisionTemplate:
        spec:
          container:
            image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
            imagePullPolicy: {{ .Values.image.pullPolicy }}
            env:
{{- range $pkey, $pval := .Values.env }}
            - name: {{ $pkey }}
              value: {{ quote $pval }}
{{- end }}
            livenessProbe:
              httpGet:
                path: {{ .Values.probePath }}
              initialDelaySeconds: {{ .Values.livenessProbe.initialDelaySeconds }}
              periodSeconds: {{ .Values.livenessProbe.periodSeconds }}
              successThreshold: {{ .Values.livenessProbe.successThreshold }}
              timeoutSeconds: {{ .Values.livenessProbe.timeoutSeconds }}
            readinessProbe:
              httpGet:
                path: {{ .Values.probePath }}
              periodSeconds: {{ .Values.readinessProbe.periodSeconds }}
              successThreshold: {{ .Values.readinessProbe.successThreshold }}
              timeoutSeconds: {{ .Values.readinessProbe.timeoutSeconds }}
            resources:
{{ toYaml .Values.resources | indent 14 }}
{{- end }}
```

To begin with, you can see that the conditional logic is reversed. The resource defined in that file will be created only if the `knativeDeploy` variable is set to `true`.

We won't go into details of the specification. I'll only say that it is similar to what we'd define as a Pod specification, and leave you to explore **Knative Serving API spec** on your own. Where **Knative** definition differs significantly from what we're used to when we work with **Deployments** and **StatefulSets**, is that we don't need to specify many of the things. There is no need for creating a **Deployment**, that defines a **ReplicaSet**, that defines pod templates. There is no definition of a Service associated with the pods. **Knative** will create all the objects required to convert our Pods into a scalable solution accessible to our users.

We can think of the **Knative** definition as being more developer-friendly than other Kubernetes resources. It dramatically simplifies things by making some assumptions. All the Kubernetes resources we're used to seeing like **Deployment**, **ReplicaSet**, and **Service** will still be created along with quite a few others. The significant difference is not only in what will be running in Kubernetes but also in how we define what we need. By focusing only on what matters, **Knative** removes clutter from `YAML` files we tend to create.

Checking the activities of `jx-knative`

Now, let's see whether the activity of the pipeline run initiated by pushing the initial commit to the newly created repository is finished.

```
jx get activities \
    --filter jx-knative \
    --watch
```

Unless you are the fastest reader on earth, the pipeline run should have finished, and you'll notice that there is no difference in the steps. It is the same no matter if we're using serverless or any other type of deployment. So, feel free to stop the activity by pressing *ctrl+c*, and we'll take a look at the pods and see whether that shows anything interesting.

Checking the activities of application deployed on staging

Before we look at the pods of the new application deployed to the staging environment, we'll confirm that the latest run of the staging environment pipeline is finished.

```
jx get activities \
    --filter environment-jx-rocks-staging/master \
    --watch
```

Feel free to press *ctrl+c* when the staging environment pipeline run is finished.

Getting the Pods

Now we can have a look at the Pods running as part of our serverless application.

```
kubectl \
    --namespace jx-staging \
    get pods \
    --selector serving.knative.dev/service=jx-knative
```

The output is as follows:

```
NAME            READY STATUS   RESTARTS AGE
jx-knative-... 2/2   Running 0        84s
```

> ⚠ If the output states that `no resources` were `found`, enough time has passed without any traffic and the application was scaled to zero replicas. We'll see a similar effect and comment on it a few more times. Just keep in mind that the next command that describes the Pod will not work if the Pod was already removed.

The Pod is there, just as we expected. The strange thing is the number of containers. There are two, even though our application needs only one.

Describing the pods

Let's describe the pod and see what we'll get.

```
kubectl \
    --namespace jx-staging \
    describe pod \
    --selector serving.knative.dev/service=jx-knative
```

The relevant parts of the output are as follows:

```
...
Containers:
  ...
  queue-proxy:
    ...
    Image: gcr.io/knative-releases/github.com/knative/serving/cmd/queue@sha256:...
    ...
```

The `queue-proxy` container was “injected” into the Pod. It serves as a proxy responsible for request queue parameters, and it reports metrics to the Autoscaler. In other words, requests are reaching our application through this container. Later on, when we explore scaling our **Knative**-based applications, that container will be the one responsible for providing metrics used to make scaling-related decisions.

Getting all resources of the namespace `jx-staging`

Let’s see what other resources were created for us.

```
kubectl \
    --namespace jx-staging \
    get all
```

The relevant parts of the output are as follows:

```
...
service/jx-knative               ...
service/jx-knative-svtns-service ...
...
deployment.apps/jx-knative-...
...
replicaset.apps/jx-knative-...
...
podautoscaler.autoscaling.internal.knative.dev/jx-knative-...
...
image.caching.internal.knative.dev/jx-knative-...
...
clusteringress.networking.internal.knative.dev/route-...
...
route.serving.knative.dev/jx-knative ...
...
service.serving.knative.dev/jx-knative ...
...
configuration.serving.knative.dev/jx-knative ...
...
revision.serving.knative.dev/jx-knative-...
```

As you can see, quite a few resources were created from a single `YAML` definition with a (`serving.knative.dev`) `Service`. There are some core Kubernetes resources we are likely already familiar with, like:

- **Deployment**
- **ReplicaSet**
- **Pod**
- **Service**

Even if that was all we had, we could already conclude that **Knative** service simplifies things since it would take us approximately double the lines in `YAML` to define the same resources (**Deployment** and **Service**, the rest was created by those) ourselves. But we got so much more. There are seven or more resources created from **Knative** specific **Custom Resource Definitions** (CRDs) and their responsibilities differ. One (`podautoscaler.autoscaling`) is in charge of scaling based on the number of requests or other metrics, the other (`image.caching`) of caching the image so that boot-up time is faster, few are making sure that networking is working as expected, and so on. We'll get more familiar with those features later.

There is one inconvenience, though. As of today (July 7, 2019), `get applications` does not report **Knative**-based applications correctly.

```
jx get applications --env staging
```

The output is as follows.

```
APPLICATION STAGING PODS URL
go-demo-6   1.0.221 3/3  http://go-demo-6.jx-staging.35.190.185.247.nip.io
knative     svtns
```

The serverless application is not reported correctly by Jenkins X. Hopefully, that will be fixed soon. Until then, feel free to monitor the progress through the issue 4635.

Querying `ksvc` to retrieve the domain

Knative defines its own **Service** as one that, just like those available in the Kubernetes core, can be queried to get the domain through which we can access the application. We can query it just as we would query the normal **Service**, the main difference being that it is called `ksvc`, instead of `svc`. We'll use it to retrieve the domain through which we can access and, therefore, test whether the newly deployed serverless application works as expected.

```
ADDR=$(kubectl \
    --namespace jx-staging \
    get ksvc jx-knative \
    --output jsonpath="{.status.url}")

echo $ADDR
```

The output should be similar to the one that follows.

```
jx-knative.jx-staging.35.243.171.144.nip.io
```

As you can see, the pattern is the same no matter whether it is a “normal” or a **Knative** service. Jenkins X is making sure that the URLTemplate we explored in the Changing URL Patterns subchapter is applied no matter the type of the Service or the **Ingress** used to route external requests to the application. In this case, it is the default one that combines the name of the service (`jx-knative`) with the environment (`jx-staging`) and the cluster domain (`35.243.171.144.nip.io`).

Confirming if the application is working

Now comes the moment of truth. Is our application working? Can we access it?

```
curl "$ADDR"
```

The good news is that we did get the `Hello` greeting as the output, so the application is working. But that might have been the slowest response you ever saw from such a simple application.

Reason for the slow response

Why did it take so long? The answer to that question lies in the scaling nature of serverless applications.

Since no one sent a request to the app before, there was no need for it to run any replica, and **Knative** scaled it down to zero a few minutes after it was deployed. The moment we sent the first request, **Knative** detected it and initiated scaling that resulted in one replica running inside the cluster. As a result, we received the familiar greeting, only after the image is pulled, the Pod was started, and the application inside it was initiated. Don’t worry about that slowness since it manifests itself only initially before **Knative** creates the cache. You’ll see soon that the boot-up time will be fast from now on.

Checking the Pod

So, let's take a look at that famous Pod that was created out of thin air.

```
kubectl \
    --namespace jx-staging \
    get pods \
    --selector serving.knative.dev/service=jx-knative
```

The output is as follows:

```
NAME           READY STATUS  RESTARTS AGE
jx-knative-... 2/2   Running 0        24s
```

We can see a single Pod created a short while ago. Now, let's observe what we'll get with a little bit of patience.

Please wait for seven minutes or more before executing the command that follows.

```
kubectl --namespace jx-staging \
    get pods \
    --selector serving.knative.dev/service=jx-knative
```

The output shows that `no resources` were `found`. The Pod is gone. No one was using our application, so **Knative** removed it to save resources. It scaled it down to zero replicas.

Knative parameter configuration

If you're anything like me, you must be wondering about the configuration.

- *What are the parameters governing **Knative** scaling decisions?*
- *Can they be fine-tuned?*

The configuration that governs scaling is stored in the `config-autoscaler` **ConfigMap**.

```
kubectl --namespace knative-serving \
    describe configmap config-autoscaler
```

The output is a well-documented configuration example that explains what we'd need to do to change any aspect of **Knative**'s scaling logic. It is too big to be need to do to change any aspect of **Knative**'s scaling logic. It is too big to be presented in a book, so I'll leave it to you to explore it.

In a nutshell, **Knative**'s scaling algorithm is based on the average number of concurrent requests. By default, it will try to target a hundred parallel requests served by a single Pod. That would mean that if there are three hundred concurrent requests, the system should scale to three replicas so that each can handle a hundred.

Now, the calculation for the number of Pods is not as simple as the number of concurrent requests divided by a hundred (or whatever we defined the `container-concurrency-target-default` variable). The **Knative** scaler calculated the average number of parallel requests over a sixty seconds window, so it takes a minute for the system to stabilize at the desired level of concurrency. There is also a six seconds window that might make the system enter into the panic mode if, during that period, the number of requests is more than double the target concurrency.

I'll let you go through the documentation and explore the details. What matters, for now, is that the system, as it is now, should scale the number of Pods if we send it more than a hundred parallel requests.

Checking for scale down

Before we test **Knative**'s scaling capabilities, we'll check whether the application scaled down to zero replicas.

```
kubectl \
    --namespace jx-staging \
    get pods \
    --selector serving.knative.dev/service=jx-knative
```

If the output states that `no resources` were `found`, the Pods are gone, and we can proceed. Otherwise, wait for a while longer and repeat the previous command.

We ensured that no Pods are running only to simplify the "experiment" that follows. When nothing is running, the calculation is as simple as the number of concurrent requests divided by the target concurrency equals the number of replicas. Otherwise, the calculation would be more complicated than that, and our "experiment" would need to be more elaborated. We won't go into those details since I'm sure that you can gather such info from the **Knative**'s documentation. Instead, we'll perform a simple experiment and check what happens when nothing is running.

Siege for testing parallel requests

So, we want to see what the result of sending hundreds of parallel requests to the application. We'll use **Siege** for that. It is a small and simple tool that allows us to stress test a single URL. It does that by sending parallel requests to a specific address.

Since I want to save you from installing yet-another-tool, we'll run **Siege** inside a Pod with a container based on the yokogawa/siege image.

Running Siege and retrieving Pods

Now, we're interested in finding out how **Knative** deployments scale based on the number of requests, so we'll also need to execute `kubectl get pod` command to see how many Pods were created. But, since **Knative** scales both up and down, we'll need to be fast. We have to make sure that the Pods are retrieved as soon as the siege is finished. We'll accomplish that by concatenating the two commands into one.

```
kubectl run siege \
    --image yokogawa/siege \
    --generator "run-pod/v1" \
     -it --rm \
     -- --concurrent 300 --time 20S \
     "$ADDR" \
     && kubectl \
     --namespace jx-staging \
    get pods \
    --selector serving.knative.dev/service=jx-knative
```

We executed three hundred concurrent requests (`-c 300`) for twenty seconds (`-t 20S`). Immediately after that, we retrieved the Pods related to the `jx-knative` deployment. The combined output is as follows.

```
If you don't see a command prompt, try pressing enter.

Lifting the server siege...      done.

Transactions:                    4920 hits
Availability:                  100.00 %
Elapsed time:                   19.74 secs
Data transferred:                0.20 MB
Response time:                   1.16 secs
Transaction rate:              249.24 trans/sec
Throughput:                      0.01 MB/sec
Concurrency:                   289.50
Successful transactions:         4920
Failed transactions:                0
```

```
Longest transaction:            6.25
Shortest transaction:           0.14

FILE: /var/log/siege.log
You can disable this annoying message by editing
the .siegerc file in your home directory; change
the directive 'show-logfile' to false.
The session ended, resume using 'kubectl attach siege -c siege -i -t' command when the pod is runn
pod "siege" deleted
NAME                            READY STATUS  RESTARTS AGE
jx-knative-cvl52-deployment-... 2/2   Running 0        18s
jx-knative-cvl52-deployment-... 2/2   Running 0        20s
jx-knative-cvl52-deployment-... 2/2   Running 0        18s
```

The `siege` output shows us that it successfully executed `4920` requests within `19.74` seconds, and all that was done with the concurrency of almost three hundred.

What is more interesting is that we got three Pods of the `jx-knative` application. If we go back to the values in the **ConfigMap** `config-autoscaler`, we'll see that **Knative** tries to maintain one replica for every hundred concurrent requests. Since we sent almost triple that number of concurrent requests, we got three Pods. Later on, we'll see what **Knative** does when that concurrency changes. For now, we'll focus on "fine-tuning" **Knative**'s configuration specific to our application.

Modifying the `ksvc.yaml` file

```
cat charts/jx-knative/templates/ksvc.yaml \
    | sed -e \
    's@revisionTemplate:@revisionTemplate:\
        metadata:\
          annotations:\
            autoscaling.knative.dev/target: "3"\
            autoscaling.knative.dev/maxScale: "5"@g' \
    | tee charts/jx-knative/templates/ksvc.yaml
```

We modified the `ksvc.yaml` file by adding a few annotations. Specifically, we set the `target` to `3` and `maxScale` to `5`. The former should result in **Knative** scaling our application with every three concurrent requests. The latter, on the other hand, will prevent the system from having more than `5` replicas. As a result, we'll have better-defined parameters that will be used to decide when to scale, but we'll also be protected from the danger of "getting more than we are willing to have."

Pushing the changes to Github and observing the activities

Now, let's push the changes to the GitHub repository and confirm that the pipeline

Now, let's push the changes to the GitLab repository and confirm that the pipeline run that will be triggered by that will complete successfully.

```
git add .

git commit -m "Added Knative target"

git push --set-upstream origin master

jx get activities \
    --filter jx-knative \
    --watch
```

Feel free to press the *ctrl+c* key to stop watching the activities when the run is finished.

Confirming the release deployed to staging

As before, we'll also confirm that the new release was deployed to the staging environment by monitoring the activities of the `environment-jx-rocks-staging` pipeline runs.

```
jx get activities \
    --filter environment-jx-rocks-staging/master \
    --watch
```

Please press *ctrl+c* to cancel the watcher once the new activity is finished.

Sending a request to the application

Finally, the last step we'll execute before putting the application under siege is to double-check that it is still reachable by sending a single request.

```
curl "$ADDR/"
```

You should see the familiar message, and now we're ready to put the app under siege.

Running Siege and retrieving pods related to the `jx-knative`

Just as before, we'll concatenate the command that will output the pods related to the `jx-knative` app.

```
kubectl run siege \
    --image yokogawa/siege \
    --generator "run-pod/v1" \
     -it --rm \

     -- --concurrent 400 --time 60S \
     "$ADDR" \
     && kubectl \
     --namespace jx-staging \
    get pods \
    --selector serving.knative.dev/service=jx-knative
```

We sent a stream of four hundred (`-c 400`) concurrent requests over one minute (`-t 60S`). The output is as follows.

```
If you don't see a command prompt, try pressing enter.

Lifting the server siege...      done.

Transactions:                  19078 hits
Availability:                 100.00 %
Elapsed time:                  59.63 secs
Data transferred:               0.78 MB
Response time:                  1.04 secs
Transaction rate:             319.94 trans/sec
Throughput:                     0.01 MB/sec
Concurrency:                  332.52
Successful transactions:       19078
Failed transactions:               0
Longest transaction:            8.29
Shortest transaction:           0.01

FILE: /var/log/siege.log
You can disable this annoying message by editing
the .siegerc file in your home directory; change
the directive 'show-logfile' to false.
The session ended, resume using 'kubectl attach siege -c siege -i -t' command when the pod is runn
pod "siege" deleted
NAME            READY STATUS   RESTARTS AGE
jx-knative-.... 2/2   Running 0        58s
jx-knative-.... 2/2   Running 0        58s
jx-knative-.... 2/2   Running 0        61s
jx-knative-.... 2/2   Running 0        58s
jx-knative-.... 2/2   Running 0        58s
```

If we'd focus only on the `target` annotation we set to `3`, we would expect to have over one hundred Pods, one for every three concurrent requests. But, we also set the `maxScale` annotation to `5`. As a result, only five Pods were created. **Knative** started scaling the application to accommodate three requests per Pod rule, but it capped at five to match the `maxScale` annotation.

Checking pods after some time

Now, let's see what happens a while later. Please execute the command that follows a minute (but not much more) after than the previous command.

```
kubectl \
    --namespace jx-staging \
    get pods \
    --selector serving.knative.dev/service=jx-knative
```

The output is as follows.

```
NAME           READY STATUS  RESTARTS AGE
jx-knative-... 2/2   Running 0        2m32s
```

As we can see, **Knative** scaled-down the application to one replica short while after the burst of requests stopped. It intentionally did not scale down to zero right away to avoid potentially slow boot-up time. Now, that will change soon as well, so let's see what happens after another short pause.

Please wait for some five to ten minutes more before executing the command that follows.

```
kubectl \
    --namespace jx-staging \
    get pods \
    --selector serving.knative.dev/service=jx-knative
```

This time the output states that `no resources` were `found`. **Knative** observed that no traffic was coming to the application for some time and decided that it should scale down the application to zero replicas. As you already saw, that will change again as soon as we send additional requests to the application. For now, we'll focus on one more annotation.

How to prevent scaling down to zero replicas

In some cases, we might not want to allow **Knative** to scale to zero replicas. Our application's boot-time might be too long, and we might not want to risk our users waiting for too long. Now, that would not often happen since such situations would occur only if there is no traffic for a prolonged time. Still, some applications might take seconds or even minutes to boot up. I'll skip a discussion in which I would try to convince you that you should not have such applications or that, if you do, you should redesign them or even throw them to trash and start over. Instead, I'll just assume that you do have an app that is slow to boot up but that you still see the benefits in adopting **Knative** for, let's say, scaling up.

So, how do we prevent it from scaling to zero replicas, and yet allow it to scale to any other number?

Let’s give it a try with the command that follows.

```
cat charts/jx-knative/templates/ksvc.yaml \
    | sed -e \
    's@autoscaling.knative.dev/target: "3"@autoscaling.knative.dev/target: "3"\
            autoscaling.knative.dev/minScale: "1"@g' \
    | tee charts/jx-knative/templates/ksvc.yaml
```

We used `sed` to add `minScale` annotation set to `1`. You can probably guess how it will behave, but we’ll still double-check that everything works as expected.

Before we proceed, please note that we used only a few annotations and that **Knative** offers much more fine-tuning. As an example, we could tell **Knative** to use HorizontalPodAutoscaler for scaling decisions. I’ll leave it up to you too to check out the project’s documentation, and we’ll get back to our task of preventing **Knative** from scaling our application to zero replicas.

```
git add .

git commit -m "Added Knative minScale"

git push

jx get activities \
    --filter jx-knative \
    --watch
```

We pushed changes to the GitHub repository, and we started watching the `jx-native` activity created as a result of making changes to our source code.

Feel free to stop watching the activities (press *ctrl+c*) once you confirm that the newly started pipeline run completed successfully.

Just as before, we’ll also confirm that the pipeline run of the staging environment completed successfully as well.

```
jx get activities \
    --filter environment-jx-rocks-staging/master \
    --watch
```

Just as a few moments ago, press *ctrl+c* when the new pipeline run is finished.

Just as a few moments ago, press *ctrl+c* when the new pipeline run is finished.

Now, let's see how many Pods we have.

```
kubectl \
    --namespace jx-staging \
    get pods \
    --selector serving.knative.dev/service=jx-knative
```

The output should show that only one Pod is running. However, that might not be definitive proof that, from now on, **Knative** will never scale our app to zero replicas. To confirm that "for real", please wait for, let's say, ten minutes, before retrieving the Pods again.

```
kubectl \
    --namespace jx-staging \
    get pods \
    --selector serving.knative.dev/service=jx-knative
```

The output is still the same single Pod we saw earlier. That proves that our application's minimum number of replicas is maintained at `1`, even if it does not receive any requests for a prolonged time.

There's probably no need to give you instructions on how to check whether the application scales up if we start sending it a sufficient number of concurrent requests. Similarly, we should know by now that it will also scale down if the number of simultaneous requests decreases. Everything is the same as it was, except that the minimum number of replicas is now `1` (it is zero by default).

> This last exercise of adding the `minScale` annotation converted our application from serverless to a microservice or whichever other architecture our app had.

In the next lesson, we will discuss how to use serverless deployments with pull requests.