Deploying your first application to Substrate

Document Level Classification

200

Introduction

This tutorial is designed to give you an introduction to Substrate using a sample application deployed to your Substrate infrastructure.

Prerequisites

1. *Complete the Prerequisites to ensure you have the appropriate access and local developer toolchain setup.*
2. *Verify access to your Substrate infrastructure. This tutorial interacts with your Substrate infrastructure using an authenticated terminal session.*

Placeholders

The instructions and code snippets in this tutorial use the following placeholders. Make sure to replace them appropriately based on your Substrate infrastructure.

Placeholder	***Description***	***Example***
`<SUBSTRATE_ACCOUNT>`	*The name of your Substrate account.*	`abcd-dev`
`<SUBSTRATE_PREFIX>`	*The prefix in your Substrate account name*	`abcd`
`<SUBSTRATE_CLUSTER>`	*The name of your Substrate cluster.*	`abcd-dev-substrate`
`<SUBSTRATE_NAMESPACE>`	*The name of your Substrate cluster namespace.*	`team-foobar`
`<DOCKER_REPO>`	*The name of your Artifactory docker repository.*	`foobar-docker-dev`
`<HELM_REPO>`	*The name of your Artifactory helm repository.*	`foobar-helm-dev`
`<USERNAME>`	*Your Okta username.*	`john.doe`
`<USERNAME_ALNUM>`	*An alpha-numeric version of your Okta username.*	`johndoe`
`<ARTIFACTORY_ACCESS_TOKEN>`		`abcdef123456`

	The Identity Token associated with your [Artifactory Profile](#).	

Tutorial

01 - Create a sample application

Substrate is optimized for running containerized workloads. Let's create a simple web application built using [nginx](#) that is packaged as a [container image](#).

A - Initialize working directory and files

Create a directory for the application:

```
# Create a working directory for the application
mkdir ~/substrate-example
cd ~/substrate-example

# Initialize as a git repository
git init -b main .
```

We recommend that you use `git` to commit changes as you are creating and updating files. Doing so will make it easier to revert changes as well as use git-diff and/or git-log to review changes made as you progress through the steps in this tutorial.

Populate the application directory with the following files:

Filename	***Description***	***Contents***

`app.js`	*Simple application written in JavaScript and uses njs.*	
`app.envsh`		

```
function home(r) {
  r.headersOut["Content-Type"] = "application/
  r.return(200, JSON.stringify({ message: "OK"
}

function showHeaders(r) {
  r.headersOut["Content-Type"] = "application/
  r.return(200, JSON.stringify({ headers: r.he
}

function showEnvVars(r) {
  r.headersOut["Content-Type"] = "application/
  r.return(200, JSON.stringify({ environmentVa
}

function showAuth(r) {
  r.headersOut["Content-Type"] = "application/

  function _jwtDecode(data) {
    var parts = data
      .split(".")
      .slice(0, 2)
      .map((v) => Buffer.from(v, "base64url").
      .map(JSON.parse);
    return { headers: parts[0], payload: parts
  }

  r.return(200, JSON.stringify(_jwtDecode(r.he
}

export default { home, showHeaders, showEnvVar
```

	Application container entrypoint.	

```
#!/bin/sh

# Load any secrets available from vault
if test -d "/vault/secrets"; then
  find "/vault/secrets" \
    -follow \
    -type f \
    -print \
    | sort -V \
    | while read -r f; do
      . "${f}"
    done
fi
```

`nginx.conf`	*nginx configuration file.*	

```
load_module modules/ngx_http_js_module.so;

events {
}

http {

  js_path "/etc/nginx/njs/";
  js_import app.js;

  server {
    listen 80;

    location / {
      js_content app.home;
    }

    location /health {
      return 200;
```

		`}`
		
		`location /show-headers {`
		`  js_content app.showHeaders;`
		`}`
		
		`location /show-env {`
		`  js_content app.showEnvVars;`
		`}`
		
		`location /show-auth {`
		`  js_content app.showAuth;`
		`}`
		`}`
		`}`

`Dockerfile`	*Instructions to build a container image.*	`# The sample application uses the nginx _Base`
		`# Reference: https://gallery.ecr.aws/nginx/ngi`
		`FROM public.ecr.aws/nginx/nginx:alpine`
		
		`# Add application files to the image`
		`COPY nginx.conf /etc/nginx/nginx.conf`
		`COPY app.js /etc/nginx/njs/app.js`
		
		`# Configure entrypoint`
		`COPY app.envsh /docker-entrypoint.d/app.envsh`
		`RUN chmod +x /docker-entrypoint.d/app.envsh`

B - Build and Run locally

Build the container image using the following command:

```
docker build -t substrate-example .
```

Run the container locally to verify that it works:

```
# Start the container, with the application listening on port 80 (you m
docker run --rm -p 80:80 --name substrate-example substrate-example

# Verify that the container is running
docker ps

# -------------------------

# Use curl to invoke the application

# The / route should return a JSON response '{"message":"OK"}'
curl -LSsf -- http://localhost/ | jq -S .

# The /health route should return an empty 200 OK response
curl -LSsfv -- http://localhost/health

# The /show-headers route should respond with the request headers
curl -LSsf -H "x-foo: bar" -- http://localhost/show-headers | jq -S .

# The /show-env route should respond with the environment variables ava
curl -LSsf -- http://localhost/show-env | jq -S .

# The /show-auth route should respond with the payload of the Authoriza
# Use aop to generate a JWT
curl -LSsf -H "Authorization: Bearer $(aop uas)" -- http://localhost/sh

# -------------------------

# To observe the logs generated by the application, use:
docker logs substrate-example
```

```
# Finally, stop the container
docker stop substrate-example
```

C - Publish to Artifactory

Use the following commands to publish the container image to Artifactory:

```
# Authenticate to Artifactory
docker login -u '<USERNAME>' -p '<ARTIFACTORY_ACCESS_TOKEN>' artifacts.

# Tag the container image
# Remove the single quotes
docker tag substrate-example 'artifacts.ol.epicgames.net/<DOCKER_REPO>/

# Publish the container image
docker push 'artifacts.ol.epicgames.net/<DOCKER_REPO>/substrate-example
```

02 - Create a Helm Chart

In the previous section you packaged the sample application as a container image. To run the container image on Kubernetes, you need to also provide configurations to describe the runtime requirements. For example, how many instances of the application should be running? What are the minimum CPU and memory requirements? What ports need to be enabled? and many more. You can define these requirements within a Helm Chart - a chart is a collection of files that describe a related set of Kubernetes resources. Charts are packages that can be published, consumed and extended (similar to software packages and container images).

A - epic-app

epic-app is an opinionated Helm Chart offered to all teams across Epic to abstract away typical resources and configurations for workloads running on Substrate infrastructure. Create the following files in your application directory to define a new chart that depends on `epic-app` *:*

Filename	***Description***	***Contents***
`deploy/chart/Chart.yaml`	*A YAML file containing information about the chart.*	`##################################` `# HELM CHART FOR `substrate-exampl` `##################################`  `#` `# Reference:` `# * https://helm.sh/docs/topics/ch` `#`  `apiVersion: v2` `name: substrate-example` `version: "1.0.0-<USERNAME_ALNUM>"` `description: Substrate example app`  `# Add a dependency on epic-app (ht` `dependencies:` ` - name: epic-app` ` repository: @substr-helm` ` # epic-app versions and as` ` # https://github.ol.epicgames.` ` version: ">= 2.4.6, < 2.6.0"`  `##################################`

`deploy/chart/values.dev.yaml`	*The* `dev` *configuration values for this chart. You may eventually have a similar* `values.live.yaml` *for* `live`*.*	`##################################`
		`# Configuration values for substra`
		`##################################`
		
		`#`
		`# NOTE:`

		# These are values passed to the e
		# under a top level property that
		# `Chart.yaml`. By convention, the
		#
		epic-app:
		  #
		  # Reference:
		  # * https://github.ol.epicgames.
		  #
		
		  resourceTags:
		  service: "<name of the service
		  owner: "<name of the owner>"
		  contact: "<email of the owner>
		  euid: "<EUID in Epic's service
		
		  containers: # This is a map of c
		  # Application container
		  substrate-example:
		  image:
		  name: "artifacts.ol.epicga
		  tag: "1.0.0-<USERNAME_ALNU
		  ports:
		  - 80
		  environment:
		  LOG_LEVEL: INFO
		
		##################################

Initialize and validate your new chart using the commands below:

```
# List available helm repositories
helm repo list
```

```
# If not listed above, add the "substr-helm" repository hosted on Artif
helm repo add substr-helm https://artifacts.ol.epicgames.net/artifactor

# get/update the epic-app dependency
helm dependency update deploy/chart
```

Running the commands above will add a couple of new files in your application directory:

Filename	***Description***
`deploy/chart/charts/epic-app-*.tgz`	*This is the epic-app chart downloaded to disk. The containing* `charts` *directory will contain downloaded copies of dependencies and should be ignored by version control (via* `.gitignore` *)*
`deploy/chart/requirements.lock`	*This is a dependency lock file that can be used to rebuild the dependencies to an exact version. It is recommended to commit this file to version control.*

03 - Configure authentication and authorization

For applications that rely on https://confluence-epicgames.atlassian.net/wiki/spaces/CE/pages/93488860 (UAS), authentication and authorization can be delegated to the https://confluence-epicgames.atlassian.net/wiki/spaces/CE/pages/93488602 (UAP). UAP is available as a container image and can be deployed alongside your application (i.e., a sidecar-proxy to your upstream application container).

A - Add UAP

UAS connection issues

If you get such errors "https://uas.beef.live.use1a.on.epicgames.com/api/v1/keys": dial tcp i/o timeout

you should ask #cloud-ops-support-ext to authorize your account on UAS service.

To add UAP to your deployment, update the `containers` *section of* `values.dev.yaml` *to include the following snippet:*

Mosaic macros cannot be exported to this format.

B - Configure UAP

UAP, by default, will ensure that all HTTP requests are authenticated and authorized before forwarding the request to your application. However, there are scenarios where unauthenticated requests must be allowed. For the sample application, requests to the /health route should not require authentication to allow for health-checks. UAP allows for this scenario using a configuration file. You can provide the configuration file to the UAP container using a combination of ConfigMaps and Volumes.

To provide a custom route configuration file to UAP, replace or update the UAP container definition and include the additional configurations from the snippet below:

Mosaic macros cannot be exported to this format.

04 - Configure ingress

The application is a service and is intended to be invoked via HTTP requests. Enabling ingress requires multiple steps:

1. *Create a Service – In Kubernetes, a Service is an abstraction which defines a logical set of Pods and a policy by which to access them.*

2. *Configure Ingress – Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster.*
3. *Control network traffic – Define rules that allow network traffic to reach your ingress.*

A - Create a Service

To expose your application as a Service, update `values.dev.yaml` *to include the following snippet:*

When using UAP, ingress must be configured to point to UAP and not the application. Otherwise, the application will receive requests that are not authenticated or authorized.

Mosaic macros cannot be exported to this format.

B - Configure Ingress and Authorize Ingress Traffic

Next, you need to configure ingress. You can use an Application Load Balancer for this. In Kubernetes, an Ingress controller is responsible for fulfilling the ingress. Your Substrate cluster comes with the AWS Load Balancer Controller deployed and can be used to provision an Application Load Balancer.

The Application Load balancer will be configured to listen on port 80 (HTTP) and port 443 (HTTPS), with all traffic on port 80 being redirected (HTTP-301) to port 443. To use an HTTPS listener, you need a valid SSL certificate provisioned in AWS Certificate Manager (ACM). Follow these instructions to create a certificate in ACM.

To add an Application Load Balancer to your application, update `values.dev.yaml` *to include the following snippet. Replace* `<ACM_CERTIFICATE_ARN>` *with the value for your certificate.*

Internal vs Internet Facing Load Balancers

The preferred default method for configuring your load balancer is `internal` so your service can be accessed internally via the Service Network. You would only configure your load balancer as internet-facing if your service would be used to serve external customers. Please be sure to use the correct examples below. They are suffixed by either `- internal` or `- internet facing` depending on your use case.

*If you are using the default preferred method to load balance traffic to an **internal** Application Load Balancer the internal examples below allow traffic via the Service Network, Office, and VPN.*

*If you are using an **internet facing** Application Load Balancer, you will need to specify a prefix list, or inbound CIDR, or a Security Group annotation (you can only choose one method of annotation). In the case of using a prefix list or inbound CIDR ingress annotation, the load balancer controller manages the security group. I.E. the attached security group will include a combination of the listeners annotations and the addresses of the specified prefix list or inbound CIDR. In the case of using a prefix list or security group, these would need to exist in AWS in order to specify the name. Likely these would have been created using Terraform. For more information refer to Managing inbound traffic to your application. For the internet facing example below we will use the prefix list annotation.*

Mosaic macros cannot be exported to this format.

Traffic to the Application Load Balancer is controlled by rules associated with a security group that is attached to it. By default, a new security group is automatically created and attached to the Application Load Balancer.

05 - Using secrets

Applications use sensitive configurations, or secrets, when accessing other backend or downstream resources. For example, a database password or an API key. For a Substrate application, secrets are stored and managed in https://confluence-epicgames.atlassian.net/wiki/spaces/CE/pages/93487474. Your application can then be configured to use these secrets.

Starting from August 2024, it is mandatory that all new Kubernetes clusters use External Secrets Operator (ESO). We strongly recommend using ESO, but the Vault Injector sidecar will still remain available with limited support, bug fixes will be provided if needed to maintain core functionality.

For the purposes of this tutorial we will be using Option 1 below to use ESO for secrets. Reference Using External Secrets Operator (ESO) in epic-app to inject secrets for documentation on using ESO.

There are currently 2 ways to use secrets. Both options are presented below. You only have to choose 1 option.

Option 1: *Publishing secrets in Substrate Vault, then using External Secrets Operator in epic-app to consume those secrets.*

Option 2: *Publishing and sharing secrets with Substrate Vault and SSSM, then using vault injector in epic-app to consume those secrets*

Option 1A - Publish and share secrets with Substrate Vault

Lets assume that the sample application needs 2 secrets to access a database. This will be a username and password. You would first need to create the secrets in substrate vault. The instructions for creating a secret in Substrate Vault for use with ESO in epic app can be found in the

document called [Using External Secrets Operator (ESO) in epic-app to inject secrets](#).

Option 1B - Access and use secrets in your application using ESO

For applications to access secrets, you can enable externalSecrets in epic-app. When the application is deployed, ESO will make the secrets available to the containers in the pod. In this example

Option 2A - Publish and share secrets with Substrate Vault and SSSM

Let's assume that the sample application needs a secret named `foo` *with a value* `bar`*. Use the [Substrate vault instructions](#) to publish the secret and share it with your Substrate cluster. The instructions below assumes that you have a secret published with a path* `secret/path/to/foo`*, make sure to update it to reflect the path used in Vault.*

Option 2B - Access and use secrets in your application using vault injector

For applications to access secrets, you can use the [HashiCorp Vault Agent Injector](#). The Vault Agent Injector is already installed in a Substrate cluster, you need to use annotations to enable it for your application. When the application is deployed, the Vault Agent Injector will fetch the secrets configured via annotations and make them available as files to your application. In this example, a file called `/vault/secrets/foo` *will be available. The application container entrypoint (* `app.envsh` *) reads the file and passes on the secrets as environment variables to the application process.*

To access secrets within the application, update `values.dev.yaml` *to include the following snippet:*

Mosaic macros cannot be exported to this format.

06 - Deploy the application

The application can be deployed using helm install.

This requires an authenticated terminal session. Make sure you have authenticated with the account you intend to deploy to as well as the correct namespace.

```
# Examine the manifests that epic-app and helm generate for the applica
helm template 'substrate-example-<USERNAME_ALNUM>' ./deploy/chart -f ./

# Deploy the application
helm install 'substrate-example-<USERNAME_ALNUM>' ./deploy/chart -f ./d

# Inspect the deployment
helm status 'substrate-example-<USERNAME_ALNUM>' --show-resources -n '<

# Review the list of resources created
helm get all 'substrate-example-<USERNAME_ALNUM>' -n '<SUBSTRATE_NAMESP
```

07 - Verify deployment

A - Access the deployed application

For internal Service Network use cases browse to `https://substrate-example-<USERNAME_ALNUM>.<SUBSTRATE_ACCOUNT>.dev.use1a.internal.epicgames.com`

For Internet facing use cases browse to `https://substrate-example-<USERNAME_ALNUM>.<SUBSTRATE_ACCOUNT>.dev.use1a.internal.epicgames.com`

Review the responses from the following routes (they should be similar to what was observed locally):

- `/health`
- `/show-headers`
- `/show-env`
- `/show-auth`

B - View logs using `kubectl`

To view logs from your application on the terminal you can use `kubectl logs` *:*

```
# Get the name of the pod
kubectl get pods -n '<SUBSTRATE_NAMESPACE>' -o jsonpath='{.items[0].met

# Get logs generated since the last hour for the application container
kubectl logs --since=1h -n '<SUBSTRATE_NAMESPACE>' -c substrate-example
```

C - View logs using Grafana

Container logs are, by default, forwarded to Loki and visualized in Grafana. To view the logs in Grafana, you can:

1. *Go to Loki User Guide*
2. *Follow the instructions under* ***Finding Your Logs*** *and* ***Querying Your Logs***

D - View metrics using Grafana

Substrate cluster metrics, by default, are forwarded to Chronosphere and visualized in Grafana. To view available metrics for your application, you can:

1. *Go to the Chronosphere User Guide*
2. *Follow the instructions under* ***Finding Your Metrics***