**Leveraging Crossplane to Deploy and Manage a Single-Tenant Architecture**

In today’s rapidly evolving technology landscape, deploying and managing cloud-native applications has become increasingly complex. Organizations often require dedicated environments that offer enhanced security, isolation, and compliance for their workloads. This is where Crossplane, an open-source Kubernetes add-on, emerges as a powerful tool that simplifies the deployment and management of single-tenant architecture. In this article, we will explore how Crossplane empowers organizations to efficiently provision and maintain dedicated environments tailored to their specific needs.

**Understanding Single-Tenant Architecture**

Single-tenant architecture refers to an approach where an application or service operates in isolation, dedicated exclusively to a single user or organization. This architecture ensures maximum privacy, data security, and customizable resource allocation. By adopting a single-tenant model, organizations gain greater control over their environment, enabling them to meet stringent regulatory requirements and deliver a highly personalized user experience.

**The Role of Crossplane**

Crossplane is an open-source Kubernetes add-on that extends the capabilities of Kubernetes by introducing a powerful concept called Infrastructure as Code (IaC). It allows organizations to define and manage infrastructure resources using Kubernetes manifests, making it easier to provision and manage dedicated environments.

1. Simplified Provisioning: Crossplane enables the provisioning of single-tenant infrastructure resources through custom resource definitions (CRDs). Organizations can define their desired infrastructure components, such as virtual machines, databases, storage volumes, and networks, using Kubernetes-style manifests. Crossplane automates the creation and configuration of these resources, reducing the time and effort required for manual provisioning.
2. Enhanced Resource Control: With Crossplane, organizations have fine-grained control over their single-tenant architecture. They can specify resource limits, isolation boundaries, and security policies according to their unique requirements. By leveraging Crossplane’s declarative approach, operators can define infrastructure templates that ensure consistent provisioning across different environments, promoting reproducibility and scalability.
3. Intelligent Workload Placement: Crossplane facilitates intelligent workload placement, enabling organizations to distribute their single-tenant applications across different cloud providers, regions, or on-premises infrastructure. This capability allows businesses to optimize resource utilization, achieve high availability, and implement disaster recovery strategies tailored to their specific needs. Crossplane’s workload placement policies empower organizations to respond dynamically to changing requirements and ensure optimal performance.
4. Lifecyle Management: Managing the lifecycle of single-tenant environments can be challenging. Crossplane provides a unified interface for managing infrastructure resources, making it easier to upgrade, scale, or decommission components as needed. By leveraging Kubernetes’ declarative approach, operators can define desired states for infrastructure resources and leverage Crossplane’s reconciliation process to ensure continuous convergence with the desired state.

**Our Story**

BioCatch, a leading company in the field of behavioral biometrics, has successfully leveraged Crossplane, ArgoCD, and the KubernetesAPI SDK to deploy and manage single-tenant architecture. This article explores how in BioCatch we are utilizing these powerful tools, shares code examples, and outlines the process of deploying Crossplane with ArgoCD while creating custom resource definitions (CRDs) using the KubernetesAPI SDK.

**Deploying Crossplane with ArgoCD**

To streamline the deployment process, we has adopted ArgoCD, a GitOps continuous delivery tool, in conjunction with Crossplane. ArgoCD enable us to automate the deployment and management of Crossplane across their Kubernetes clusters.

In the deployment proccess we are using Helm, a popular package manager for Kubernetes, along with the ArgoCD ApplicationSet which is a declarative way to define multiple instances of an application with varying parameters in ArgoCD, ApplicationSet enables us to dynamically generate and manage multiple applications using a single configuration template.

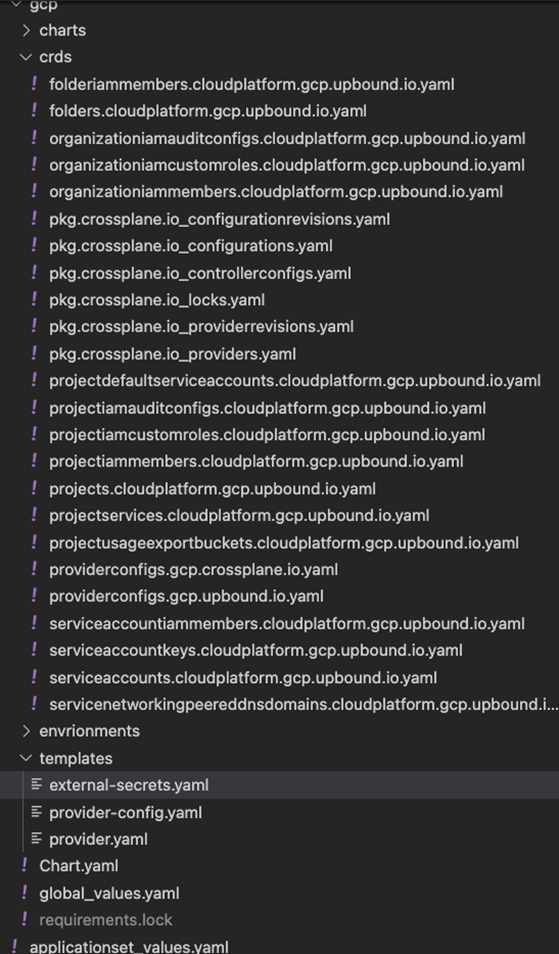
Within the ApplicationSet, we are specifies the Helm chart as the source, along with the values and configurations for the necessary Crossplane custom resource definitions (CRDs) specific to each instance.

To begin the provisioning process, We have created an inhouse Helm chart that defines the desired state of the Crossplane installation, including the required CRDs. The Helm chart contains the necessary templates, values, and dependencies to deploy Crossplane.

To illustrate how we provision Crossplane with the required CRDs with ArgoCD, let’s explore the code snippet below.



In this code snippet, we are dynamically declaring the ApplicationSet to instruct ArgoCD how to deploy our Crossplane Application across multiple Kubernetes clusters. With that we can easily deploy and manage multiple instances of Crossplane.



The above code structure represents the Helm directory where the ApplicationSet refers to. ArgoCD discovers automatically the CRDs directory within the chart structure, packaging it alongside the rest of the Helm chart resources and templates.

During the Helm deployment process, ArgoCD follows the specified order of resource creation. As CRDs must be created before any resources that depend on them, ArgoCD deploys the CRDs first.

Under the “templates” directory in the Helm chart for ArgoCD, we have the following resources:

*ExternalSecret*  
The ExternalSecret custom resource enables secure integration of external secret stores with Kubernetes for seamless management of sensitive information.

As part of our deployment process, we are using ExternalSecret CRD for retrieving Cloud Provider Credentials Authentication from Hashicorp Vault by calling to the “key=value” via description “path.”



For more about ExternalSecret and how BioCatch is using it, [refer to the article here](https://medium.com/@BioCatchTechBlog/step-up-your-gitops-game-using-external-secrets-operator-cf60d9e8bf9f).

**Crossplane Provider**

Similar to Terraform, Crossplane has a provider package that fully retrieves the SDK of the provider in declarative mode of Crossplane (Like HCL).



**Crossplane ProviderConfig**

The “ProviderConfig” custom resource in Crossplane specifies the configuration settings and credentials required to establish a connection with a specific cloud or infrastructure service provider. It enables Crossplane to securely authenticate and interact with the provider’s resources on behalf of the user or application.



In the code snippet, “secretRef” declarative “telling” the provider what authentication credentials secret and key to use.

By merging this code to the master, ArgoCD will provision Crossplane across all the “workloads” cluster that we have defined with the required CRDs.

**Automating Customer Onboarding and Resource Provisioning in the BioCatch Connect Single-Tenant Architecture**

In the BioCatch Connect automation process framework, we have implemented a Python SDK that empowers us to seamlessly create new customers within our single-tenant architecture. This SDK serves as a framework that integrates with different components, including the Kubernetes API, enabling us to automate the provisioning of all the required Kubernetes resources for each customer.

Using the Python SDK, we programmatically generate and configure dedicated namespaces and Blob Storage resources for individual customers. Each customer is allocated a separate namespace and Blob Storage to ensure logical isolation and secure boundaries within our single-tenant environment. This namespace-per-customer approach allows us to encapsulate and manage the resources associated with each customer’s workload application effectively.

**The Way of Crossplane**

In the workload identity approach within Google Cloud Platform (GCP), it is essential to establish a binding between the service account in GCP and the service account within the Kubernetes cluster’s namespace. Since the binding is namespace-based and needs to be dynamically created, we leverage Crossplane to facilitate the creation of the Google service account and establish the necessary association between them.

When adopting the workload identity approach in GCP, each Kubernetes namespace requires its own service account to authenticate and access GCP resources. However, creating and managing these service accounts manually can be cumbersome and error prone. This is where Crossplane comes in as a valuable tool.

Crossplane integrates with GCP and provides a declarative approach to manage infrastructure and resources, including service accounts. By leveraging Crossplane, we can dynamically create Google service accounts directly from Kubernetes manifests.

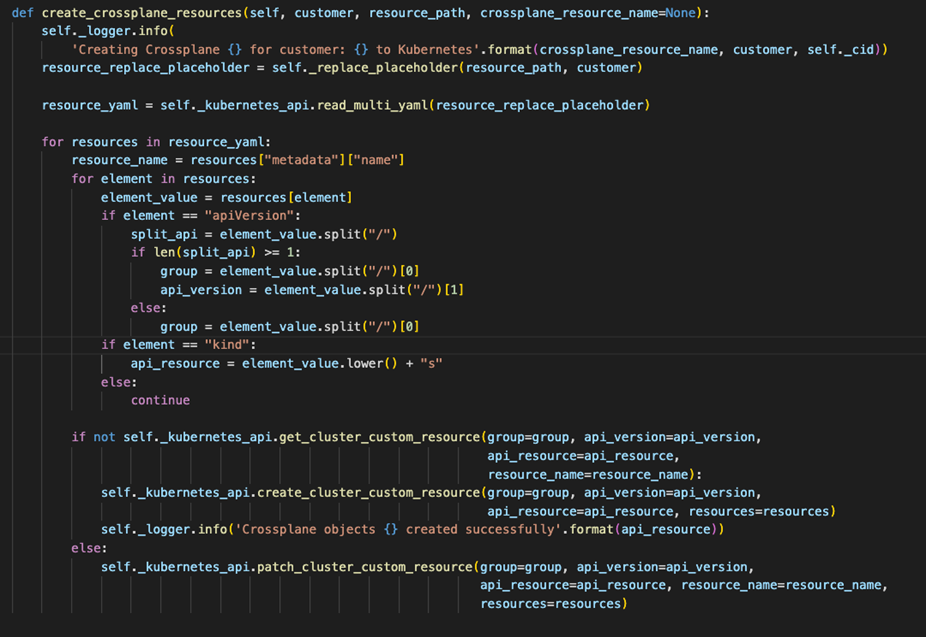
The provided code demonstrates how Python SDK utilizes the Kubernetes API to automate the creation of a Service Account and a Crossplane Custom Resources within the Kubernetes cluster.

**K8S Service Account Creation**



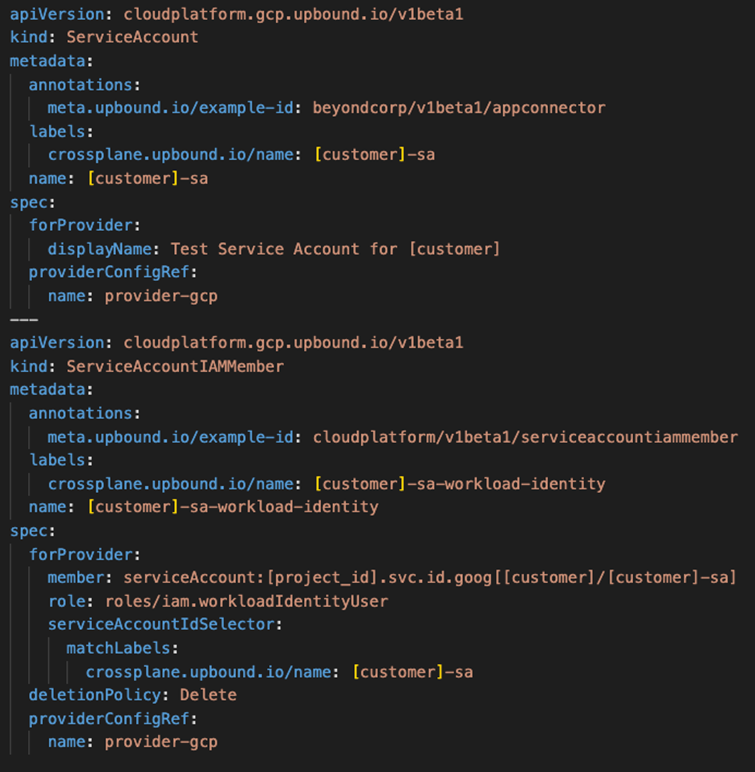
**Kubernetes Python SDK**

The Kuberentes Python SDK code snippet below shows how BioCatch Connect dynamically creates Custom Resources in Kubernetes.



**Crossplane Custom Resources**

Google Cloud Service Account / ServiceAccount IAM Member (Workload Identity)

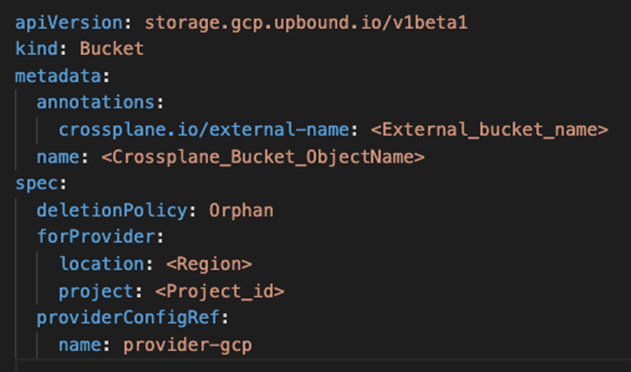


In this code snippet, Kubernetes creates “ServiceAccount” and “ServiceAccountIAMMember” Crossplane custom resources.  
  
Crossplane upbound package will create a new Google Cloud service account in our project and will provide a Service Account IAM member of “workloadIdentityUser” IAM role.

Crossplane then orchestrates the creation of the Google service account in GCP ensuring that the account is properly configured and associated with the Kubernetes namespace. This allows for seamless authentication and access to GCP resources from within the Kubernetes cluster.

**Google Cloud Cloud Storage Bucket/BucketIAMMember**

As we have mentioned, BioCatch Connect uses a single tenant approach, therefore each customer has its own Blob Storage Cloud resource with restriction privileges only to the Customer Kubernetes service account.



In the Customer creation process, the Blob Storage Google Bucket are already created by the Python Google Cloud SDK, therefore we are using the following:

1. **“deletionPolicy: Orphan”** to restrict Crossplane for deleting the Google Cloud Bucket if the Crossplane custom resource is accidently deleted.
2. **“crossplane.io/external-name:** <External\_bucket\_name>” to let Crossplane know that we want to import existing Google Cloud Bucket to the custom resource.

Now we need to allow only the specific Customer Kubernetes service account a privilege to reach to his own Blob storage (Google Cloud bucket) and only to him.

BioCatch Connect achieved that by using Crossplane BucketIAMMember custom resources:

1. **bucketRef:**describing the Google Cloud Bucket resource name
2. **Member:**Google Cloud service account
3. **Role:**specifying the role for the BucketIAMMember

