Kubernetes-compatible API for EVE Controller

This document reviews the standard functionality of an EVE controller, and how those might be implemented using Kubernetes. It is not tied to a particular implementation of EVE controller, and as such may not cover all use cases. However, it should be extensible enough for any particular controller to adopt this and then add the specific needs.

As of this writing, there are two known implementations of controllers: Open Source Adam, and commercial Zedcloud by ZEDEDA.

Adam is a reference OSS EVE controller. It has a very limited, but simple, custom REST API, described here as its `adminHandler`. Specifically, it enables:

- Adding, deleting, updating and reading onboard certificates
- Adding, deleting, updating and reading an edge device
- Getting and setting global options
- Getting and setting device specific options
- Reading a device's logs, information, metrics and certificates
- Reading and writing a device's configuration as a single json blob, per the EVE API.

It is important to note that Adam provides no granular control over a device; it only knows how to return the entire device config as json, or receive an entirely new device config as json.

Zedcloud is an enterprise-class commercial solution, offered as a fully-managed SaaS solution, along with device support and professional services, from ZEDEDA. In addition to management services, and all of the abilities that Adam provides, ZEDEDA offers:

- Multiple forms of authentication/authorization as well as enterprise SSO
- An advanced UI
- Reading and writing individual components of a device's configuration, as well as validations, which, in turn, are composed by Zedcloud into a device's configuration.
- Libraries of components, applications and images
- Predefined templates
- Many more features

The goal of this document is to provide an API design that:

- Is compatible with Kubernetes.
- Supports all of current Adam functionality.
- Enables getting and setting of individual elements within the EVE API, thus enabling Adam to combine them into a full json device configuration.
- Is extensible, so that commercial controllers, such as Zedcloud, could use the API as a basis for their own API which is 100% compatible with this API, yet supports their rich additional features.

On the basis of the above:

1. Anything that is in Adam or an element of the EVE API will be covered here.
2. Anything that is not in the above will not be covered here.
3. The method of extensions will be covered here.

Note that elements of the EVE API that are intended entirely for control between the controller and device, will not have a Kubernetes-compatible component, as those are not intended to be exposed to the end-user.

Why *would* we want to standardize on a Kubernetes-compatible API? The Kubernetes API comes with several advantages:

- It provides a standardized interface.
- It enables controller designers to leverage existing reliable CLI and SDK tooling with which large numbers of technologists and IT departments are familiar.
- It allows controller designers to replace entire parts of their code with an off-the-shelf component that is mature, reliable, production-tested, and capable.
- It enables controller providers to plug into almost any Kubernetes ecosystem component.
- It eases locating skilled staff and shortens their learning curve.

- Globally available CLI and clients in multiple languages, supported by a large distributed team of people, with a very large existing installed base
- Ability of customers to define all EVE resources as code, in human-readable template files, and stored optionally in version control
- Easy integration with any system that works with Kubernetes, e.g. CI systems or monitoring systems
- Freely available server-side tooling that supports REST endpoints, validation, authentication, distributed highly-available processing, eventual reconciliation, role-based access control based on API paths, including individual resource-level control; all standardized and used in production, supported by a large number of engineers globally.

Custom Resources vs Native Kubernetes

Kubernetes is an API and a system for declaring workloads and scheduling them on nodes.

EVE includes an API and a system for declaring workloads and scheduling them on nodes.

The basic concepts of EVE and Kubernetes are very similar. In addition to those concepts, EVE provides node management, and Kubernetes provides higher-order scheduling options.

From that perspective, it is possible to use native Kubernetes resources:

- Pod for AppInstance workload
- Node for Edge Device

However, doing so brings minimal advantages and some real disadvantages.

1. Kubernetes itself will try to schedule the AppInstance workloads onto nodes, as it thinks they are normal Kubernetes pods. We can prevent that using Pod node affinity, but it requires additional steps.
2. Kubernetes itself will try to schedule other workloads onto the Edge Devices, as it thinks they are normal Kubernetes nodes. We can prevent that using node taints and pod tolerations, but it requires additional steps.
3. Kubernetes expects nodes to join in the normal fashion, using either a node certificate signed by an appropriate CA, or a one-time registration token. These are similar, but not identical, to the process used by EVE. We can get around that by registering the nodes separately, but it is going against the Kubernetes grain and can lead to issues.
4. Kubernetes expects nodes to be accessible and in regular communication with the control plane kube-apiserver, while EVE expects Edge Devices to be out of touch, at times for extended periods. This will cause Kubernetes to mark these nodes as unavailable for scheduling, and eventually remove them. The apiserver-node API is built with cloud assumptions of regular and reliable connectivity.
5. Kubernetes expects nodes to speak the undocumented kubelet API. This includes the internal elements, but also receiving pod specifications. Using this would require implementing the kubelet API on the Edge Device or somehow convincing Kubernetes that it was.
6. Overloading the Node and Pod resources forces the cluster to be in a confusing state for users. This would make it difficult to deploy an EVE controller onto a regular Kubernetes cluster, creating barriers to adoption.

For the above reasons, we implement the entire EVE API in custom resources (CRDs) and do not use the native APIs.

For the purposes of edification, we include a table comparing the two approaches here, and include what a native option looks like in an appendix.

The following table summarizes all resources. Original resources - either from the higher-level Adam abstraction or from the native EVE config - are in black, reused native Kubernetes resources are in blue, custom resources are in green. Where a native resource inherently works well, even the custom resource column will use the native resource, marked in blue.

EVE resource	Native resource	Custom resources
Onboarding		OnboardCertificate OnboardCertificateAuthority DeviceCertificateAuthority
Device serial		Device
Device certificate		Device property
Edge Device	Node	Device
Global options	ConfigMap	ConfigMap
Edge Device options	Annotations	Device Annotations
Edge Application Instance	Pod	Application
Base OS		Device Annotation
Device Config		Device properties
Network Config		NetworkConfig
Device Network		DeviceNetwork
Application Network		Annotations
Volume	Persistent Volume	Volume
Data Store	StorageClass	StorageClass
Content Tree	Image	Image
Scheduling (controller)	Deployment DaemonSet	ApplicationDeployment ApplicationDaemonSet

Items that require special treatment:

- Device Config - the entire json that is composed by a controller and sent to a device. The core controller reads the other elements and creates this resource (which can be modified by the end-user). A separate controller reads it and applies it to the device via the API.
- Read-only elements from the Edge Device - log, info, metrics - do not receive their own resources. Kubernetes resources are designed to be created/modified, remain in relatively stable state and are of small size. They are stored in the Kubernetes backing store, by default etcd, and do not fit large streaming elements like logs or metrics. When Kubernetes itself enables reading log information from a pod, it does so by leveraging an element of the control-plane-to-kubelet API and open real-time streaming of logs.

The streaming elements - log, info, metrics - will stream to the controller according to the current EVE API design, are stored by the controller in its own datastore, and controller clients open a streaming channel to the controller. There currently is no native construct within the Kubernetes API that knows how to work with this flow, nor do CRDs have such elements.

Since Kubernetes has no native construct for this, we create a special “streaming” pod and service. This has a Web server and a Service in front of it, exposing endpoints. We use RBAC to enable access to this pod.

This pod exposes endpoints:

- `/edgenode/<uuid>/metrics`
- `/edgenode/<uuid>/info`
- `/edgenode/<uuid>/log`

The edgenode uuid must match one that is provided by the CRDs. Headers determine if the data should be streaming continuously or only show existing data, while query parameters determine start and end times and filter parameters.

Design

Although we are creating entirely new custom resources, we follow these rules:

- Follow native Kubernetes semantics as much as possible.
- Where concepts closely parallel those that exist in native Kubernetes resources, we adopt the nomenclature and semantics as much as possible into our custom resources.
- Where standard annotations already exist to describe a concept, use those annotations.
- Where a native resource is 100% reusable, without mismatches, reuse it.

Extensibility

Various controllers may choose to implement resources in different ways, including adding features to those resources that are not part of the primary open source specification here.

The mechanism for extensibility in all cases is annotations and additional CRDs.

Annotations

If a controller needs to add functionality to a resource defined in this specification, it should add an annotation that matches its requirement. The annotation’s prefix must match the controller provider’s unique domain. The prefix eve.lfedge.org is reserved for standard annotations that are used as part of this specification.

As each controller implementation must implement the control loops for the various resources, including the standard ones, it is up to the specific controller’s implementation of the control loop to manage the additional annotations.

Other controllers must ignore such annotations.

For example, if a fictitious commercial controller provider named edgesmart, with domain `edgesmart.tv`, wishes to add features to the DeviceNetwork resource, it adds annotations using that domain.

```
apiVersion: "eve.lfedge.org/v1beta1"
kind: DeviceNetwork
metadata:
  name: default-ipv4
  namespace: enterprise1
  annotations:
    ctrl.edgesmart.tv/network-control: 42
spec:
  networkConfig: default-ipv4
```

Custom Resources

Controllers are not limited to the CRDs defined in this specification. Any controller may implement additional CRDs.

In doing so they must not use the apiVersion prefix defined in this specification, but rather must use one that matches a domain owned by them.

For example, if a fictitious commercial controller provider named edgesmart, with domain `edgesmart.tv`, wishes to have a new type of device security configuration, they can create a CRD for it. Notice the `apiVersion` field.

```
apiVersion: "ctrl.edgesmart.tv/v1beta1"
kind: DeviceSecurityConfig
metadata:
  name: device-security-high
  namespace: enterprise1
spec:

...
```

Resource Versioning

Every resource type in Kubernetes has a unique identifier of group/version/kind. See the following references:

- https://kubernetes.io/docs/reference/using-api/api-concepts/#resource-uris
- https://kubernetes.io/docs/concepts/overview/kubernetes-api/#api-groups-and-versioning
- https://kubernetes.io/docs/reference/using-api/#api-groups.

Using the above example:

```
apiVersion: "ctrl.edgesmart.tv/v1beta1"
kind: DeviceSecurityConfig
metadata:
  name: device-security-high
  namespace: enterprise1
spec:

...
```

The resource is:

- group: ctrl.edgesmart.tv
- version: v1beta1
- kind: DeviceSecurityConfig

For all cases where we use native Kubernetes resources, we follow the native resource's group/version/kind scheme.

For all CRDs created here:

- Versions should follow the normal Kubernetes naming scheme of `v1alpha1`, `v1alpha2`, … `v1beta1`, `v1beta2`, … `v1`, `v2alpha1`, … `v2`, etc.
- Group must be `eve.lfedge.org`
- Group for resources defined by third-party extensions, such as commercial controllers, must use their own domain and **not** reuse `eve.lfedge.org`

Unique Resource Identification

Every instance of a resource type is uniquely identified either by name or, for namespace-scoped resources, by namespace/name.

The EVE API, on the other hand, uses UUIDs, normally generated by the controller, to uniquely identify and link resources.

The API defined here follows the Kubernetes convention in using group/version/kind for identifying resource types, and name, with optional namespace, for uniquely identifying instances of resources.

It is up to the controller to create UUIDs, link them to specific name/namespace of resources internally, and map them to UUIDs sent to edge nodes via the EVE API.

Authentication

Authentication leverages normal Kubernetes authentication and authorization paths, whether via static users, client certificates or SSO via OIDC.

Edge Device

The Device is very similar to the native Node, except that specification items that would be loaded into annotations are made part of the core spec.

```
apiVersion: eve.lfedge.org/v1beta1
kind: Device
metadata:
  labels:
    beta.kubernetes.io/arch: amd64
    beta.kubernetes.io/instance-type: eve
    beta.kubernetes.io/os: linux
    kubernetes.io/arch: amd64
    kubernetes.io/hostname: eve-device.lab1
    kubernetes.io/os: eve
    node.kubernetes.io/instance-type: eve
  annotations:
    eve.lfedge.org/node-type: virtual
    eve.lfedge.org/location: "texas/usa"
    eve.lfedge.org/activate: true

  name: eve-device.lab1
  Namespace: enterprise1
spec:

  eve-os-version: 8.10.0-kvm-amd64
  certificate:
TFMwdExTMUNSVWRKVGlCRFJWSlVTVVpKUTBGVVJTMHRMUzB0Q2sxSlNVTTFla05EUVdNclowRjNTVUpCWjBsQ1FWUkJUa0puYTNGb2EybEhPWGN3
UWtGUmMwWkJSRUZXVFZKTmQwVlJXVVJXVVZGRVJYZHdjbVJYU213S1kyMDFiR1JIVm5wTlFqUllSRlJKZVUxRWEzZE5WRUV6VGtSTmVrOVdiMWhF
VkUxNVRVUm5lVTlVUVROT1JFMTZUMVp2ZDBaVVJWUk5Ra1ZIUVRGVlJRcEJlRTFMWVROV2FWcFlTblZhV0ZKc1kzcERRMEZUU1hkRVVWbEtTMjlh
U1doMlkwNUJVVVZDUWxGQlJHZG5SVkJCUkVORFFWRnZRMmRuUlVKQlRDdG5DbWgxZVVzMEswZ3dSRGRXUVZWUUwwdEdWbFZRUkhRcllrTnhiMEZT
YkRNeVlteEhaV0pxYlZJeGRuZEdZalJYYlhkSVdEWmhOVFZMU0hWbmNGb3lUbThLVVVjNVQzcEhkMjVNWmpKeGRGSlBZbkpvTHk5bFkxbFBaRmxD
VjNWRWIySk1lbkYyYW05WllWZFlZVnB1TVhKVllYY3dSekZuYUhab1ZYUTVSbE56YlFwd1NHeHNXVzh3YzJOU1ZGRnhkMWRyUXpOaVprMXpkWFpY
Ukd0cGFsRlVjMnhzYTFBNVdXZENjMFJZYlZBNU4zaEdkR0pXT0ZodFVtOXpkREZhUTJOS0Nua3hiRUpSU0RGM1R6TlNNR2h4YkUxUGNGZG9ZVzlI
TUhObldqWldhR3g1YW5OVk0xZHFiWGhFVm1abE5tczNObmx0WmpCM1MxZGhVWHBoUjFGdlpGZ0tlQ3RQWTJSaWJHcDNlVXh4VWs1U2NWaFljRGRI
T1VOWVlXRkVPWGxYT1cxTlZ6UkRiVFI2VDFweU1VZG5SMDF3YkZSMGFWVTBiWFJSY1dKUmNYTjFWQXBaVVVRekwwaEVTVGg0TmpSMVZWQk5UbXRG
UTBGM1JVRkJZVTVEVFVWQmQwUm5XVVJXVWpCUVFWRklMMEpCVVVSQlowdHJUVUU0UjBFeFZXUkZkMFZDQ2k5M1VVWk5RVTFDUVdZNGQwaFJXVVJX
VWpCUFFrSlpSVVpJVWt0SVdHRXZTRFpPUlZaUFIzUjJSV00xVkRWM1UybHVkMGhOUVRCSFExTnhSMU5KWWpNS1JGRkZRa04zVlVGQk5FbENRVkZD
VDNKeWQzSllkbVo1U2xWUFIxaFZiMXB6ZEdoTFUxZ3ZhRGRtZVdGa01WSnNiV3RQYzBoMldtMDNORUpNZURGNVFncHhTREZMU1hjMlozQktWMUpH
UkVKc01GUlBPWFYxWWtvemNHMXRaMWx1WW5wUFRqRXJVVFJPV1dscWNsYzVXVEpwTXk5bWNDdEpTWEZFV0VwTFRYZHpDbFpsYjNWeWFHZFRTVkEx
U0RaSFJVMHZUalJLVjBJeU4xVTFXVWswVlZKQlNEWjFkRU14Vm5sblZVOUNUMHN3Tm5wQmVXbEJabWROVERseE0wVkZXRFVLVlZwVmVUQnVSRUV2
WVhGUFZqVkdZa3hMTmxWek1tWnRNREpXUW05SVZFNTRURXBWYlV4b2FFSTJWVkZQUkRKSlpGRTJiRWhhUXpNdk9HVk5PVTR3T0FvcmNWSklObXM1
UjI5dGRHbElUM2R5WkVwd1dqTklVa3huYVdoV1ZrUjZhbVIxWkRoMFVWUnRhVGwyUTBsWVFXeFJiek5vZWxabVJUbERhVTVEVEVSRENtWkZia2xV
WTA1Q2VXbEtRbU42TldaVVVuZFhWa1JKWlc5QlYzTkZabTVJY1ROSGVRb3RMUzB0TFVWT1JDQkRSVkpVU1VaSlEwRlVSUzB0TFMwdENnPT0= #
base64 encoded
  serial: "6654abbcc44"
  onboard:
TFMwdExTMUNSVWRKVGlCRFJWSlVTVVpKUTBGVVJTMHRMUzB0Q2sxSlNVTTFla05EUVdNclowRjNTVUpCWjBsQ1FWUkJUa0puYTNGb2EybEhPWGN3
UWtGUmMwWkJSRUZXVFZKTmQwVlJXVVJXVVZGRVJYZHdjbVJYU213S1kyMDFiR1JIVm5wTlFqUllSRlJKZVUxRWEzZE5WRUV6VGtSTmVrOVdiMWhF
VkUxNVRVUm5lVTlVUVROT1JFMTZUMVp2ZDBaVVJWUk5Ra1ZIUVRGVlJRcEJlRTFMWVROV2FWcFlTblZhV0ZKc1kzcERRMEZUU1hkRVVWbEtTMjlh
U1doMlkwNUJVVVZDUWxGQlJHZG5SVkJCUkVORFFWRnZRMmRuUlVKQlRDdG5DbWgxZVVzMEswZ3dSRGRXUVZWUUwwdEdWbFZRUkhRcllrTnhiMEZT
YkRNeVlteEhaV0pxYlZJeGRuZEdZalJYYlhkSVdEWmhOVFZMU0hWbmNGb3lUbThLVVVjNVQzcEhkMjVNWmpKeGRGSlBZbkpvTHk5bFkxbFBaRmxD
VjNWRWIySk1lbkYyYW05WllWZFlZVnB1TVhKVllYY3dSekZuYUhab1ZYUTVSbE56YlFwd1NHeHNXVzh3YzJOU1ZGRnhkMWRyUXpOaVprMXpkWFpY
Ukd0cGFsRlVjMnhzYTFBNVdXZENjMFJZYlZBNU4zaEdkR0pXT0ZodFVtOXpkREZhUTJOS0Nua3hiRUpSU0RGM1R6TlNNR2h4YkUxUGNGZG9ZVzlI
TUhObldqWldhR3g1YW5OVk0xZHFiWGhFVm1abE5tczNObmx0WmpCM1MxZGhVWHBoUjFGdlpGZ0tlQ3RQWTJSaWJHcDNlVXh4VWs1U2NWaFljRGRI
T1VOWVlXRkVPWGxYT1cxTlZ6UkRiVFI2VDFweU1VZG5SMDF3YkZSMGFWVTBiWFJSY1dKUmNYTjFWQXBaVVVRekwwaEVTVGg0TmpSMVZWQk5UbXRG
UTBGM1JVRkJZVTVEVFVWQmQwUm5XVVJXVWpCUVFWRklMMEpCVVVSQlowdHJUVUU0UjBFeFZXUkZkMFZDQ2k5M1VVWk5RVTFDUVdZNGQwaFJXVVJX
VWpCUFFrSlpSVVpJVWt0SVdHRXZTRFpPUlZaUFIzUjJSV00xVkRWM1UybHVkMGhOUVRCSFExTnhSMU5KWWpNS1JGRkZRa04zVlVGQk5FbENRVkZD
VDNKeWQzSllkbVo1U2xWUFIxaFZiMXB6ZEdoTFUxZ3ZhRGRtZVdGa01WSnNiV3RQYzBoMldtMDNORUpNZURGNVFncHhTREZMU1hjMlozQktWMUpH
UkVKc01GUlBPWFYxWWtvemNHMXRaMWx1WW5wUFRqRXJVVFJPV1dscWNsYzVXVEpwTXk5bWNDdEpTWEZFV0VwTFRYZHpDbFpsYjNWeWFHZFRTVkEx
U0RaSFJVMHZUalJLVjBJeU4xVTFXVWswVlZKQlNEWjFkRU14Vm5sblZVOUNUMHN3Tm5wQmVXbEJabWROVERseE0wVkZXRFVLVlZwVmVUQnVSRUV2
WVhGUFZqVkdZa3hMTmxWek1tWnRNREpXUW05SVZFNTRURXBWYlV4b2FFSTJWVkZQUkRKSlpGRTJiRWhhUXpNdk9HVk5PVTR3T0FvcmNWSklObXM1
```

```
UjI5dGRHbElUM2R5WkVwd1dqTklVa3huYVdoV1ZrUjZhbVIxWkRoMFVWUnRhVGwyUTBsWVFXeFJiek5vZWxabVJUbERhVTVEVEVSRENtWkZia2xV
WTA1Q2VXbEtRbU42TldaVVVuZFhWa1JKWlc5QlYzTkZabTVJY1ROSGVRb3RMUzB0TFVWT1JDQkRSVkpVU1VaSlEwRlVSUzB0TFMwdENnPT0= #
base64 encoded
status:
  eve-os-version: 6.12.2

  uuid: EC232B65-602A-F2A9-287B-5D95721116E6
  addresses:
  - address: 172.19.0.3
    type: InternalIP
  - address: k3d-k3s-default-server-0
    type: Hostname
  allocatable:
    cpu: "4"
    ephemeral-storage: "296591664715"
    hugepages-1Gi: "0"
    hugepages-2Mi: "0"
    memory: 16235544Ki
    pods: "110"
  capacity:
    cpu: "4"
    ephemeral-storage: 304884524Ki
    hugepages-1Gi: "0"
    hugepages-2Mi: "0"
    memory: 16235544Ki
    pods: "110"
  conditions:
  - lastHeartbeatTime: "2021-11-23T12:57:09Z"
    lastTransitionTime: "2021-10-10T10:33:38Z"
    message: kubelet is posting ready status
    reason: KubeletReady
    status: "True"
    type: Ready
  nodeInfo:
    architecture: amd64
    bootID: dc703fd4-543b-4801-96be-4d6d29afb41e
    containerRuntimeVersion: containerd://1.4.9
    kernelVersion: 5.10.1
    machineID: ""
    operatingSystem: eve
    osImage: eve
    systemUUID: EC232B65-602A-F2A9-287B-5D95721116E6
```

A Device can be created in one of two ways:

- User: A user can create a Device resource by adding the resource. It can include a serial, onboard certificate and/or device certificate. When the device attempts to connect to the controller, it is the controller's onboarding policy that determines how it authenticates, using the device certificate, onboard certificate, serial, CA on certificates, or any combination of the above.
- Device: A device can self-register, automatically causing a Device resource to be created, if and only if the controller policy allows it. In this case, either it must be one of:
    - Open to all self-registration.
    - Allows a device certificate signed by a known CA to self-register.
    - Allows an onboard certificate signed by a known CA to self-register.
    - Allows a known onboard certificate to self-register.

The resources that enable the following are:

- `OnboardCertificate`
- `OnboardCA`
- `DeviceCA`

These are all namespaced.

Note that if a user desires to allow a particular onboard certificate and serial combination, and the controller supports it, they simply create the Device resource, providing the onboard certificate and serial as part of the spec.

Onboard Certificate

Any device presenting this onboard certificate can self-register.

```
apiVersion: "eve.lfedge.org/v1beta1"
kind: OnboardCertificate
metadata:
  name: onboard-cert-25
  namespace: enterprise1
spec:
  certificate:
TFMwdExTMUNSVWRKVGlCRFJWSlVTVVpKUTBGVVJTMHRMUzB0Q2sxSlNVTTFla05EUVdNclowRjNTVUpCWjBsQ1FWUkJUa0puYTNGb2EybEhPWGN3
UWtGUmMwWkJSRUZXVFZKTmQwVlJXVVJXVVZGRVJYZHdjbVJYU213S1kyMDFiR1JIVm5wTlFqUllSRlJKZVUxRWEzZE5WRUV6VGtSTmVrOVdiMWhF
VkUxNVRVUm5lVTlVUVROT1JFMTZUMVp2ZDBaVVJWUk5Ra1ZIUVRGVlJRcEJlRTFMWVROV2FWcFlTblZhV0ZKc1kzcERRMEZUU1hkRVVWbEtTMjlh
U1doMlkwNUJVVVZDUWxGQlJHZG5SVkJCUkVORFFWRnZRMmRuUlVKQlRDdG5DbWgxZVVzMEswZ3dSRGRXUVZWUUwwdEdWbFZRUkhRcllrTnhiMEZT
YkRNeVlteEhaV0pxYlZJeGRuZEdZalJYYlhkSVdEWmhOVFZMU0hWbmNGb3lUbThLVVVjNVQzcEhkMjVNWmpKeGRGSlBZbkpvTHk5bFkxbFBaRmxD
VjNWRWIySk1lbkYyYW05WllWZFlZVnB1TVhKVllYY3dSekZuYUhab1ZYUTVSbE56YlFwd1NHeHNXVzh3YzJOU1ZGRnhkMWRyUXpOaVprMXpkWFpY
Ukd0cGFsRlVjMnhzYTFBNVdXZENjMFJZYlZBNU4zaEdkR0pXT0ZodFVtOXpkREZhUTJOS0Nua3hiRUpSU0RGM1R6TlNNR2h4YkUxUGNGZG9ZVzlI
TUhObldqWldhR3g1YW5OVk0xZHFiWGhFVm1abE5tczNObmx0WmpCM1MxZGhVWHBoUjFGdlpGZ0tlQ3RQWTJSaWJHcDNlVXh4VWs1U2NWaFljRGRI
T1VOWVlXRkVPWGxYT1cxTlZ6UkRiVFI2VDFweU1VZG5SMDF3YkZSMGFWVTBiWFJSY1dKUmNYTjFWQXBaVVVRekwwaEVTVGg0TmpSMVZWQk5UbXRG
UTBGM1JVRkJZVTVEVFVWQmQwUm5XVVJXVWpCUVFWRklMMEpCVVVSQlowdHJUVUU0UjBFeFZXUkZkMFZDQ2k5M1VVWk5RVTFDUVdZNGQwaFJXVVJX
VWpCUFFrSlpSVVpJVWt0SVdHRXZTRFpPUlZaUFIzUjJSV00xVkRWM1UybHVkMGhOUVRCSFExTnhSMU5KWWpNS1JGRkZRa04zVlVGQk5FbENRVkZD
VDNKeWQzSllkbVo1U2xWUFIxaFZiMXB6ZEdoTFUxZ3ZhRGRtZVdGa01WSnNiV3RQYzBoMldtMDNORUpNZURGNVFncHhTREZMU1hjMlozQktWMUpH
UkVKc01GUlBPWFYxWWtvemNHMXRaMWx1WW5wUFRqRXJVVFJPV1dscWNsYzVXVEpwTXk5bWNDdEpTWEZFV0VwTFRYZHpDbFpsYjNWeWFHZFRTVkEx
U0RaSFJVMHZUalJLVjBJeU4xVTFXVWswVlZKQlNEWjFkRU14Vm5sblZVOUNUMHN3Tm5wQmVXbEJabWROVERseE0wVkZXRFVLVlZwVmVUQnVSRUV2
WVhGUFZqVkdZa3hMTmxWek1tWnRNREpXUW05SVZFNTRURXBWYlV4b2FFSTJWVkZQUkRKSlpGRTJiRWhhUXpNdk9HVk5PVTR3T0FvcmNWSklObXM1
UjI5dGRHbElUM2R5WkVwd1dqTklVa3huYVdoV1ZrUjZhbVIxWkRoMFVWUnRhVGwyUTBsWVFXeFJiek5vZWxabVJUbERhVTVEVEVSRENtWkZia2xV
WTA1Q2VXbEtRbU42TldaVVVuZFhWa1JKWlc5QlYzTkZabTVJY1ROSGVRb3RMUzB0TFVWT1JDQkRSVkpVU1VaSlEwRlVSUzB0TFMwdENnPT0= #
base64 encoded
```

Onboard CA

Any device presenting a certificate signed by this CA can self-register.

```
apiVersion: "eve.lfedge.org/v1beta1"
kind: OnboardCertificateAuthority
metadata:
  name: onboard-ceriticate-authority-13
  namespace: enterprise1
spec:
  certificate:
TFMwdExTMUNSVWRKVGlCRFJWSlVTVVpKUTBGVVJTMHRMUzB0Q2sxSlNVTTFla05EUVdNclowRjNTVUpCWjBsQ1FWUkJUa0puYTNGb2EybEhPWGN3
UWtGUmMwWkJSRUZXVFZKTmQwVlJXVVJXVVZGRVJYZHdjbVJYU213S1kyMDFiR1JIVm5wTlFqUllSRlJKZVUxRWEzZE5WRUV6VGtSTmVrOVdiMWhF
VkUxNVRVUm5lVTlVUVROT1JFMTZUMVp2ZDBaVVJWUk5Ra1ZIUVRGVlJRcEJlRTFMWVROV2FWcFlTblZhV0ZKc1kzcERRMEZUU1hkRVVWbEtTMjlh
U1doMlkwNUJVVVZDUWxGQlJHZG5SVkJCUkVORFFWRnZRMmRuUlVKQlRDdG5DbWgxZVVzMEswZ3dSRGRXUVZWUUwwdEdWbFZRUkhRcllrTnhiMEZT
YkRNeVlteEhaV0pxYlZJeGRuZEdZalJYYlhkSVdEWmhOVFZMU0hWbmNGb3lUbThLVVVjNVQzcEhkMjVNWmpKeGRGSlBZbkpvTHk5bFkxbFBaRmxD
VjNWRWIySk1lbkYyYW05WllWZFlZVnB1TVhKVllYY3dSekZuYUhab1ZYUTVSbE56YlFwd1NHeHNXVzh3YzJOU1ZGRnhkMWRyUXpOaVprMXpkWFpY
Ukd0cGFsRlVjMnhzYTFBNVdXZENjMFJZYlZBNU4zaEdkR0pXT0ZodFVtOXpkREZhUTJOS0Nua3hiRUpSU0RGM1R6TlNNR2h4YkUxUGNGZG9ZVzlI
TUhObldqWldhR3g1YW5OVk0xZHFiWGhFVm1abE5tczNObmx0WmpCM1MxZGhVWHBoUjFGdlpGZ0tlQ3RQWTJSaWJHcDNlVXh4VWs1U2NWaFljRGRI
T1VOWVlXRkVPWGxYT1cxTlZ6UkRiVFI2VDFweU1VZG5SMDF3YkZSMGFWVTBiWFJSY1dKUmNYTjFWQXBaVVVRekwwaEVTVGg0TmpSMVZWQk5UbXRG
UTBGM1JVRkJZVTVEVFVWQmQwUm5XVVJXVWpCUVFWRklMMEpCVVVSQlowdHJUVUU0UjBFeFZXUkZkMFZDQ2k5M1VVWk5RVTFDUVdZNGQwaFJXVVJX
VWpCUFFrSlpSVVpJVWt0SVdHRXZTRFpPUlZaUFIzUjJSV00xVkRWM1UybHVkMGhOUVRCSFExTnhSMU5KWWpNS1JGRkZRa04zVlVGQk5FbENRVkZD
VDNKeWQzSllkbVo1U2xWUFIxaFZiMXB6ZEdoTFUxZ3ZhRGRtZVdGa01WSnNiV3RQYzBoMldtMDNORUpNZURGNVFncHhTREZMU1hjMlozQktWMUpH
UkVKc01GUlBPWFYxWWtvemNHMXRaMWx1WW5wUFRqRXJVVFJPV1dscWNsYzVXVEpwTXk5bWNDdEpTWEZFV0VwTFRYZHpDbFpsYjNWeWFHZFRTVkEx
U0RaSFJVMHZUalJLVjBJeU4xVTFXVWswVlZKQlNEWjFkRU14Vm5sblZVOUNUMHN3Tm5wQmVXbEJabWROVERseE0wVkZXRFVLVlZwVmVUQnVSRUV2
WVhGUFZqVkdZa3hMTmxWek1tWnRNREpXUW05SVZFNTRURXBWYlV4b2FFSTJWVkZQUkRKSlpGRTJiRWhhUXpNdk9HVk5PVTR3T0FvcmNWSklObXM1
UjI5dGRHbElUM2R5WkVwd1dqTklVa3huYVdoV1ZrUjZhbVIxWkRoMFVWUnRhVGwyUTBsWVFXeFJiek5vZWxabVJUbERhVTVEVEVSRENtWkZia2xV
WTA1Q2VXbEtRbU42TldaVVVuZFhWa1JKWlc5QlYzTkZabTVJY1ROSGVRb3RMUzB0TFVWT1JDQkRSVkpVU1VaSlEwRlVSUzB0TFMwdENnPT0= #
base64 encoded
```

Device CA

Any device presenting a device certificate signed by this CA can self-register.

```
apiVersion: "eve.lfedge.org/v1beta1"
kind: DeviceCertificateAuthority
metadata:
  name: device-certificate-authority-16
  namespace: enterprise1
spec:
  certificate:
TFMwdExTMUNSVWRKVGlCRFJWSlVTVVpKUTBGVVJTMHRMUzB0Q2sxSlNVTTFla05EUVdNclowRjNTVUpCWjBsQ1FWUkJUa0puYTNGb2EybEhPWGN3
UWtGUmMwWkJSRUZXVFZKTmQwVlJXVVJXVVZGRVJYZHdjbVJYU213S1kyMDFiR1JIVm5wTlFqUllSRlJKZVUxRWEzZE5WRUV6VGtSTmVrOVdiMWhF
VkUxNVRVUm5lVTlVUVROT1JFMTZUMVp2ZDBaVVJWUk5Ra1ZIUVRGVlJRcEJlRTFMWVROV2FWcFlTblZhV0ZKc1kzcERRMEZUU1hkRVVWbEtTMjlh
U1doMlkwNUJVVVZDUWxGQlJHZG5SVkJCUkVORFFWRnZRMmRuUlVKQlRDdG5DbWgxZVVzMEswZ3dSRGRXUVZWUUwwdEdWbFZRUkhRcllrTnhiMEZT
YkRNeVlteEhaV0pxYlZJeGRuZEdZalJYYlhkSVdEWmhOVFZMU0hWbmNGb3lUbThLVVVjNVQzcEhkMjVNWmpKeGRGSlBZbkpvTHk5bFkxbFBaRmxD
VjNWRWIySk1lbkYyYW05WllWZFlZVnB1TVhKVllYY3dSekZuYUhab1ZYUTVSbE56YlFwd1NHeHNXVzh3YzJOU1ZGRnhkMWRyUXpOaVprMXpkWFpY
Ukd0cGFsRlVjMnhzYTFBNVdXZENjMFJZYlZBNU4zaEdkR0pXT0ZodFVtOXpkREZhUTJOS0Nua3hiRUpSU0RGM1R6TlNNR2h4YkUxUGNGZG9ZVzlI
TUhObldqWldhR3g1YW5OVk0xZHFiWGhFVm1abE5tczNObmx0WmpCM1MxZGhVWHBoUjFGdlpGZ0tlQ3RQWTJSaWJHcDNlVXh4VWs1U2NWaFljRGRI
T1VOWVlXRkVPWGxYT1cxTlZ6UkRiVFI2VDFweU1VZG5SMDF3YkZSMGFWVTBiWFJSY1dKUmNYTjFWQXBaVVVRekwwaEVTVGg0TmpSMVZWQk5UbXRG
UTBGM1JVRkJZVTVEVFVWQmQwUm5XVVJXVWpCUVFWRklMMEpCVVVSQlowdHJUVUU0UjBFeFZXUkZkMFZDQ2k5M1VVWk5RVTFDUVdZNGQwaFJXVVJX
VWpCUFFrSlpSVVpJVWt0SVdHRXZTRFpPUlZaUFIzUjJSV00xVkRWM1UybHVkMGhOUVRCSFExTnhSMU5KWWpNS1JGRkZRa04zVlVGQk5FbENRVkZD
VDNKeWQzSllkbVo1U2xWUFIxaFZiMXB6ZEdoTFUxZ3ZhRGRtZVdGa01WSnNiV3RQYzBoMldtMDNORUpNZURGNVFncHhTREZMU1hjMlozQktWMUpH
UkVKc01GUlBPWFYxWWtvemNHMXRaMWx1WW5wUFRqRXJVVFJPV1dscWNsYzVXVEpwTXk5bWNDdEpTWEZFV0VwTFRYZHpDbFpsYjNWeWFHZFRTVkEx
U0RaSFJVMHZUalJLVjBJeU4xVTFXVWswVlZKQlNEWjFkRU14Vm5sblZVOUNUMHN3Tm5wQmVXbEJabWROVERseE0wVkZXRFVLVlZwVmVUQnVSRUV2
WVhGUFZqVkdZa3hMTmxWek1tWnRNREpXUW05SVZFNTRURXBWYlV4b2FFSTJWVkZQUkRKSlpGRTJiRWhhUXpNdk9HVk5PVTR3T0FvcmNWSklObXM1
UjI5dGRHbElUM2R5WkVwd1dqTklVa3huYVdoV1ZrUjZhbVIxWkRoMFVWUnRhVGwyUTBsWVFXeFJiek5vZWxabVJUbERhVTVEVEVSRENtWkZia2xV
WTA1Q2VXbEtRbU42TldaVVVuZFhWa1JKWlc5QlYzTkZabTVJY1ROSGVRb3RMUzB0TFVWT1JDQkRSVkpVU1VaSlEwRlVSUzB0TFMwdENnPT0= #
base64 encoded
```

Networks

Node Network

Creating an EVE-style device network requires the usage of two CRDs, one for configuration information, which can be reused, and one for the on-device network itself.

Note that the CRD `NetworkConfig` (below) is very similar in principle to the Kubernetes NetworkAttachmentDefinition.

Network configuration:

```
apiVersion: "eve.lfedge.org/v1beta1"
kind: NetworkConfig
metadata:
  name: default-ipv4
  namespace: enterprise1
spec:
  ip: dhcp
  proxies:
  - https://10.100.100.1:8888
```

Network instantiation:

```
apiVersion: "eve.lfedge.org/v1beta1"
kind: DeviceNetwork
metadata:
  name: default-ipv4
  namespace: enterprise1
spec:
  networkConfig: default-ipv4

  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: name
            operator: In
            values:
            - lab1-nuc
            - lab2-nuc
```

Workload Network

We leverage the cncf standard annotations on the workload to indicate desired networks on the actual workload.

```
annotations:
    k8s.v1.cni.cncf.io/networks: default-ipv4,macvlan2 # must exist on edge device
```

Storage

The EVE semantics for storage are as follows.

Every `AppInstanceConfig` needs one or more `Volume`, which in turn are based either on a blank volume or a `ContentTree`. When creating an Edge App Instance, these are converted either into disk images which are attached to VMs or mount points attached to containers. The first provided `Volume` is the bootable one for VMs or container image for containers. Subsequent Volumes may be read-only or read-write.

- AppInstanceConfig
  - Volume1
    - ContentTree
  - Volume2
    - ContentTree
  - Volume3
    - Blank
  - …

Kubernetes already supports the basic structures - multiple volumes, custom storage volumes and custom storage drivers - with the `StorageClass` resource. We create `StorageClass` for blank and each source:

- eve-blank: for a blank disk or mountpoint
- eve-quay: from container image on quay.io
- eve-docker: from container image on docker hub
- etc.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: eve-quay
  namespace: enterprise1
provisioner: eve
parameters:
  type: container  # must be supported type: container, http, ftp, etc.
  URL: https://quay.io
  credentialsSecret: quay-creds # Secret enterprise1/quay-creds
```

for blank:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: eve-blank
provisioner: eve
parameters:
  type: blank
```

Credentials secrets, if needed, are affiliated with the StorageClass as `credentialsRef`.

We define Custom Resources for Image, and then use admissions controllers to validate that the requested resources exist when deploying a Pod that references them.

```
apiVersion: "eve.lfedge.org/v1beta1"
kind: Image
metadata:
  name: golden-ubuntu-2004
  namespace: enterprise1
spec:
  ref: corp1/ubuntu:20.04
  storageClass: eve-quay  # must match the name of a StorageClass

  type: user  # can be any field; a controller may define special names; eve-os is reserved
```

The Image name is then used in a `PersistentVolumeClaim`. See below.

Workloads

We define a workload called `Application` for an application to run on an Edge Device. It references all of the necessary elements. Note that it is called an `Application`, and not an `ApplicationInstance`. An *instance* is something that occurs when deployed on an Edge Device. The `Application` is the Kubernetes resource describing the *intent* to deploy.

We reuse standard constructs from native Kubernetes workloads, i.e. pods, as much as possible:

- To determine the node(s) to which an `Application` is to deploy, we use the standard Kubernetes node affiliation constructs for pods, specifically `nodeAffinity` and `nodeSelector`.
- To determine the `DeviceNetwork`(s) to which the Application should attach, we use the standard networks annotation `k8s.v1.cni.cncf.io/networks`.
- For storage volumes:
    - For the boot image, we use the `image` field, which must match a named Image.
    - For other volumes, we use Kubernetes `PersistentVolumeClaim`.
- Support for multiple containers enables future packaging of multiple workloads together.

Boot image

The `image` field refers to the name of a defined `Image`.

Additional Volumes

For all additional storage volumes, we use Kubernetes Volume resources, specifically `PersistentVolumeClaim`.

For example:

Golden filesystem image stored on FTP site, mounted as a filesystem. Defined using the StorageClass `eve-ftp`.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: fsclaim
spec:
  accessModes:
    - ReadWriteOnce  # can be ReadWriteOnce, ReadOnlyMany, etc.
  volumeMode: Filesystem  # can be Filesystem or Block
  resources:
    requests:
      storage: 8Gi  # this is for the size
  storageClassName: eve-ftp
  dataSourceRef:
    group: eve.lfedge.org/v1beta1
    kind: image
    name: golden-ubuntu-2004
```

Golden VM image stored on FTP site, mounted as a block device. Defined using the StorageClass `eve-ftp`.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ubuntuclaim
spec:
  accessModes:
    - ReadWriteOnce  # can be ReadWriteOnce, ReadOnlyMany, etc.
  volumeMode: Block  # can be Filesystem or Block
  resources:
    requests:
      storage: 8Gi  # this is for the size
  storageClassName: eve-image
  dataSourceRef:
    group: eve.lfedge.org/v1beta1
    kind: image
    name: golden-ubuntu-2004
```

Blank disk volume.

```
kind: PersistentVolumeClaim
metadata:
  name: blankdisk
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem  # can be Filesystem or Block
  resources:
    requests:
      storage: 8Gi  # this is for the size
  storageClassName: eve-blank
```

Status

The state of an `Application`, as reported by the controller, is set on the `ApplicationStatus`. For example:

```
apiVersion: eve.lfedge.org/v1beta1
kind: Application
metadata:
  name: app-ubuntu
  namespace: enterprise1
  annotations:
    k8s.v1.cni.cncf.io/networks: wlan-local,vpn-corp # must be known
spec:
  nodeSelector:  # reuse this because it is native to many resources
    name: edge-node-01
...
status:
  key: value
  key: value
```

The `ApplicationStatus` field is similar to the Kubernetes PodStatus, albeit not identical. The fields are as follows.

- state: current state of the Application.
- statuses: array of historical states of the application. Each state includes two fields:
  - state: the state when this status occurred.
  - timestamp: the timestamp when this status occurred.

The states of the application are the ones currently supported by the EVE API. E.g. `BOOTING`, `RUNNING`, `STARTED`.

Complete Example

```
apiVersion: eve.lfedge.org/v1beta1
kind: Application
metadata:
  name: app-ubuntu
  namespace: enterprise1
  annotations:
    k8s.v1.cni.cncf.io/networks: wlan-local,vpn-corp # must be known
spec:
  nodeSelector:  # reuse this because it is native to many resources
    name: edge-node-01
  containers:
    - name: frontend
      image: golden-ubuntu-2004  # must be an Image resource
      resources:
        requests:
          cpu: 1.0
          memory: 256M
          storage: 8G
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
      volumeDevices:
      - devicePath: "/dev/sda2"
        name: ubuntu
      - devicePath: "/dev/sda3"
        name: raw
  volumes:
    - name: pd
      persistentVolumeClaim:
        claimName: fsclaim
    - name: ubuntu
      persistentVolumeClaim:
        claimName: ubuntuclaim
    - name: raw
      ephemeral:
        volumeClaimTemplate:
          spec:
            accessModes:
              - ReadWriteOnce
            volumeMode: Block
            resources:
              requests:
                storage: 8Gi
            storageClassName: blankdisk
```

Scheduling

We define higher-level scheduling constraints, specifically `ApplicationDeployment`, `ApplicationDaemonSet`, `ApplicationStatefulSet`. These are optional; a controller MAY implement them, but is not required to do so.

These follow the same semantics as normal Pod-scheduling `Deployment`, `DaemonSet` and `ApplicationSet`, except that the `spec.template.spec` references an `Application` rather than a `Pod`.

BaseOS

Setting the baseOS version on a device is performed by changing the spec field `eve-os-version`. Note that the status field `eve-os-version` reports what version actually is installed and running on the device, while the spec field `eve-os-version` is the desired version.

If the spec field `eve-os-version` is blank, then there is no specific request; it simply is whatever was installed on the device. The field should be changed only when requesting a change of the eve-os version on the device. The status field is to be updated when the update is complete.

Controllers may have policies that require an eve-os image be registered on the controller, and possibly in the specific namespace, before allowing it to be applied to a device. This is a policy question for the controller, and is not covered by this specification.

Appendix A - Native Resources

Edge Device

The concepts of a Kubernetes node and an EVE edge device are very similar, with support for various properties. The flows of adding and managing them differ.

EVE

EVE's API for joining a controller uses a combination of three distinct elements from the device:

- Onboard certificate
- Serial
- Device certificate

EVE's API does not define how the controller will use those, and in what combination. Nor does it define if a certificate will be validated by advance knowledge of the certificate *itself*, or by acceptance of the CA that signed the cert, if any.

The controller can use any combination of the above it desires to authenticate and then register the device.

In practice, most controllers provide one or more of the following:

- Device pre-allocated and device certificate itself known in advance.
- Device pre-allocated and onboard certificate itself known in advance.
- Device pre-allocated and combination of onboard certificate and serial known in advance.
- Device not pre-allocated; onboard certificate known and accepted for device registration.
- Device not pre-allocated; onboard certificate's signer known and accepted for device registration.

The process is:

1. User: Create a new device, with some combination, depending on controller, of:
    a. Device certificate, only available after device boots with physical access
    b. Onboarding certificate
    c. Unique serial
2. Device: Connect to controller, identify using the above
3. Device now is onboarded

Kubernetes

Any node that can connect to the control plane and prove it has a valid identity is accepted as a worker node. This proof is one of:

- A node certificate signed by an acceptable CA - this CA is valid for all nodes, not just for one node.
- A pre-shared secret token - this token is valid for any node, not just for one node. Kubernetes uses that token to generate a new node certificate for the node.

Note that these two methods of joining - an existing "node certificate" and a token - are similar in concept to the EVE API "device certificate" and "onboard certificate". However, their usage is quite different.

In Kubernetes one normally does not create a node via the API; the node exists by virtue of its joining a cluster. However, it is possible to create one via the API. It is unclear how the node, upon joining, will reconcile with the existing node resource.

	Kubernetes Node Certificate	**EVE Device Certificate**
Validation	Signed by valid CA	Actual certificate in controller

	Kubernetes Token	**EVE Onboard Certificate**
Validation	Shared secret	Actual certificate in controller
Usage	Generate node certificate	Accept presented device certificate

All additional features and properties of the node that are not directly related to the cluster itself, including taints and tolerations, are handled via metadata, specifically labels and annotations. Since these are semi-arbitrary key-value pairs, anything can be placed here.

We use annotations to determine whether or not to onboard the Node, i.e. its activation status.

When the Edge Device onboards, changed the `NodeStatus` to Ready.

Note that the OS, architecture and other descriptive elements of the Node are natively part of `NodeStatus.NodeInfo`.

The following is a sample. Note that the `status` section normally is returned by the device, rather than set. However, it can be set via a client.

```
apiVersion: v1
kind: Node
metadata:
  annotations:
    eve.lfedge.org/node-type: virtual
    eve.lfedge.org/eve-version: 6.12.2
    eve.lfedge.org/location: "texas/usa"
    eve.lfedge.org/activate: true
    eve.lfedge.org/network-eth0: default-ipv4
    eve.lfedge.org/network-eth1: management-only
  labels:
    beta.kubernetes.io/arch: amd64
    beta.kubernetes.io/instance-type: eve
    beta.kubernetes.io/os: linux
    kubernetes.io/arch: amd64
    kubernetes.io/hostname: eve-device.lab1
    kubernetes.io/os: eve
    node.kubernetes.io/instance-type: eve
  name: eve-device.lab1
spec:
  providerID: eve://eve-device.lab1
status:
  addresses:
  - address: 172.19.0.3
    type: InternalIP
  - address: k3d-k3s-default-server-0
    type: Hostname
  allocatable:
    cpu: "4"
    ephemeral-storage: "296591664715"
    hugepages-1Gi: "0"
    hugepages-2Mi: "0"
    memory: 16235544Ki
    pods: "110"
  capacity:
    cpu: "4"
    ephemeral-storage: 304884524Ki
    hugepages-1Gi: "0"
    hugepages-2Mi: "0"
    memory: 16235544Ki
    pods: "110"
  conditions:
  - lastHeartbeatTime: "2021-11-23T12:57:09Z"
    lastTransitionTime: "2021-10-10T10:33:38Z"
    message: kubelet is posting ready status
    reason: KubeletReady
    status: "True"
    type: Ready
  nodeInfo:
    architecture: amd64
    bootID: dc703fd4-543b-4801-96be-4d6d29afb41e
    containerRuntimeVersion: containerd://1.4.9
    kernelVersion: 5.10.1
    machineID: ""
    operatingSystem: eve
    osImage: eve
    systemUUID: EC232B65-602A-F2A9-287B-5D95721116E6
```

We use taints to prevent any normal pods from being deployed to the node. Only targeted EVE pods which have the correct tolerations are deployed to the node.

Networks

EVE has 2 network constructs that simply do not exist in Kubernetes:

- device network: one or more network definitions on each Edge Device
- workload network: the device network to which a workload should connect

For Kubernetes, a network only comes into existence when a workload pod is created. The kubelet (responsible for running the pod):

1. Creates the container, including network namespace
2. Calls CNI, passing it the container network namespace, which:
    a. Creates the network interface in the container network namespace
    b. Plumbs the network interface to whichever network it desires
    c. Allocates and attaches an IP

Node Network

The network for a node is completely out of Kubernetes scope. How the node connects to networks, what it uses its physical NICs for, configuration of them - DHCP, DNS, WiFi credentials, etc. - is something that is dealt with prior to onboarding the node and unrelated to it.

In EVE, each NIC on an Edge Device is given a specific Network definition via the EdgeDevConfig config with which to work. These *must* be defined, otherwise the NIC will be considered unmanaged and not used.

Thus, the node network, or device network, in the Native Resources option is identical to and uses the same CRDs as the official design CRD option.

Workload Network

In Kubernetes, each workload container natively gets two virtual NICs, `eth0` and `lo`. While `lo` is connected to loopback, `eth0` is connected via CNI to whichever network is the default on that host.

Once it is connected, it is assumed to be able to communicate with all other workloads and hosts, as well as the larger outside network itself, subject to network policy rules. The idea of multiple networks to which some workloads are connected and some are not, simply does not exist.

As a result, in Kubernetes, there is no Network object, nor is there native support for multiple network implementations. By extension, workloads have no native property for “network to join”, because all containers by default join just the loopback `lo` and `eth0` via CNI.

However, CNI is capable of doing almost anything it wants, including adding multiple interfaces connected to the same networks, connecting workloads to multiple networks, or even to no networks. On the basis of this, there are standards for defining different networks and implementations of CNI, notably the official reference implementation multus, that know how to use those definitions. Note that these are not built into Kubernetes, but rather take advantage of CRDs and CNI.

To *declare* the usage of networks, there are standard CNCF annotations to be applied to the workload that indicate the desired networks:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
  annotations:
    k8s.v1.cni.cncf.io/networks: default-ipv4,macvlan2 # must exist on edge device
```

We leverage the cncf standard annotations to *declare* desired networks in EVE as well. However, the *implementation* is specific to the endpoint. In the EVE case, we use the existing network structures.

Network ACLs

Within the single flat Kubernetes workload network, there is optional access control using NetworkPolicy.

- If no `NetworkPolicy` is declared, then all workloads (pods) have unfettered access to all other workloads.
- If even a single `NetworkPolicy` is declared in a namespace, then access to any workloads in that namespace is denied unless explicitly allowed by a `NetworkPolicy`.

These policies can restrict ingress, egress or both. They apply to workloads based on normal selectors, usually labels. They can allow based on the remote end’s IP address/range/cidr, port, namespace, label or any combination thereof.

`NetworkPolicy` always is "default deny". Once applied in a namespace, ingress and egress will be allowed only if expressly allowed in a `NetworkPolicy`. There can be as many `NetworkPolicy` applied as desired.

We do not use `NetworkPolicy` as part of declaring network access for EVE. In the future, we may consider controlling EVE network ACLs with `NetworkPolicy`.

Storage

Kubernetes supports independent storage as the resources `Volume` and `PersistentVolume`. Each Volume can be one of several defined types, but generally is one of two categories:

- Ephemeral: created with the workload on the node and eliminated when the workload goes away.
- Persistent: backed by some more permanent form of storage, either network or local disk.

The key difference is not network vs local, but persistence beyond the life of a single workload.

EVE does not currently support network-mounted volumes. EVE currently preserves all volumes upon termination of a workload, until explicitly deleted.

The resource structure is identical to that described in CRDs.

Workloads

EVE workloads, defined by `AppInstanceConfig`, map to Kubernetes Pods. Kubernetes also has higher order scheduling constructs - `Deployment`, `DaemonSet`, `StatefulSet` - which we will use as well, but only as optionally supported by the controller.

Kubernetes Pods are not inherently attached to a specific Node. There are constructs in the Pod that can require or prefer it to be on a certain node or class of nodes, but it is not the default or native way of scheduling.

The specific uniquenesses of Edge Apps are handled using Kubernetes constructs as follows:

- Edge Device selection: `nodeSelector`
- Networking: the standard networks annotation `k8s.v1.cni.cncf.io/networks` is used to list the desired named networks available on the node. The network is checked for existence by an admissions controller.
- Storage:
    - For the boot image, we use the `image` field, which must match a named Image.
    - For other volumes, we use Kubernetes `PersistentVolumeClaim`.

Storage is defined further below.

Boot image

The `image` field of the Pod spec normally refers to a special URL indicating the registry, repository name, and identifier - tag and/or hash - of an OCI-spec compliant container image.

We overload the image field of the podspec by providing a special URL that indicates the image it is being taken from. Valid would be:

- `quay.io/etcd/etcd:3.2.1` - currently acceptable normal OCI image
- `<image-name>` - reference a local image

In order to indicate that the `image` field references an `Image` to be referenced rather than a normal OCI image to be pulled from a registry, we set an annotation on the Pod:

```
annotations:
   eve.lfedge.org/image-source: local
```

These are identical to the CRD image solution, except that the annotation is necessary only when using native Kubernetes pod resources.

Additional Volumes

For all additional storage volumes, we use Kubernetes `Volume` resources, specifically `PersistentVolumeClaim`, in the same manner as the CRD solution.

Examples:

Golden filesystem image stored on FTP site, mounted as a filesystem. Defined using the StorageClass `eve-ftp`.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: fsclaim
spec:
  accessModes:
    - ReadWriteOnce  # can be ReadWriteOnce, ReadOnlyMany, etc.
  volumeMode: Filesystem  # can be Filesystem or Block
  resources:
    requests:
      storage: 8Gi  # this is for the size
  storageClassName: eve-ftp
  dataSourceRef:
    group: eve.lfedge.org/v1beta1
    kind: image
    name: golden-ubuntu-2004
```

Golden VM image stored on FTP site, mounted as a block device. Defined using the StorageClass `eve-ftp`.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ubuntuclaim
spec:
  accessModes:
    - ReadWriteOnce  # can be ReadWriteOnce, ReadOnlyMany, etc.
  volumeMode: Block  # can be Filesystem or Block
  resources:
    requests:
      storage: 8Gi  # this is for the size
  storageClassName: eve-image
  dataSourceRef:
    group: eve.lfedge.org/v1beta1
    kind: image
    name: golden-ubuntu-2004
```

Blank disk volume.

```
kind: PersistentVolumeClaim
metadata:
  name: blankdisk
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem  # can be Filesystem or Block
  resources:
    requests:
      storage: 8Gi  # this is for the size
  storageClassName: eve-blank
```

Complete Example

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
  namespace: enterprise1
  annotations:
    k8s.v1.cni.cncf.io/networks: wlan-local,vpn-corp # must be known
    eve.lfedge.org/image-source: local
spec:
  containers:
    - name: myfrontend
      image: golden-ubuntu-2004  # must be an Image resource
      nodeSelector:
        name: edge-node-01
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
      volumeDevices:
      - devicePath: "/dev/sda2"
        name: ubuntu
      - devicePath: "/dev/sda3"
        name: raw
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: fsclaim
    - name: ubuntu
      persistentVolumeClaim:
        claimName: ubuntuclaim
    - name: raw
      ephemeral:
        volumeClaimTemplate:
          spec:
            accessModes:
              - ReadWriteOnce
            volumeMode: Block
            resources:
              requests:
                storage: 8Gi
            storageClassName: blankdisk
```

Scheduling

Since the workload is a native Kubernetes resource Pod, we use the higher-level scheduling resources to schedule multiple: Deployment, StatefulSet, Dae monSet. These are used in the normal fashion, with the usual node affinities, specifically nodeSelector and nodeAffinity, as described here.