Application Single Sign-On for VMware Tanzu v1.0

Application Single Sign-On for VMware Tanzu 1.0

You can find the most up-to-date technical documentation on the VMware website at:

https://docs.vmware.com/

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Contents

Application Single Sign-On for VMware Tanzu® (1.0.0)

- VMware Slack #app-sso
- Learn more about us here

Application Single Sign-On for VMware Tanzu®, short *AppSSO*, provides APIs for curating and consuming a “Single Sign-On as a service” offering on *Tanzu Application Platform*.

- Want to get started with AppSSO? Start with the Getting Started guide.

With AppSSO *Service Operators* can configure and deploy authorization servers. *Application Operators* can then configure their Workloads with these authorization servers to provide Single Sign-On to their end-users.

AppSSO allows to integrate authentication and authorization decisions early in the software development and release lifecycle. It provides a seamless transition for workloads from development to production when including Single Sign-On solutions in your software.

It’s easy to get started with AppSSO; deploy an authorization server with static test users. Eventually, progress to multiple authorization servers of production-grade scale with token key rotation, multiple upstream identity providers and client restrictions.

AppSSO’s authorization server is based off of Spring Authorization Server.

Getting started

This article assumes AppSSO is installed on your TAP cluster. To install AppSSO, refer to the instructions in Install AppSSO.

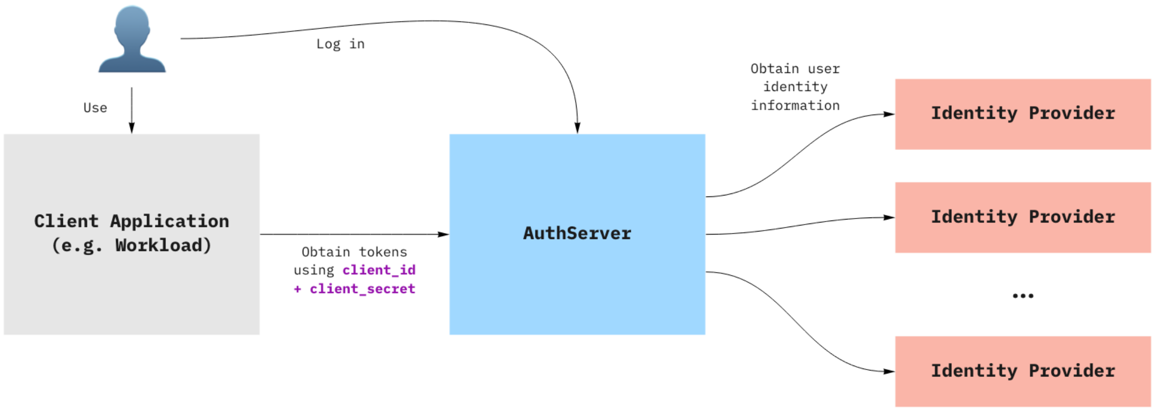
In this section, you will:

1. Get an overview of AppSSO
2. Set up your first authorization server, and validate that it is running
3. Expose it over HTTP through an HTTPProxy, and validate it can be reached
4. Provision a ClientRegistration, and validate it is working
5. Deploy an application that uses the provisioned ClientRegistration to enable SSO

Once you have completed the above steps, you can continue by securing a Workload.

AppSSO Overview

At the core of AppSSO is the concept of an Authorization Server, outlined by the AuthServer custom resource. Service Operators create those resources to provision running Authorization Servers, which are OpenID Connect Providers. They issue ID Tokens to Client applications, which contain identity information about the End-User (such as email, first name, last name, etc).



When a Client application uses an AuthServer to authenticate an End-User, the typical steps are:

1. The End-User visits the Client application
2. The Client application redirects the End-User to the AuthServer, with an OAuth2 request
3. The End-User logs in with the AuthServer, usually using an external Identity Provider (e.g. Google, Azure AD)

    1. Identity Providers are set up by Service Operators
    2. AuthServers may use various protocols to obtain identity information about the user, such as OpenID Connect, SAML or LDAP, which may involve additional redirects

4. The AuthServer redirects the End-User to the Client application with an authorization code
5. The Client application exchanges with the AuthServer for an `id_token`
    1. The Client application does not know how the identity information was obtained by the AuthServer, it only gets identity information in the form of an ID Token.

ID Tokens are JSON Web Tokens containing standard Claims about the identity of the user (e.g. name, email, etc) and about the token itself (e.g. “expires at”, “audience”, etc). Here is an example of an `id_token` as issued by an Authorization Server:

```
{
  "iss": "https://appsso.example.com",
  "sub": "213435498y",
  "aud": "my-client",
  "nonce": "fkg0-90_mg",
  "exp": 1656929172,
  "iat": 1656928872,
  "name": "Jane Doe",
  "given_name": "Jane",
  "family_name": "Doe",
  "email": "jane.doe@example.com"
}
```

ID Tokens are signed by the AuthServer, using Token Signature Keys. Client applications may verify their validity using the AuthServer’s public keys.

Getting started

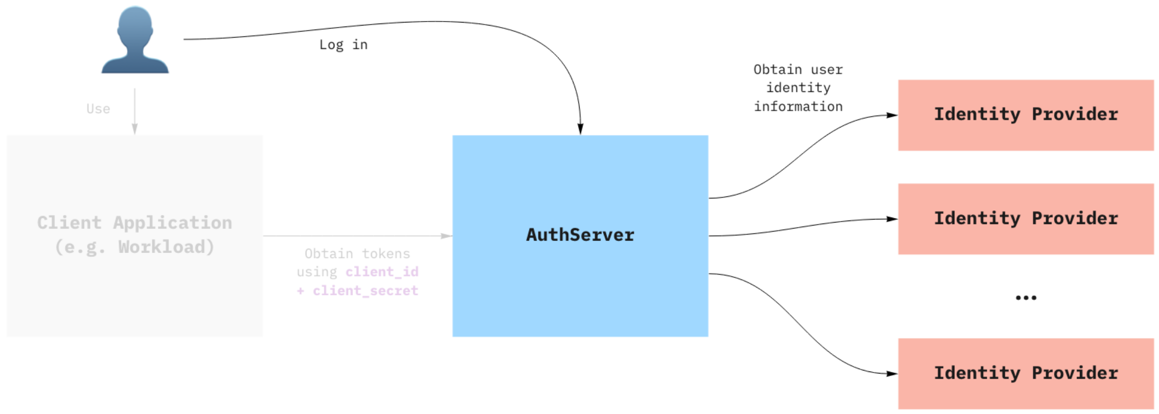
> Move on to Provision your first AuthServer

Provision an AuthServer

> This article assumes AppSSO is installed on your TAP cluster. To install AppSSO, refer to the instructions in Install AppSSO.
>
> AppSSO is installed automatically installed with the `run`, `iterate`, and `full` TAP profiles, no extra steps required.
>
> To make sure AppSSO is installed on your cluster, you can run:
>
> ```
> tanzu package installed list -A | grep "sso.apps.tanzu.vmware.com"
> ```

In this tutorial, you are going to:

1. Set up your first authorization server, in the `default` namespace
2. Ensure it is running, that users can log in



Provision an AuthServer

First, deploy your first Authorization Server, along with a secret key for signing tokens.

> Note that we used `spec.issuerURI` = `http://authserver.example.com`, but you should customize the URL to match the domain of your TAP cluster.

```
---
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
metadata:
  name: my-authserver-example
  namespace: default
  labels:
    name: my-first-auth-server
    env: tutorial
  annotations:
    sso.apps.tanzu.vmware.com/allow-client-namespaces: "default"
    sso.apps.tanzu.vmware.com/allow-unsafe-issuer-uri: ""
    sso.apps.tanzu.vmware.com/allow-unsafe-identity-provider: ""
spec:
  replicas: 1
  issuerURI: "http://authserver.example.com"
  tokenSignature:
    signAndVerifyKeyRef:
      name: "authserver-signing-key"
  identityProviders:
    - name: "internal"
      internalUnsafe:
        users:
          - username: "user"
            password: "$2a$10$201z9o/tHlocFsHFTo0plukh03ApBYe4dRiXcqeyRQH6CNNtS8jWK"
            email: "user@example.com"
            roles:
              - "user"
---
apiVersion: secretgen.k14s.io/v1alpha1
kind: RSAKey
metadata:
  name: authserver-signing-key
  namespace: default
spec:
  secretTemplate:
    type: Opaque
    stringData:
      key.pem: $(privateKey)
      pub.pem: $(publicKey)
```

Validate that the auth-server runs, by checking the `AuthServer` resource's status. Both the `IssuerURIReady` and `Ready` conditions should be False, but all other conditions should be True. This is because your `AuthServer` is not accessible yet.

```
kubectl get authservers my-authserver-example -n default -o yaml
# If you want to check which conditions are not ready, you may use jq for example:
kubectl get authserver my-authserver-example -n default -o json | jq ".status.conditions[] | select(.status != \"True\") | .type"
# IssuerURIReady
# Ready
```

Then, validate that the deployment is responding over HTTP by exposing it:

```
kubectl port-forward -n default deploy/my-authserver-example-auth-server 7777:8080
```

And navigating in your browser to http://localhost:7777. There you should see a login page. Log in using username = `user` and password = `password`.

Note that if you are using TKGm or TKGs, which have customizable in-cluster communication CIDR ranges, there is a known issue regarding AppSSO making requests to external identity providers with `http` rather than `https`.

The AuthServer spec, in detail

Here is a detailed explanation of the `AuthServer` you have applied in the above section. This is intended to give you an overview of the different configuration values that were passed in. It is not intended to describe all the ins-and-outs, but there are links to related docs in each section.

Feel free to skip ahead.

Metadata

```
metadata:
  labels:
    name: my-first-auth-server
    env: tutorial
  annotations:
    sso.apps.tanzu.vmware.com/allow-client-namespaces: "default"
    sso.apps.tanzu.vmware.com/allow-unsafe-issuer-uri: ""
    sso.apps.tanzu.vmware.com/allow-unsafe-identity-provider: ""
```

The `metadata.labels` uniquely identify the AuthServer. They are used as selectors by `ClientRegistrations`, to declare from which authorization server a specific client obtains tokens from.

The `sso.apps.tanzu.vmware.com/allow-client-namespaces` annotation restricts the namespaces in which you can create a `ClientRegistrations` targeting this authorization server. In this case, the authorization server will only pick up client registrations in the `default` namespace.

The `sso.apps.tanzu.vmware.com/allow-unsafe-...` annotations enable “development mode” features, useful for testing. Those should not be used for production-grade authorization servers.

Lean more about Metadata.

Issuer URI

```
spec:
  issuerURI: "http://authserver.example.com"
```

This is the URL that the auth server will serve traffic from. The authorization server will issue tokens containing this `issuerURI`, and clients will use it to validate that the token comes from a trusted source.

Note: HTTP access is for getting-started development only! Learn more about a production ready Issuer URI

Lean more about Issuer URI.

Token Signature

```
---
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
# ...
spec:
  tokenSignature:
    signAndVerifyKeyRef:
      name: "authserver-signing-key"
---
apiVersion: secretgen.k14s.io/v1alpha1
kind: RSAKey
metadata:
  name: authserver-signing-key
  namespace: default
spec:
  secretTemplate:
    type: Opaque
    stringData:
      key.pem: $(privateKey)
      pub.pem: $(publicKey)
```

The token signing key is the private RSA key used to sign ID Tokens, using JSON Web Signatures, and clients use the public key to verify the provenance and integrity of the ID tokens. The public keys used for validating messages are published as JSON Web Keys at `{issuerURI}/oauth2/jwks`. When using the port-forward declared in the section above, JWKs are available at http://localhost:7777/oauth2/jwks.

The `spec.tokenSignature.signAndVerifyKeyRef.name` references a secret containing PEM-encoded RSA keys, both `key.pem` and `pub.pem`. In this specific example, we are using Secretgen-Controller, a TAP dependency, to generate the key for us.

Lean more about Token Signature.

Identity providers

```
spec:
  identityProviders:
    - name: "internal"
      internalUnsafe:
        users:
          - username: "user"
            password: "$2a$10$201z9o/tHlocFsHFTo0plukh03ApBYe4dRiXcqeyRQH6CNNtS8jWK"
            email: "user@example.com"
            roles:
              - "user"
```

AppSSO's authorization server delegate login and user management to external identity providers (IDP), such as Google, Azure Active Directory, Okta, etc. See diagram at the top of this page.

In this example, we use an `internalUnsafe` identity provider. As the name implies, it is *not* an external IDP, but rather a list of hardcoded user/passwords. As the name also implies, this is not considered safe for production. Here, we declared a user with username = `user`, and password = `password`, stored as a BCrypt hash. For production setups, consider using OpenID Connect IDPs instead.

The `email` and `roles` fields are optional for internal users. However, they will be useful when we want to use SSO with a client application later in this guide.

Expose your authorization server through HTTPProxy

This article assumes that you have completed the previous step in this Getting Started guide. If not, please refer to instructions in Provision an AuthServer.

This article assumes that you are using the TAP-provided Contour ingress. Please refer to instructions in Tanzu Application Platform documentation.

In this tutorial, you are going to:

1. Expose your running authorization server through a `Service` + `HTTPProxy`
2. Ensure it is accessible from outside the cluster

Expose through HTTPProxy

Assuming you deployed an `AuthServer` called `my-authserver-example` in the `default` namespace, expose it by creating a `Service` + `HTTPProxy`.

Note that we used `HTTPProxy.spec.virtualhost.fqdn` = `authserver.example.com`, but you should customize the URL to match the domain of your TAP cluster. The FQDN should match the `issuerURI` that you declared in Provision an AuthServer.

```
---
apiVersion: v1
kind: Service
metadata:
  name: my-authserver-example
  namespace: default
spec:
  selector:
    app.kubernetes.io/part-of: my-authserver-example
    app.kubernetes.io/component: authorization-server
  ports:
    - port: 80
      targetPort: 8080
---
apiVersion: projectcontour.io/v1
kind: HTTPProxy
metadata:
  name: my-authserver-example
  namespace: default
spec:
  virtualhost:
    fqdn: authserver.example.com
  routes:
    - conditions:
        - prefix: /
      services:
        - name: my-authserver-example
          port: 80
```

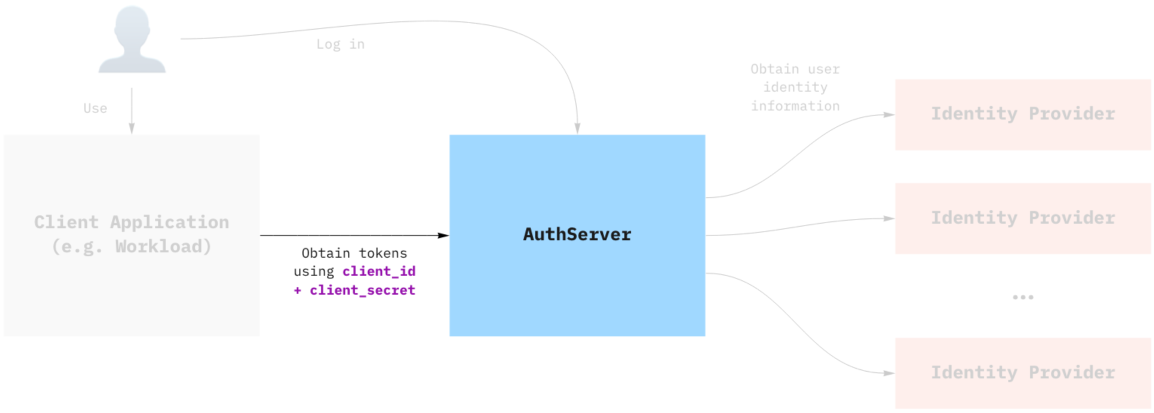
By applying the above resources, your authorization server should become accessible outside the cluster, through http://authserver.example.com.

Provision a client registration

This article assumes that you have completed the previous step in this Getting Started guide. If not, please refer to instructions in Exposing the AuthServer through HTTPProxy.

In this tutorial, you are going to:

1. Obtain credentials for the Authorization Server you have provisioned in Provision your first AuthServer
2. Do a basic check that the credentials are valid using client-credentials flow.


Log in
Use
Obtain user identity information
Identity Provider
Identity Provider
...
Identity Provider
Client Application (e.g. Workload)
AuthServer
Obtain tokens using client_id + client_secret

Creating the ClientRegistration

Assuming you have deployed the AuthServer as described previously, you can create the following client registration:

Note that we used `ClientRegistration.spec.redirectURIs[0]` = `test-app.example.com`, but you should customize the URL to match the domain of your TAP cluster. This will be the URL you use to expose your test application in the next section.

```
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: ClientRegistration
metadata:
   name: my-client-registration
   namespace: default
spec:
   authServerSelector:
      matchLabels:
         name: my-first-auth-server
         env: tutorial
   redirectURIs:
      - "http://test-app.example.com/oauth2/callback"
   requireUserConsent: false
   clientAuthenticationMethod: basic
   authorizationGrantTypes:
      - "client_credentials"
      - "authorization_code"
   scopes:
      - name: "openid"
      - name: "email"
      - name: "profile"
      - name: "roles"
      - name: "message.read"
```

The AuthServer should pick it up. There are two ways to validate this, either by looking at the ClientRegistration `.status` field, or looking at the authserver itself.

```
# Check the client registration
kubectl get clientregistration my-client-registration -n default -o yaml
# Check the authserver
kubectl get authservers
# NAME                    REPLICAS   ISSUER URI                     CLIENTS   TOKEN KE
YS
# my-authserver-example   1          http://authserver.example.com  1         1
#                                                                   ^
#                                 the AuthServer now has one client ^
```

AppSSO will create a secret containing the credentials that client applications will use, named after the client registration. The type of the secret is `servicebindings.io/oauth2`. You can obtain the values in the secret by running:

```
kubectl get secret my-client-registration -n default  -o json | jq ".data | map_values
(@base64d)"
# {
#   "authorization-grant-types": "client_credentials,authorization_code",
#   "client-authentication-method": "basic",
#   "client-id": "default_my-client-registration",
#   "client-secret": "PLACEHOLDER",
#   "issuer-uri": "http://authserver.example.com",
#   "provider": "appsso",
#   "scope": "openid,email,profile,roles,message.read",
#   "type": "oauth2"
# }
```

Validating that the credentials are working

Before you deploy an app and make use of SSO, you can try the credentials from your machine to try and obtain an `access_token` using the `client_credentials` grant. You need the client_id and client_secret that were created as part of the client registration.

```
CLIENT_ID=$(kubectl get secret my-client-registration -n default -o jsonpath="{.data.client-id}" | base64 -d)
CLIENT_SECRET=$(kubectl get secret my-client-registration -n default -o jsonpath="{.data.client-secret}" | base64 -d)
ISSUER_URI=$(kubectl get secret my-client-registration -n default -o jsonpath="{.data.issuer-uri}" | base64 -d)
curl -XPOST "$ISSUER_URI/oauth2/token?grant_type=client_credentials&scope=message.read" -u "$CLIENT_ID:$CLIENT_SECRET"
```

You can decode the `access_token` using an online service, such as JWT.io.

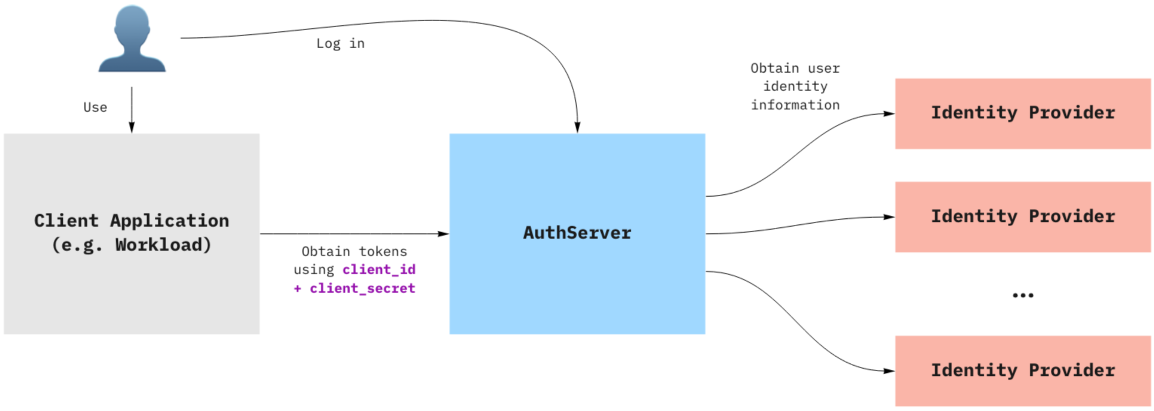
To learn more about grant types, see Grant Types

Deploy an application

This article assumes that you have completed the previous step in this Getting Started guide. If not, please refer to instructions in Provision a client registration.

In this tutorial, you are going to:

1. Deploy a minimal Kubernetes application that uses the credentials created through the ClientRegistration and be protected through SSO.


Log in
Use
Obtain user identity information
Identity Provider
Client Application (e.g. Workload)
Obtain tokens using client_id + client_secret
AuthServer
Identity Provider
...
Identity Provider

Deploy a minimal application

You are going to deploy a two-container pod, as a test application.

Note that we used `HTTPProxy.spec.virtualhost.fqdn` = `test-app.example.com`, but you should customize the URL to match the domain of your TAP cluster. This URL should match what was set up in `ClientRegistration.spec.redirectURIs[0]` in the Previous section

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-application
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      name: test-application
  template:
    metadata:
      labels:
        name: test-application
    spec:
      containers:
        - image: bitnami/oauth2-proxy:7.3.0
          name: proxy
          ports:
            - containerPort: 4180
              name: proxy-port
              protocol: TCP
          env:
            - name: ISSUER_URI
              valueFrom:
                secretKeyRef:
                  name: my-client-registration
                  key: issuer-uri
            - name: CLIENT_ID
              valueFrom:
                secretKeyRef:
                  name: my-client-registration
                  key: client-id
            - name: CLIENT_SECRET
              valueFrom:
                secretKeyRef:
                  name: my-client-registration
                  key: client-secret
          command: [ "oauth2-proxy" ]
          args:
            - --oidc-issuer-url=$(ISSUER_URI)
            - --client-id=$(CLIENT_ID)
            - --insecure-oidc-skip-issuer-verification=true
            - --client-secret=$(CLIENT_SECRET)
            - --cookie-secret=0000000000000000
            - --cookie-secure=false
            - --http-address=http://:4180
            - --provider=oidc
            - --scope=openid email profile roles
            - --email-domain=*
            - --insecure-oidc-allow-unverified-email=true
            - --oidc-groups-claim=roles
            - --upstream=http://127.0.0.1:8000
            - --redirect-url=http://test-app.example.com/oauth2/callback
            - --skip-provider-button=true
            - --pass-authorization-header=true
            - --prefer-email-to-user=true
        - image: python:3.9
          name: application
          resources:
            limits:
              cpu: 100m
              memory: 100Mi
          command: [ "python" ]
          args:
            - -c
            - |
              from http.server import HTTPServer, BaseHTTPRequestHandler
              import base64
              import json

              class Handler(BaseHTTPRequestHandler):
                  def do_GET(self):
                      if self.path == "/token":
                          self.token()
                          return
                      else:
                          self.greet()
                          return

                  def greet(self):
                      username = self.headers.get("x-forwarded-user")
                      self.send_response(200)
                      self.send_header("Content-type", "text/html")
                      self.end_headers()
                      page = f"""
                      <h1>It Works!</h1>
                      <p>You are logged in as <b>{username}</b></p>
                      """
                      self.wfile.write(page.encode("utf-8"))

                  def token(self):
                      token = self.headers.get("Authorization").split("Bearer ")[-1]
                      payload = token.split(".")[1]
                      decoded = base64.b64decode(bytes(payload, "utf-8") + b'==').deco
de("utf-8")
                      self.send_response(200)
                      self.send_header("Content-type", "application/json")
                      self.end_headers()
                      self.wfile.write(decoded.encode("utf-8"))

              server_address = ('', 8000)
              httpd = HTTPServer(server_address, Handler)
              httpd.serve_forever()
---
apiVersion: v1
kind: Service
metadata:
  name: test-application
  namespace: default
spec:
  ports:
    - port: 80
      targetPort: 4180
  selector:
    name: test-application
---
apiVersion: projectcontour.io/v1
kind: HTTPProxy
metadata:
  name: test-application
  namespace: default
spec:
  virtualhost:
    fqdn: test-app.example.com
  routes:
    - conditions:
        - prefix: /
      services:
        - name: test-application
          port: 80
```

Now you can navigate to http://test-app.example.com/. It may ask you to log into the AuthServer you haven't already. You can also navigate to http://test-app.example.com/token if you wish to see the contents of the ID token.

Deployment manifest explained

The application was deployed as a two-container pod: one for the app, and one for handling login.

- The main container is called `application`, and runs a bare-bones Python HTTP server, that reads from the `Authorization` header from incoming requests and returns the decoded `id_token`.
- The second container, called `proxy`, is a sidecar container, an "Ambassador". It receives traffic for the Pod, performs OpenID authentication using OAuth2 Proxy, and proxies requests to the `application` with some added headers containing identity information.

Along with this deployment, there is a `Service` + `HTTPProxy`, to expose the application to the outside world.

Notes on OAuth2-Proxy

The setup of the above OAuth2 Proxy is minimal, and is not considered suitable for production use. To configure it for production, please refer to the official documentation.

Note that OAuth2 Proxy requires some claims to be present in the `id_token`, notably the `email` claim and the non-standard `groups` claim. The `groups` claim maps to AppSSO's `roles` claim. Therefore, for this proxy to work with AppSSO, users *MUST* have an e-mail defined, and at least one entry in `roles`. If the proxy container logs an error stating `Error redeeming code during OAuth2 callback: could not get claim "groups" [...]`, make sure that the user has `roles` provided in the `identityProvider`.

AppSSO for Platform Operators

Learn how to manage the AppSSO package installation and what it installs.

- Installation
- Uninstallation
- Upgrades
- RBAC

Installing AppSSO on TAP

What’s inside

The AppSSO package will install the following resources:

- The `appsso` Namespace with a Deployment of the AppSSO operator and Services for Webhooks
- A `ServiceAccount` with RBAC outlined in detail here
- `AuthServer` and `ClientRegistration` CRDs

Prerequisites

> If you are already running TAP with `run`, `iterate`, or `full` profiles, AppSSO is installed automatically, and you may **skip** the instructions below.

Before installing AppSSO, please ensure you have Tanzu Application Platform v1.2.0 installed on your Kubernetes cluster.

Installation

1. Learn more about the AppSSO package:

   ```
   tanzu package available get sso.apps.tanzu.vmware.com --namespace tap-install
   ```

2. Install the AppSSO package:

   ```
   tanzu package install appsso \
    --namespace tap-install \
    --package-name sso.apps.tanzu.vmware.com \
    --version 1.0.0
   ```

3. Confirm the package has reconciled successfully:

   ```
   tanzu package installed get appsso --namespace tap-install
   ```

Uninstalling AppSSO from TAP

Uninstall the AppSSO package and repository following resource naming introduced in the Installation section:

```
# Delete the Package
tanzu package installed delete appsso \
  --yes --namespace tap-install

# Delete the PackageRepository
tanzu package repository delete appsso-package-repository \
  --yes --namespace tap-install

# Delete the TanzuNet credentials secret
tanzu secret registry delete appsso-registry --yes
```

RBAC

The AppSSO package aggregates the following permissions into TAP's well-known roles:

- app-operator

  ```
  - apiGroups:
    - sso.apps.tanzu.vmware.com
  resources:
    - clientregistrations
  verbs:
    - "*"
  ```

- app-viewer

  ```
  - apiGroups:
    - sso.apps.tanzu.vmware.com
  resources:
    - clientregistrations
  verbs:
    - get
    - list
    - watch
  ```

For the purpose of managing the life cycle of AppSSO CRDs the AppSSO operator's `ServiceAccount` has a `ClusterRole` with the following permissions:

```
- apiGroups:
    - sso.apps.tanzu.vmware.com
  resources:
    - authservers
  verbs:
    - get
    - list
```

```
    - watch
- apiGroups:
    - sso.apps.tanzu.vmware.com
  resources:
    - authservers/status
  verbs:
    - patch
    - update
- apiGroups:
    - sso.apps.tanzu.vmware.com
  resources:
    - clientregistrations
  verbs:
    - get
    - list
    - watch
- apiGroups:
    - sso.apps.tanzu.vmware.com
  resources:
    - clientregistrations/status
  verbs:
    - patch
    - update
- apiGroups:
    - ""
  resources:
    - secrets
    - configmaps
    - services
    - serviceaccounts
  verbs:
    - "*"
- apiGroups:
    - apps
  resources:
    - deployments
  verbs:
    - "*"
- apiGroups:
    - rbac.authorization.k8s.io
  resources:
    - roles
    - rolebindings
  verbs:
    - "*"
- apiGroups:
    - cert-manager.io
  resources:
    - certificates
    - issuers
  verbs:
    - "*"
- apiGroups:
    - ""
  resources:
    - events
  verbs:
    - create
    - update
    - patch
- apiGroups:
    - coordination.k8s.io
  resources:
    - leases
  verbs:
    - create
    - get
    - update
```

AppSSO for Service Operators

`AuthServer` represents the request for an OIDC authorization server. It results in the deployment of an authorization server backed by Redis over mTLS.

You can configure the labels with which clients can select an `AuthServer`, the namespaces it allows clients from, its issuer URI, its token signature keys, identity providers and further details for its deployment.

For the full available configuration, `spec` and `status` see the API reference.

The following sections outline the essential steps to configure a fully operational authorization server.

- Annotations & Labels
- Issuer URI
- Token signature
- Identity providers
- Readiness
- Scale
- Troubleshooting
- Known limitation

Annotation & labels

An `AuthServer` is selectable by `ClientRegistration` through labels. The namespace an `AuthServer` allows `ClientRegistrations` from is controlled with an annotation.

Labels

`ClientRegistrations` select an `AuthServer` with `spec.authServerSelector`. Therefore, an `AuthServer` must have a set of labels that uniquely identifies it amongst all `AuthServer`. Clients won't be able to register if they match no or too many `AuthServer`.

For example:

```
---
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
metadata:
  labels:
    env: dev
    ldap: True
    saml: True
# ...
---
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
metadata:
  labels:
    env: prod
    saml: True
# ...
```

Allowing client namespaces

`AuthServer` controls which namespace it allows `ClientRegistrations` with the annotation:

```
---
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
metadata:
  annotations:
    sso.apps.tanzu.vmware.com/allow-client-namespaces: "*"
```

To allow `ClientRegistrations` from all or a restricted set of Namespaces this annotation must be set. Its value is a comma-separated list of allowed Namespaces, e.g. `"app-team-red,app-team-green"`, or `"*"` if it should allow clients from all namespaces.

⚠ If the annotation is missing, no clients are allowed.

Unsafe configuration

`AuthServer` is designed to enforce secure and production-ready configuration. However, sometimes it is necessary to opt-out of those constraints, e.g. when deploying `AuthServer` on an *iterate* cluster.

> *WARNING:* Allowing **unsafe** is not recommended for production!

Unsafe identity provider

It's not possible to use an `InternalUnsafe` identity provider, unless it's explicitly allowed by including the annotation `sso.apps.tanzu.vmware.com/allow-unsafe-identity-provider` like so:

```
---
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
metadata:
  annotations:
    sso.apps.tanzu.vmware.com/allow-unsafe-identity-provider: ""
spec:
  identityProviders:
    - name: static-users
      internalUnsafe:
      # ...
```

If the annotation is not present and an `InternalUnsafe` identity provider is configured the `AuthServer` will not apply.

Unsafe issuer URI

It's not possible to use a plain HTTP issuer URI, unless it's explicitly allowed by including the annotation `sso.apps.tanzu.vmware.com/allow-unsafe-issuer-uri` like so:

```yaml
---
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
metadata:
  annotations:
    sso.apps.tanzu.vmware.com/allow-unsafe-issuer-uri: ""
spec:
  issuerURI: http://this.is.unsafe
```

If the annotation is not present and a plain HTTP issuer URI configured the `AuthServer` will not apply.

Issuer URI

Before you can apply an `AuthServer` you need an issuer URI. This issuer URI is the entry point for its clients and their end-users. It needs to be reachable by clients, end-users and the AppSSO operator. Therefore, we need to configure a `Service` and a form of ingress for the `AuthServer` to receive traffic.

It is essential to configure Ingress with HTTPS. An authorization server is a critical piece of your security. Using plain HTTP is discouraged. Refer to External access with TLS section for more details on securing traffic.

> This section benefits from your input. Please, share feedback in our Slack channel #app-sso.

Configure a Service for AuthServer

⚠ If you are deploying your `Service` with kapp make sure to set the annotation `kapp.k14s.io/disable-default-label-scoping-rules: ""` to avoid that kapp amends `Service.spec.selector`.

To create a `Service` for an `AuthServer` it must select the authorization server's `Deployment` and configure ports as follows:

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: my-authserver # please, edit
  namespace: authservers # please, edit
  annotations:
    kapp.k14s.io/disable-default-label-scoping-rules: ""
spec:
  type: NodePort
  selector:
    app.kubernetes.io/part-of: my-authserver # replace this with your AuthServer's nam
e
    app.kubernetes.io/component: authorization-server
  ports:
    - port: 80
      targetPort: 8080
```

Once you have configured ingress with HTTPS for this `Service` you should have an issuer URI you can use for your `Authserver`:

```
spec:
  issuerURI: https://my-authserver.my-domain
```

Note: This issuerURI that you add to the `Authserver` config must be the same as `Service`'s `Ingress` or `LoadBalancer` you configured.

If everything goes well, the `IssuerURIReady` condition in `AuthServer.status.conditions` will have `status: "True"`. If not, it will tell you why.

If you need to configure a plain HTTP issuer URI, see unsafe configuration

Enabling external access with TLS

This section will step you through the process of enabling TLS on your authorization server using LetsEncrypt.

The following guide should be used as an example of an approach you can take, and not necessarily *the* only approach that may be feasible.

The following guide has been verified using *Amazon Elastic Kubernetes Service*(EKS) and *Google Kubernetes Engine* (GKE).

Prerequisites

You must have already created a `Service` resource for your authorization server.

This guide will be based on the following example prerequisite values (your values will differ):

TAP values contains the following fields:

- `shared.ingress_domain: example.com`
- `contour.envoy.service.type: LoadBalancer`

`AuthServer` custom resource contains the following:

- `.metadata.name: my-auth-server`
- `.metadata.namespace: authservers`
- `.spec.issuerUri: https://login.example.com` <- this is the URL desired for this auth server

Guide

Create a `ClusterIssuer` resource. This will be the Certificate issuer authority; in this case, the issuer will be LetsEncrypt ‘staging’.

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: appsso-letsencrypt-staging
spec:
  acme:
    email: <SERVICE-OPERATOR-EMAIL-ADDRESS> # please, edit
    privateKeySecretRef:
      name: appsso-letsencrypt-staging      # This will be auto-generated for you when
this resource is created
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    solvers:
      - http01:
          ingress:
            class: contour
```

Create a `Certificate` resource. This will be the TLS certificate that will be attached to the authorization server subdomain.

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: appsso-my-authserver
  namespace: authservers
spec:
  commonName: login.example.com    # Please, edit; this should be your Issuer URI, wit
hout https prefix
  dnsNames:
    - login.example.com           # This should be your Issuer URI, without https pref
ix
  issuerRef:
    name: appsso-letsencrypt-staging # This is the name of the ClusterIssuer from abov
e
    kind: ClusterIssuer
  secretName: appsso-my-authserver   # This secret will be created by this Certificat
e, just give it a good name
```

Wait for the `Certificate` to be issued:

```
kubectl get certificate appsso-my-authserver --namespace authservers -o wide --watch
```

It should eventually look like:

```
NAME                   READY   SECRET                 ISSUER                        STA
TUS                                         AGE
appsso-my-authserver   True    appsso-my-authserver   appsso-letsencrypt-staging   Cer
tificate is up to date and has not expired   45s
```

Create an `HTTPProxy` resource. This resource will map the Issuer URI subdomain to the authorization server Service and apply a TLS certificate.

```
apiVersion: projectcontour.io/v1
kind: HTTPProxy
```

```
metadata:
  name: appsso-my-authserver
  namespace: authservers
spec:
  virtualhost:
    fqdn: login.example.com           # This should be your Issuer URI, without https
prefix
    tls:
      secretName: appsso-my-authserver   # This is the Secret that was created by the
Certificate
  routes:
    - conditions:
        - prefix: /
      services:
        - name: my-authserver # please, edit; this is the name of the Service from abo
ve
          port: 80
```

Wait until `Status` becomes `valid`:

```
kubectl get httpproxy appsso-my-authserver --namespace authservers --watch
```

Once reconciled, the authorization server should be available at: `https://login.example.com`

⚠ LetsEncrypt *staging* does not provide a *trusted* level certificate verification and so your browser will not trust the certificate, however you may still continue using the authorization server for testing purposes.

Use LetsEncrypt *production* issuer to certify trusted certificates; be aware of API rate limits. To use LetsEncrypt production, change the field `.spec.acme.server` to `https://acme-v02.api.letsencrypt.org/directory` in your `ClusterIssuer`

Perform a readiness check, by querying the OpenID Connect discovery endpoint:

```
curl --insecure https://login.example.com/.well-known/openid-configuration
```

You should receive a `200 OK` JSON response with authorization server information.

Further reading

This guide relies on the following sources:

- Deploying HTTPS services with Contour and cert-manager
- httpproxy Documentation

Identity providers

An `AuthServer` does not manage users internally. Instead, users log in through external identity providers (IdPs). Currently, `AuthServer` supports OpenID Connect providers, as well a list of “static” hard-coded users for development purposes. `AuthServer` also has limited, experimental support for LDAP and SAML providers.

Identity providers are configured under `spec.identityProviders`, learn more from the API reference.

⚠ Changes to `spec.identityProviders` take some time to be effective as the operator will roll out a new deployment of the authorization server.

End-users will be able to log in with these providers when they go to `{spec.issuerURI}` in their browser.

Learn how to configure identity providers for an `AuthServer`:

- OpenID Connect providers
- LDAP (experimental)
- SAML (experimental)
- Internal, static user
- Restrictions

OpenID Connect providers

To set up an OpenID Connect provider, provide the following information for your `AuthServer`:

```
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
metadata:
# ...
spec:
  identityProviders:
    - name: my-oidc-provider
      openID:
        # REQUIRED
        # The issuer identifier. If the provider supports OpenID Connect Discovery,
        # this value will be used to auto-configure the provider, by obtaining informa
tion
        # at https://issuer-uri/.well-known/openid-configuration
        issuerURI: https://openid.example.com
        # Obtained when registering a client with the provider, often through a web UI
        clientID: my-client-abcdef
        # Obtained when registering a client with the provider, often through a web UI
        clientSecretRef:
          name: my-openid-client-secret
        # The URI for performing an authorization request and obtaining an authorizati
on_code
        authorizationUri: https://example.com/oauth2/authorize
        # The URI for performing a token request, and obtaining a token
        tokenUri: https://example.com/oauth2/token
        # The JWKS endpoint for obtaining the JSON Web Keys, used to verify token sign
atures
        jwksUri: https://example.com/oauth2/jwks
        # Scopes used in the authorization request
        # MUST contain "openid". Other common OpenID values are "profile", "email".
        scopes:
          - "openid"
          - "other-scope"
        # OPTIONAL
        claimMappings:
          # The "my-oidc-provider-groups" claim from the ID token issued by "my-oidc-p
rovider"
          # will be mapped into the "roles" claim in tokens issued by AppSSO
          roles: my-oidc-provider-groups
  # ...
---
apiVersion: v1
kind: Secret
metadata:
  name: my-openid-client-secret
  # ...
stringData:
  clientSecret: very-secr3t
```

It is essential that `openID.clientSecretRef` is a `Secret` with the entry `clientSecret`.

You can define as many OpenID providers as you like.

Verify the configuration by visiting the `AuthServer`'s issuer URI in your browser and select `my-oidc-provider`.

Note for registering a client with the identity provider

The `AuthServer` will set up redirect URIs based on the provider name in the configuration. For example, for a provider with `name: my-provider`, the redirect URI will be `{spec.issuerURI}/login/oauth2/code/my-provider`. The externally accessible user URI for the `AuthServer`, including scheme and port is `spec.issuerURI`. If the `AuthServer` is accessible on `https://appsso.company.example.com:1234/`, the redirect URI registered with the identity provider should be `https://appsso.company.example.com:1234/login/oauth2/code/my-provider`.

LDAP (experimental)

WARNING: Support for LDAP providers is considered “experimental”.

At most one `ldap` identity provider can be configured.

For example:

```
---
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
metadata:
# ...
spec:
  identityProviders:
    - name: ldap
      ldap:
        server:
          scheme: ldap
          host: my-ldap.com
          port: 389
          base: ""
        bind:
          dn: uid=binduser,ou=Users,o=5d03d6ac6eed091436a8d664,dc=jumpcloud,dc=com
          passwordRef:
            name: ldap-password
        user:
          searchFilter: uid={0}
          searchBase: ou=Users,o=5d03d6ac6eed091436a8d664,dc=jumpcloud,dc=com
        group:
          searchFilter: member={0}
          searchBase: ou=Users,o=5d03d6ac6eed091436a8d664,dc=jumpcloud,dc=com
          searchSubTree: true
          searchDepth: 10
          roleAttribute: cn
  # ...
---
apiVersion: v1
kind: Secret
metadata:
  name: ldap-password
  namespace: default
stringData:
  password: very-z3cret
```

It is essential that `ldap.bind.passwordRef` is a `Secret` with the entry `password`.

Verify the configuration by visiting the `AuthServer`'s issuer URI in your browser and select `my-oidc-provider`.

SAML (experimental)

WARNING: Support for SAML providers is considered "experimental".

For SAML providers only autoconfiguration through `metadataURI` is supported.

```
---
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
metadata:
# ...
spec:
  - name: my-saml-provider
    saml:
      metadataURI: https://saml.example.com/sso/saml/metadata # required
      claimMappings: # optional
        # Map SAML attributes into claims in id_tokens issued by AppSSO. The key
        # on the left represents the claim, the value on the right the attribute.
        # For example:
        # The "saml-groups" attribute from the assertion issued by "my-saml-provider"
        # will be mapped into the "roles" claim in id_tokens issued by AppSSO
        roles: saml-groups
        givenName: FirstName
        familyName: LastName
        emailAddress: email
```

Note for registering a client with the identity provider

The `AuthServer` will set up SSO and metadata URLs based on the provider name in the configuration. For example, for a SAML provider with `name: my-provider`, the SSO URL will be `{spec.issuerURI}/login/saml2/sso/my-provider`. The metadata URL will be {spec.issuerURI}/saml2/service-provider-metadata/my-provider. spec.issuerURI is the externally accessible issuer URI for an AuthServer, including scheme and port. If the AuthServer is accessible on https://appsso.company.example.com:1234/, the SSO URL registered with the identity provider should be https://appsso.company.example.com:1234/login/saml2/sso/my-provider.

Internal users

> *WARNING:* InternalUnsafe considered **unsafe**, and **not** recommended for production!

During development, static users may be useful for testing purposes. **At most one** internalUnsafe identity provider can be configured.

For example:

```
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
metadata:
  annotations:
    sso.apps.tanzu.vmware.com/allow-unsafe-identity-provider: ""
  # ...
spec:
  identityProviders:
    - name: test-users
      internalUnsafe:
        users:
          - username: ernie
            password: "$2a$10$201z9o/tHlocFsHFTo0plukh03ApBYe4dRiXcqeyRQH6CNNtS8jWK" # bcrypt-hashed "password"
            roles:
              - "silly"
          - username: bert
            password: "$2a$10$201z9o/tHlocFsHFTo0plukh03ApBYe4dRiXcqeyRQH6CNNtS8jWK" # bcrypt-hashed "password"
            roles:
              - "grumpy"
  # ...
```

InternalUnsafe needs to be explicitly allowed by setting the annotation sso.apps.tanzu.vmware.com/allow-unsafe-identity-provider: "".

It is important that password is bcrypt-hashed (see below).

Verify the configuration by visiting the AuthServer's issuer URI in your browser and logging in as ernie/password.

Generating a bcrypt hash from a plain-text password

There are multiple options for generating bcrypt hashes:

1. Use an online bcrypt generator
2. On Unix platforms, use htpasswd. Note, you may need to install it, for example on Ubuntu by running apt install apache2-utils

```
htpasswd -bnBC 12 "" your-password-here | tr -d ':\n'
```

Restrictions

Each identity provider has a declared `name`. The following conditions apply:

- the names must be unique
- the names must not be blank
- the names must follow Kubernetes' DNS Subdomain Names guidelines
    - contain no more than 253 characters
    - contain only lowercase alphanumeric characters, '-' or '.'
    - start with an alphanumeric character
    - end with an alphanumeric character
- the names may not start with `client` or `unknown`

There can be at most one of each `internalUnsafe` and `ldap`.

Token signature

An `AuthServer` must have token signature keys configured to be able to mint tokens.

Learn about token signatures and how to manage keys of an `AuthServer`:

- Token signature 101
- Token signature in AppSSO
- Creating keys
- Rotating keys
- Revoking keys

"Token signature key" or just "key" is AppSSO's wording for a public/private key pair that is tasked with signing and verifying JSON Web Tokens (JWTs). For more information, please refer to the following resources:

- JSON Web Token (JWT) spec
- JSON Web Signature (JWS) spec

Token signature 101

Token signature keys are used by an `AuthServer` to sign JSON Web Tokens (JWTs) - producing a JWS Signature and attaching it to the JOSE Header of a JWT. The client application later is able to verify the JWT signature. A private key is used to sign a JWT, and a public key is used to verify the signature of a signed JWT.

The sign-and-verify mechanism serves multiple security purposes:

- **Authenticity**: signature verification ensures that the issuer of the JWT is from a source that is advertised.

- **Integrity**: signature verification ensures that the JWT has not been altered in transit or during its issued lifetime. Integrity is a foundational pillar of the CIA triad concept in Information Security.
- **Non-repudiation**: signature verification ensures that the authorization server that signed the JWT cannot deny that they have signed it after its issuance (granted that the signing key that signed the JWT is available).

Token signature of an `AuthServer`

An `AuthServer` receives its keys under `spec.tokenSignature`, e.g.:

```
spec:
  tokenSignature:
    signAndVerifyKeyRef:
      name: sample-token-signing-key
    extraVerifyKeyRefs:
      - name: sample-token-verification-key-1
      - name: sample-token-verification-key-2
```

There can only be **one** token signing key `spec.tokenSignature.signAngVerifyKeyRef` at any given time, and arbitrarily many token verification keys `spec.tokenSignature.extraVerifyKeyRefs`. The token signing key is used to sign and verify actively issued JWTs in circulation, whereas token verification keys are used to verify issued JWTs signatures. Token verification keys are thought to be previous token signing keys but have been rotated into verify only mode as a rotation mechanism measure, and can potentially be slated for eviction at a predetermined time.

As per OAuth2 spec, `AuthServer` serves its public keys at `{spec.issuerURI}/oauth2/jwks`. For example:

```
❯ curl -s authserver-sample.default/oauth2/jwks | jq
{
  "keys": [
    {
      "kty": "RSA",
      "e": "AQAB",
      "kid": "sample-token-signing-key",
      "n": "0iCinir7sWKZE_3QXq4eTub_GU-lvdAKFI9dzDlwX7XZwwSERuzzQQ_Fs7i9djMl5bpv2ma_3Z
B-j2W9pR9ZIa3nqBI29AHqx2zmVQ8w-GxPDGRMkBdMOWNwyDQGIRlQnJFpXRoSQ5_viM9gYA56WthkDghrupGU
iB_zqGFYlgnz7sd4lC-thgEkDi9vY68DLIFdsXOQIXFqakyEIo43n_0vg6JRGQW1LU_32Ok6OgA3r6bYcE8VQh
JW3sE1qOSFcP0JrPA3YgmTNuDV6GoCLZeMxDdMDKdDcH5UgERLQe1qMMKwlMCeKamOWgo9eBvcFnWNR0I_MJV6
F14U1WbIcQ"
    },
    {
      "kty": "RSA",
      "e": "AQAB",
      "kid": "sample-token-verification-key-1",
      "n": "wc7uOACU62Yu_zKT9YrI4v-_X3L47nbVlcByi4UTVhg8o001OkiYAPAEoDCEHnDg_54gTWxe3h
DRcOJrd72PkTAaxH8aFdikoyakRVG9NvAPbcfzvI8R8plepUbs1U7TPPDEDARm_fZX6QdVyz0CTSafrz-yktTA
DxJhYPgvFLeHq7g7RouB1szTWDCM1haoxKa4960_x9meghNn87z0uF3cAd7TM_k3capYnxNOUT5g1vjJ05Vk14
JUl4R294OpMXPCGcFuvu9auXeBqXyKxxTAnLkDdNrgtT0FJHwnh4RGnrNqjYZOwlRvGbzwQ7du97aU2-qgbKkJ
rWYZWcw2bQ"
    },
    {
      "kty": "RSA",
      "e": "AQAB",
      "kid": "sample-token-verification-key-2",
      "n": "qELrLiaD-IVp_nthVn2EsLuShtU9ovyVIPkLVf47AqKogPV2frE_6Sv8k7Zim-SgDXfjLEg-UG
lQrb4KFm_WkaK2Uf6PCapiBnMi1Q5P8qC0WC5LT6XyPY1exCQbMrEsyd89oS0sKxgoc3Qv0XV24jGYiWQyJ7I0
Rub_QEldGM_dSlfbI-1Qt_U6Ll22OEc1D6P1A3MdDrgbur6N7ZemxlKI26-OAdlbNi0u-lFNj3Ss-pfTVi_fD2
hAajRRmc4tmHejQjH36M4F1NSW_gTbb6VX5EerVuDwSCCK0EuGvhcb1hg6kYEoO-qws54AQ0PywBXT5qksCMBm
mzjP6qO4Ow"
    }
  ]
}
```

⚠ Changes to `spec.tokenSignature.signAngVerifyKeyRef` have immediate effect.

As a *service operator*, you have control over which keys are used for certain purposes. Navigate to the next few sections for more information.

Creating keys

You can deploy an `AuthServer` without `spec.tokenSignature` but it won't be able to mint tokens. Therefore, keys must be configured to make it fully operational. The following describe how to create and apply a keys for an `AuthServer`.

An RSA key can be created multiple ways. Below are two recommended approaches – choose one.

Using secretgen-controller

NOTE: This section assumes you have TAP running in your cluster, with `secretgen-controller` installed.

An `RSAKey` CR allows for expedited creation of a Secret resource containing PEM-encoded public and private keys required by an `AuthServer`.

1. Create an `AuthServer` with `RSAKeys` as follows:

   ```
   apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
   kind: AuthServer
   metadata:
    name: authserver-sample
    namespace: default
   spec:
    tokenSignature:
      signAndVerifyKeyRef:
        name: my-token-signing-key
      extraVerifyKeyRefs:
        - name: my-token-verification-key
    # ...
   ---
   apiVersion: secretgen.k14s.io/v1alpha1
   kind: RSAKey
   metadata:
    name: my-token-signing-key
    namespace: default
   spec:
    secretTemplate:
      type: Opaque
   ```

```
   stringData:
     key.pem: $(privateKey)
     pub.pem: $(publicKey)
---
apiVersion: secretgen.k14s.io/v1alpha1
kind: RSAKey
metadata:
 name: my-token-verification-key
 namespace: default
spec:
 secretTemplate:
   type: Opaque
   stringData:
     key.pem: $(privateKey)
     pub.pem: $(publicKey)
```

2. Observe the creation of an underlying `Secrets`. The name of the each `Secret` is the same as the `RSAKey` names:

   ```
   # Verify Secret exists
   kubectl get secret my-token-signing-key

   # View the base64-encoded keys
   kubectl get secret my-token-signing-key -o jsonpath='{.data}'
   ```

   You should be able to see two fields within the Secret resource: `key.pem` (private key) and `pub.pem` (public key).

3. Verify that the `AuthServer` serves its keys

   ```
   curl -s authserver-sample.default/oauth2/jwks | jq
   ```

If you encounter any issues with this approach, be sure to check out Carvel Secretgen Controller documentation

Using OpenSSL

You can generate an RSA key yourself using OpenSSL. Here are the steps:

1. Generate a PEM-encoded RSA key pair

   This guide references the freely published OpenSSL Cookbook and the approaches mentioned therein around generating a public and private key pair.

   ```
   # Generate an 4096-bit RSA key
   openssl genpkey -out privatekey.pem -algorithm RSA -pkeyopt rsa_keygen_bits:4096
   # -> privatekey.pem
   # The resulting private key output is in the PKCS#8 format

   # Next, extract the public key
   openssl pkey -in privatekey.pem -pubout -out publickey.pem
   # -> publickey.pem
   # The resulting public key output is in the PKCS#8 format
   ```

```
# To view details of the private key
openssl pkey -in privatekey.pem -text -noout
```

More OpenSSL key generation examples here.

2. Create a Secret resource in sso4k8s namespace using key generated from previous step:

```
# Base64 encode the key files
cat privatekey.pem | base64 > privatekey-base64.pem
cat publickey.pem | base64 > publickey-base64.pem

# Create Secret resource
kubectl create secret generic my-key \
--from-file=key.pem=privatekey-base64.pem \
--from-file=pub.pem=publickey-base64.pem \
--namespace sso4k8s
```

3. Apply your `AuthServer`:

```
  apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
  kind: AuthServer
  metadata:
    name: authserver-sample
    namespace: default
  spec:
    tokenSignature:
      signAndVerifyKeyRef:
        name: my-key
    # ...
```

4. Verify that the `AuthServer` serves its keys

```
curl -s authserver-sample.default/oauth2/jwks | jq
```

Rotating keys

This section describes how to “rotate” token signature keys for an `AuthServer`.

The action of “rotating” means moving the active token signing key into the set of token verification keys, generating a new cryptographic key, and assigning it to be the designated token signing key.

Assuming that you have an `AuthServer` with token signature keys configured, rotate keys as follows:

1. Generate a new token signing key first. See creating keys. Verify that the new `Secret` exists before proceeding to the next step.

2. Edit `AuthServer.spec.tokenSignature`, append the existing `spec.tokenSignature.signAndVerifyKeyRef` to `spec.tokenSignature.extraVerifyKeys` and set your new key as `spec.tokenSignature.signAndVerifyKeyRef`.

   For example:

```
# Before
---
```

```
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
metadata:
 name: authserver-sample
 namespace: default
spec:
 tokenSignature:
   signAndVerifyKeyRef:
     name: old-key
   extraVerifyKeys: []
 # ...
```

```
# After
---
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
metadata:
 name: authserver-sample
 namespace: default
spec:
 tokenSignature:
   signAndVerifyKeyRef:
     name: new-key
   extraVerifyKeys:
     - name: old-key
 # ...
```

Once you apply your changes, key rotation is effective immediately.

Moving the active token signing key to be a token verification key is an *optional* step – check out the Revoking keys section for more.

Revoking keys

This section describes how to “revoke” token signature keys for an `AuthServer`.

The action of “revoking” a key means to entirely remove the key from circulation by an `AuthServer`, whether it be a token signing key or a token verification key. This action might be needed if your organization requires a complete key refresh where older keys are never retained. Another scenario might be in the case of an emergency in which a key or a session has been compromised and a complete revocation is warranted.

To revoke an existing key or keys, you may remove any references to the keys in the `spec.tokenSignature` resource. By removing the reference to the key, the system shall no longer acknowledge that the key is used for signing or verifying JWTs.

For example, if you have a token signing key and a few verification keys:

```
---
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
metadata:
  name: authserver-sample
  namespace: default
spec:
  tokenSignature:
    signAndVerifyKeyRef:
      name: key-3
    extraVerifyKeys:
      - name: key-2
      - name: key-1
  # ...
```

To revoke an existing verification key, remove it from the `extraVerifyKeys` list. In the example above, you can remove “key-2” and “key-1” from the list; JWTs signed with those keys will no longer be verifiable.

To revoke an existing token signing key, remove it from `signAndVerifyKeyRef` field. However, if you remove an existing token signing key without a replacement key, the `AuthServer` will not be able to issue access tokens until a valid token signing key is provided. In the example above, “key-3” would be removed; the system will not be able to sign or verify JWTs.

References and further reading

- JSON Web Token (JWT) - rfc7519 (ietf.org)
- JSON Web Signature (JWS) - rfc7515 (ietf.org)

Readiness

Generally, `AuthServer.status` is a reliable source to judge an `AuthServer`’s readiness.

However, you are encouraged to verify your `AuthServer` with the following checks:

- [ ] Ensure that there is at least one token signing key configured

  ```
  curl -X GET {spec.issuerURI}/oauth2/jwks
  ```

  The response body should yield at least one key in the list. If there are no keys, please apply a token signing key

- [ ] Ensure that OpenID discovery endpoint is available

  ```
  curl -X GET {spec.issuerURI}/.well-known/openid-configuration
  ```

  The response body should yield a valid JSON body containing information about the `AuthServer`.

Client registration check

It is helpful to verify an `AuthServer` by executing a test run with a test `ClientRegistration`. This check also ensures that app developers will also be able to register clients with the `AuthServer` successfully.

Follow the steps below to ensure that your installation can:

1. Add a test client.
2. Get an access token.
3. Invalidate/remove the test client.

Prerequisites

Ensure that you have successfully applied a token signing key to your `AuthServer` before proceeding.

Define and apply a test client

Apply a `ClientRegistration` to your cluster in a Namespace that the `AuthServer` should allow clients from:

```
---
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: ClientRegistration
metadata:
  name: test-client
  namespace: default
spec:
  authServerSelector:
    matchLabels:
    # appropriate labels for your `AuthServer`
  authorizationGrantTypes:
    - client_credentials
  clientAuthenticationMethod: basic
```

Check out the ClientRegistration API reference for more field definitions.

This defines a test `ClientRegistration` with the `client_credentials` OAuth grant type.

Apply the `ClientRegistration`:

```
kubectl apply -f appsso-test-client.yaml
```

Once the `ClientRegistration` is applied, inspects its status and verify it's ready.

Get an access token

You should be able to get a token with the client credentials grant for example:

```
# Get client id (`base64` command has to be available on the command line)
export APPSSO_TEST_CLIENT_ID=$(kubectl get secret test-client -n default -o jsonpath="
{.data['client-id']}" | base64 --decode)

# Get client secret (`base64` command has to be available on the command line)
export APPSSO_TEST_CLIENT_SECRET=$(kubectl get secret test-client -n default -o jsonpa
th="{.data['client-secret']}" | base64 --decode)

# Attempt to fetch access token
curl \
 --request POST \
 --location "{spec.issuerURI}/oauth2/token" \
 --header "Content-Type: application/x-www-form-urlencoded" \
 --header "Accept: application/json" \
 --data "grant_type=client_credentials" \
 --basic \
 --user $APPSSO_TEST_CLIENT_ID:$APPSSO_TEST_CLIENT_SECRET
```

You should see a response JSON containing populated field `access_token`. If so, the system is working as expected, and client registration check is successful.

Make sure to delete the test `ClientRegistration` once you are done.

Scale

The number of authorization server replicas for an `AuthServer` can be specified under `spec.replicas`.

Furthermore, `AuthServer` implements the `scale` subresource. That means you can scale it scale an `AuthServer` with existing tooling. For example:

```
kubectl scale authserver authserver-sample --replicas=3
```

The resource of the authorization server and Redis `Deployments` can be configured under `spec.resources` and `spec.redisResources` respectively. See the API reference for details.

Authorization server audit logs

AppSSO `AuthServer`s do the following:

- Handle user authentication
- Issue `id_token` and `access_token`

Each audit event contains the following:

- `ts` - date/time of the event
- `remoteIpAddress` - the IP of the user-authentication or if not attainable, the IP of the last proxy

Authentication

`AuthServer` produce the following authentication events:

- `AUTHENTICATION_SUCCESS`
  - **Trigger** successful authentication
  - **Data recorded** Username, Provider ID, Provider Type (INTERNAL, OPENID, …)
- `AUTHENTICATION_LOGOUT`
  - **Trigger** successful logout
  - **Data recorded** Username, Provider ID, Provider Type (INTERNAL, OPENID, …)
- `AUTHENTICATION_FAILURE`
  - **Trigger** failed authentication using either `internalUnsafe` or `ldap` identity provider
  - **Data recorded** Username, Provider ID, Provider Type (INTERNAL or LDAP)
- `INVALID_UPSTREAM_PROVIDER_CONFIGURATION`
  - **Trigger** some cases of failed authentication with an `openId` or `saml` identity provider
  - **Data recorded** Provider ID, Provider Type, error

  - **Note** usually followed by a human-readable help message, with `"logger": "appsso.help"`

Token flows

`AuthServer` produce the following authorization_code and token events:

- `AUTHORIZATION_CODE_ISSUED`
  - **Trigger** `authorization_code` grant type, successful call to `/oauth2/authorize`
  - **Data recorded** Username, Provider ID, Provider Type, Client ID, Scopes requested, Redirect URI
- `AUTHORIZATION_CODE_REQUEST_REJECTED`
  - **Trigger** `authorization_code` grant type, unsuccessful call to `/oauth2/authorize`, for example invalid Client ID, invalid Redirect URI, …
  - **Data recorded** Error, Error Code (ex: `invalid_scope`), Client ID, Scopes requested Redirect URI, Username (may be `anonymousUser`), Provider ID and Provider Type if available
- `TOKEN_ISSUED`
  - **Trigger** successful call to `/oauth2/token`
  - **Data recorded** Scopes, Client ID, Grant Type (`authorization_code` or `client_credentials`), Username
- `TOKEN_REQUEST_REJECTED`
  - **Trigger** unsuccessful call to `/oauth2/token`, for example invalid Client Secret
  - **Data recorded** Client ID, Scopes requested, Error

Troubleshooting

Why is my AuthServer not working?

Generally, `AuthServer.status` is designed to provide you with helpful feedback to debug a faulty `AuthServer`.

Find all AuthServer-related Kubernetes resources

All `AuthServer` components can be identified with Kubernetes common labels , e.g.:

```
app.kubernetes.io/part-of: my-authserver
```

Logs of all AuthServers

With stern you can tail the logs of all AppSSO managed `Pods` inside your cluster with:

```
stern --all-namespaces --selector=app.kubernetes.io/managed-by=sso.apps.tanzu.vmware.com
```

Change propagation

When applying changes to an `AuthServer`, keep in mind that changes to issuer URI, IDP, server and logging configuration take a moment to be effective as the operator will roll out the authorization server `Deployment`.

My Service is not selecting the authorization server's Deployment

If you are deploying your `Service` with kapp make sure to set the annotation `kapp.k14s.io/disable-default-label-scoping-rules: ""` to avoid that kapp amends `Service.spec.selector`.

Redirect URIs are redirecting to http instead of https with a non-internal identity provider

Follow this workaround, adding IP ranges for the `AuthServer` to trust.

Known Limitations

As of 1.0.0, the following are known product limitations to be aware of.

Limited number of `ClientRegistrations` per `AuthServer`

The number of `ClientRegistration` for an `AuthServer` is limited at **~2,000**. This is a soft limitation, and if you are attempting to apply more `ClientRegistration` resources than the limit, we cannot guarantee those clients applied past the limit to be in working order. This is subject to change in future product versions.

AppSSO for App Operators

To secure a `Workload` with AppSSO you need a `ClientRegistration` with these ingredients:

- A unique label selector for the `AuthServer` you want to register a client for
- Remaining configuration of your OAuth2 client

Talk to your *Service Operator* to learn which `AuthServers` they are running and which labels you should use. Once you have those labels, you can create a `ClientRegistration` as follows:

```
---
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: ClientRegistration
metadata:
  name: my-client
  namespace: my-team
spec:
  authServerSelector:
    matchLabels: # for example
      env: staging
      ldap: True
      team: my-team
```

Continue with learning how to customize your `ClientRegistration` by securing a `Workload` with SSO.

Learn more about grant types.

Register an app with AppSSO

Topics

- Client registration
- Workloads

Client registration

Applications/Clients must register with AppSSO to allow users to sign in with single sign on within a Kubernetes cluster. This registration will result in the creation of a Kubernetes secret

To do this, apply a `ClientRegistration` to the appropriate Kubernetes cluster.

To confirm that the `ClientRegistration` was successfully processed, check the status:

```
kubectl describe clientregistrations.sso.apps.tanzu.vmware.com <client-name>
```

It is also possible, but not recommended, to register clients statically while deploying AppSSO.

Note: It is recommended to register clients dynamically after AppSSO has been deployed. When registering a client statically, properties cannot be changed without triggering a rollout of AppSSO itself.

Grant Types

Workloads

This guide will walk you through steps necessary to secure your deployed `Workload` with AppSSO.

Prerequisites

Before attempting to integrate your workload with AppSSO, please ensure that the following items are addressed:

- Tanzu Application Platform (TAP) `v1.2.0` or above is available on your cluster.
- Tanzu CLI `v0.11.6` or above is available on your command line.
- AppSSO package `v1.0.0` or above is available on your cluster.

Configuring a Workload with AppSSO

AppSSO and your Workload need to establish a bidirectional relationship: AppSSO is aware of your Workload and your Workload is aware of AppSSO. How does that work?

- To make AppSSO aware of your Workload (i.e. that AppSSO should be responsible for authentication and authorization duties), you have to create and apply a ClientRegistration resource .
- To make your Workload aware of AppSSO (i.e. that your application shall now rely on AppSSO for authentication and authorization requests), you must specify a service resource claim which produces the necessary credentials for your Workload to consume.

The following sections elaborate on both of the concepts in detail.

Create and apply a ClientRegistration resource

Define a ClientRegistration resource for your Workload. Here is an example:

```
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: ClientRegistration
metadata:
  name: my-workload-client-registration
  namespace: my-workload-namespace
spec:
  authServerSelector:
    matchLabels:
    # ask your Service Operator for labels to target an `AuthServer`
  authorizationGrantTypes:
    - client_credentials
    - authorization_code
    - refresh_token
  clientAuthenticationMethod: basic
  requireUserConsent: true
  redirectURIs:
    - "<MY_WORKLOAD_HOSTNAME>/redirect-back-uri"
  scopes:
    - name: openid
```

Once applied successfully, this resource will create the appropriate credentials for your Workload to consume. More on this in the next section.

Please refer to the ClientRegistration custom resource documentation page for additional details on schema and specification of the resource.

Add a service resource claim to your Workload

Once a ClientRegistration resource has been defined, you can now create a service resource claim by using Tanzu CLI:

```
tanzu service claim create my-client-claim \
  --namespace my-workload-namespace \
  --resource-api-version sso.apps.tanzu.vmware.com/v1alpha1 \
  --resource-kind ClientRegistration \
  --resource-name my-workload-client-registration \
  --resource-namespace my-workload-namespace
```

Alternatively, you may create the claim as a `ResourceClaim` custom resource:

```
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ResourceClaim
metadata:
  name: my-client-claim
  namespace: my-workload-namespace
spec:
  ref:
    apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
    kind: ClientRegistration
    name: my-workload-client-registration
    namespace: my-workload-namespace
```

Observe the status of the service resource claim by running `tanzu service claim list -n my-workload-namespace -o wide`:

```
NAMESPACE              NAME             READY   REASON   CLAIM REF
my-workload-namespace  my-client-claim  True             services.apps.tanzu.vmware.com/
v1alpha1:ResourceClaim:my-client-claim
```

The created service resource claim is now referable within a Workload:

```
apiVersion: carto.run/v1alpha1
kind: Workload
metadata:
  labels:
    apps.tanzu.vmware.com/workload-type: web
  name: my-workload
  namespace: my-workload-namespace
spec:
  source:
    git:
      ref:
        branch: main
      url: ssh://git@github.com/my-company/my-workload.git
  serviceClaims:
    - name: my-client
      ref:
        apiVersion: services.apps.tanzu.vmware.com/v1alpha1
        kind: ResourceClaim
        name: my-client-claim
```

Alternatively, you can refer to your `ClientRegistration` when deploying your workload with the `tanzu` CLI. Like so

```
tanzu apps workload create my-workload \
  --service-ref "my-client=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:my-cl
ient-claim" \
  # ...
```

What this service claim reference binding does under the hood is ensures that your Workload's Pod is mounted with a volume containing the necessary credentials required by your application to become aware of AppSSO. Learn more about Service Bindings.

The credentials provided by the service claim are:

- **Client ID** - the identifier of your Workload that AppSSO is registered with. This is a unique identifier.
- **Client Secret** - secret string value used by AppSSO to verify your client during its interactions. Keep this value secret.
- **Issuer URI** - web address of AppSSO, and the primary location that your Workload will go to when interacting with AppSSO.
- **Authorization Grant Types** - list of desired OAuth 2 grant types that your wants to support.
- **Client Authentication Method** - method in which the client application requests an identity or access token
- **Scopes** - list of desired scopes that your application's users will have access to.

The above credentials are mounted onto your Workload's Pod(s) as individual files at the following locations:

```
/bindings
  /<name-of-service-claim>
    /client-id
    /client-secret
    /issuer-uri
    /authorization-grant-types
    /client-authentication-method
    /scope
```

Taking our example from above, the location of credentials can be found at:

```
/bindings/my-client/{client-id,client-secret,issuer-uri,authorization-grant-types,clie
nt-authentication-method,scope}
```

Given these auto-generated values, your Workload is now able to load them at runtime and bind to AppSSO at start-up time. Reading the values from the file system is left to the implementor as to the approach taken.

Grant types

These are the grant types/flows for apps to get an access token on behalf of a user. If not included, the default will be `['client_credentials']`. They take effect by being included in the `authorizationGrantTypes` property list in the Client Registration.

To register a client/application, apply the `yaml` with your specifications to your cluster `kubectl apply -f <path-to-your-yaml>`.

Topics

- Client Credentials Grant
- Authorization Code Grant

Client Credentials Grant Type

This grant type allows an application to get an access token for resources about the client itself, rather than a user.

Dynamic Client Registration (via `ClientRegistration` custom resource):

```
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: ClientRegistration
metadata:
  name: <your client name>
spec:
  authorizationGrantTypes:
    - client_credentials
  # ...
```

Ensure that you are able to retrieve a token through your setup

1. Apply your ClientRegistration

   ```
   kubectl apply -f <path-to-the-clientregistration-yaml>
   ```

2. Verify your `ClientRegistration` was created

   ```
   kubectl get clientregistrations
   ```

   –> you should see a `ClientRegistration` with the name you provided

3. Verify your Secret was created

   ```
   kubectl get secrets
   ```

–> you should see a Secret with that same name you provided for the `ClientRegistration`

4. Get the client secret and decode it

   ```
   kubectl get secret <your-client-registration-name> -o jsonpath="{.data.client-secret}" | base64 -d
   ```

5. Get the client id (or get it from your configuration)

   ```
   kubectl get secret <your-client-registration-name> -o jsonpath="{.data.client-id}" | base64 -d
   ```

6. Request token

   ```
   curl -X POST <AUTH-DOMAIN>/oauth2/token?grant_type=client_credentials -v -u "YOUR_CLIENT_ID:DECODED_CLIENT_SECRET"
   ```

Authorization Code Grant Type

This grant type allows clients to exchange this code for access tokens.

Dynamic Client Registration (via `ClientRegistration` custom resource):

```
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: ClientRegistration
metadata:
  name: <your client name>
spec:
  authorizationGrantTypes:
    - authorization_code
  scopes:
     - openid
  # ...
```

Ensure that you are able to retrieve a token through your setup

Ensure there is an Identity Provider configured

1. Get your authserver's label name

   ```
   kubectl get authserver sso4k8s -o jsonpath="{.metadata.labels.name}"
   ```

2. Apply this sample ClientRegistration (read more about ClientRegistrations

   The following is an example ClientRegistration that will work in this setup. The required scopes are `openid, email, profile, roles`. The redirect URI here has been set to match that of `oauth2-proxy`.

   ```
   apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
   kind: ClientRegistration
   metadata:
    name: oauth2-proxy-client
    namespace: <your-namespace>
   spec:
    authServerSelector:
    matchLabels:
   ```

```
   name: <your-authserver-label-name>
 authorizationGrantTypes:
   - client_credentials
   - authorization_code
 requireUserConsent: false
 redirectURIs:
   - http://127.0.0.1:4180/oauth2/callback
 scopes:
   - name: openid
   - name: email
   - name: profile
   - name: roles
```

```
kubectl apply -f <path-to-the-clientregistration-yaml>
```

3. Verify your `ClientRegistration` was created

   ```
   kubectl get clientregistrations
   ```

   –> you should see a `ClientRegistration` with the name you provided

4. Verify your Secret was created

   ```
   kubectl get secrets
   ```

   –> you should see a Secret with that same name you provided for the `ClientRegistration`

5. Get the client secret and decode it

   ```
   CLIENT_SECRET=$(kubectl get secret <your-client-registration-name> -o jsonpath="{.data.client-secret}" | base64 -d)
   ```

6. Get the client id (or get it from your configuration)

   ```
   CLIENT_ID=$(kubectl get secret <your-client-registration-name> -o jsonpath="{.data.client-id}" | base64 -d)
   ```

7. Get the issuer uri

   ```
   ISSUER_URI=$(kubectl get secret <your-client-registration-name> -o jsonpath="{.data.issuer-uri}" | base64 -d)
   ```

8. Use the oauth2-proxy to spin up a quick trial run of the configured Authserver and run it with docker.

   ```
   docker run -p 4180:4180 --name oauth2-proxy bitnami/oauth2-proxy:latest \
   --oidc-issuer-url "$ISSUER_URI" \
   --client-id "$CLIENT_ID" \
   --insecure-oidc-skip-issuer-verification true \
   --client-secret "$CLIENT_SECRET" \
   --cookie-secret "0000000000000000" \
   --http-address "http://:4180" \
   --provider oidc \
   --scope "openid email profile roles" \
   --email-domain='*' \
   ```

```
--insecure-oidc-allow-unverified-email true \
--upstream "static://202" \
--oidc-groups-claim "roles" \
--oidc-email-claim "sub" \
--redirect-url "http://127.0.0.1:4180/oauth2/callback"
```

Note: Ensure that your issuer url does not resolve to `127.0.0.1`

9. Check your browser at `127.0.0.1:4180` to see if your configuration allows you to sign in.

   You should see a message that says “Authenticated”.

Securing your first Workload

This tutorial will walk you through the steps to add an authentication mechanism to a sample Spring Boot application using AppSSO service, running on Tanzu Application Platform (TAP).

Prerequisites

Before starting the tutorial, please ensure that the following items are addressed:

- **RECOMMENDED** Familiarity with Workloads and AppSSO
- Tanzu CLI `v0.11.6` or above is available locally.
- Tanzu Application Platform (TAP) `v1.2.0` or above is available and fully reconciled in your cluster.
  - Please ensure that you are using one of the following TAP Profiles
    - `run` (deploy-only) deploy the existing application from existing image and existing GitOps manifest.
    - `iterate` **Recommended** (build,deploy) build application from scratch and deploy from generated GitOps manifest.
    - `full` (build,deploy) ^^.
- AppSSO package `v1.0.0` or above is available and reconciled successfully on your cluster.
- AppSSO has at least one identity provider configured.
- Access to AppSSO Starter Java accelerator used in this tutorial.

Getting started

> Skip to step-by-step instructions if you are already familiar with the accelerator used in this tutorial.

Understanding the sample application

In this tutorial, you will be working with a sample Servlet-based Spring Boot application that uses Spring Security OAuth2 Client library .

You can find the source code for the application here. To follow along, be sure to Git clone the repository onto your local environment.

The application, once launched, has two pages:

- a **publicly-accessible home page (`/home`)**, available to everyone.
- a **user home page (`/authenticated/home`)**, for signed-in users only.

The security configuration for the above is located at `com.vmware.tanzu.apps.sso.sampleworkload.config.WebSecurityConfig`.

For more in-depth details about how apps are configured with Spring Security OAuth2 Client library, be sure to check out the official Spring Boot and OAuth2 tutorial.

By default, there is no application properties file in our sample application and this is by design: even the simplest application can be deployed with AppSSO, you can even go to start.spring.io and download a Spring Boot app with Spring Security OAuth2 Client library, and you are good to go! There is yet another reason for the absence of any properties files: a demonstration of Spring Cloud Bindings in action, which removes the need for any OAuth related properties. Spring Cloud Bindings will be introduced later in this tutorial.

The sample application's `ClientRegistration`

A critical piece of integration with AppSSO is to create a `ClientRegistration` custom resource definition. A `ClientRegistration` is a way for AppSSO to learn about the sample application. In the sample application, you can find the definition file named `client.yaml`, at the root of the source directory.

The `ClientRegistration` resource definition contains a few critical pieces in its specification:

- `authorizationGrantTypes` is set to a list of one: `authorization_code`. Authorization Code grant type is required for OpenID Connect authentication which we will be using in this tutorial.
- `redirectURIs` is set to a list of two URIs: a remote URI and a local URI (i.e. `127.0.0.1`). The remote URI will be the full URL to which AppSSO will redirect the user back upon successful authentication. The local URI is only meant for debugging purposes and can be ignored unless desired. The suffix of both URIs is important for Spring Security - it adheres to the default redirect URI template .
- `scopes` is set to a list of one scope, the `openid` scope. The `openid` scope is required by OpenID Connect specification in order to issue identity tokens which designate a user as 'signed in'.

For more details about `ClientRegistration` custom resource, see ClientRegistration CRD.

The `client.yaml` file is using ytt templating conventions. If you have the Tanzu Cluster Essentials installed, you should already have `ytt` available on your command line. Later in the tutorial, we will generate a final output `ClientRegistration` declaration that will look similar to the below:

```
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: ClientRegistration
metadata:
  name: appsso-starter-java
  namespace: workloads
```

```
spec:
  authServerSelector:
    matchLabels:
    # ask your Service Operator for labels to target an `AuthServer`
  clientAuthenticationMethod: basic
  authorizationGrantTypes:
    - authorization_code
  redirectURIs:
    - http://<app-url>/login/oauth2/code/<claim-name>
    - http://127.0.0.1:8080/login/oauth2/code/appsso-starter-java
  scopes:
    - name: openid
```

Understanding `Workload`s

To deploy the sample application onto a TAP cluster, we must first craft it as a `Workload` resource (a Cartographer CRD). A `Workload` resource can be thought of as a manifest for a process you want to execute on the cluster, and in this context, the type of workload is `web` - a web application. TAP clusters provide the capability to apply `Workload` resources out of the box within the proper profiles, as described in the prerequisites section.

To deploy a workload, it is best to work in a separate workload-specific namespace. Once created, there are required TAP configurations that need to be applied before a `Workload` in a specific namespace can be deployed properly.

Deploying the sample application as a Workload

To tie it all together and deploy the sample application, the following are the steps involved.

Create workload namespace

Create a workload namespace called `workloads`:

```
kubectl create namespace workloads
```

Apply required TAP workload configurations

Within the `workloads` namespace, apply TAP required developer namespaces as described.

Follow along with TAP developer namespace setup example in the Appendix.

Apply the `ClientRegistration`

Apply the `client.yaml` definition file (described above)

⚠ Make sure to set `auth_server_name` field to the name of the AuthServer custom resource.

```
ytt \
  --file client.yaml \
  --data-value namespace=workloads \
  --data-value workload_name=appsso-starter-java \
  --data-value domain=127.0.0.1.nip.io \
  --data-value auth_server_name="" \
  --data-value claim_name=appsso-starter-java | \
   kubectl apply -f-
```

A bit more detail on the above YTT data values:

- **namespace** - the namespace in which the workload will run.
- **workload_name** - the distinct name of the instance of the accelerator being deployed.
- **domain** - the domain name under which the workload will be deployed. The workload instance will use a subdomain to distinguish itself from other workloads. If working locally, `127.0.0.1.nip.io` is the easiest approach to get a working DNS route on a local cluster.
- **auth_server_name** - the name of the AuthServer resource that you have installed and want to use with your Workload.
- **claim_name** - the service resource claim name being assigned for this workload, this is the binding between the workload and AppSSO. You may choose any reasonably descriptive name for this, it will be used in the next step.

This command has generated a `ClientRegistration` definition and applied it to the cluster. To check the status of the client registration, run:

```
kubectl get clientregistration appsso-starter-java --namespace workloads
```

You should see the `ClientRegistration` entry listed.

Create a ClientRegistration service resource claim for the workload

Using Tanzu Services plugin CLI, create a service resource claim for the workload:

⚠ Name of the claim must be the same as the value of `claim_name` from previous step.

⚠ Resource name must be the same name as the workload name.

```
tanzu services claims create appsso-starter-java \
    --namespace workloads \
    --resource-namespace workloads \
    --resource-name appsso-starter-java \
    --resource-kind ClientRegistration \
    --resource-api-version "sso.apps.tanzu.vmware.com/v1alpha1"
```

Once applied, you may check the status of the claim like so:

```
tanzu services claim list --namespace workloads
```

You should see `appsso-starter-java` claim with `Ready` status as `True`.

For more information about service claims, check out the Services Toolkit docs here .

Deploy the workload

The Tanzu CLI command to create a workload for the sample application should look like the following:

⚠ You must have access to `gitops-appsso-starter-java.git` repository in Pivotal org. If you do not have access, create your own empty repository with a single commit on it, then point `gitops_repository` parameter to it. You must use `ssh` protocol if you are creating a private repository.

```
# When using 'iterate' or 'full' TAP profile(s) - build from source and deploy from ge
nerated GitOps manifest
tanzu apps workload create appsso-starter-java \
    --namespace workloads \
    --type web \
    --label app.kubernetes.io/part-of=appsso-starter-java \
    --service-ref "appsso-starter-java=services.apps.tanzu.vmware.com/v1alpha1:Resourc
eClaim:appsso-starter-java" \
    --git-repo ssh://git@github.com/sample-accelerators/appsso-starter-java.git \
    --git-branch main \
    --param gitops_repository=ssh://git@github.com/pivotal/gitops-appsso-starter-java.
git \
    --live-update \
    --yes
```

OR when using 'run' TAP profile - deploy workload via existing GitOps manifest

```
tanzu apps workload create appsso-starter-java \
    --namespace workloads \
    --type web \
    --label app.kubernetes.io/part-of=appsso-starter-java \
    --service-ref "appsso-starter-java=services.apps.tanzu.vmware.com/v1alpha1:Resourc
eClaim:appsso-starter-java" \
    --param gitops_repository=ssh://git@github.com/pivotal/gitops-appsso-starter-java.
git \
    --live-update \
    --yes
```

The above command creates a web `Workload` named 'appsso-starter-java' in the `workloads` namespace. The sample applications' source code repository is defined in the `git-repo` and `git-branch` parameters. Workloads are usually built from scratch and later deployed – the mechanism that allows a built artifact to be deployed is managed via GitOps approach, and so we specify a GitOps specific repository for the workload specified with `--param gitops_repository` parameter. From the previous step, we specify a `ClientRegistration` service resource claim via the `service-ref` parameter. In doing so, we enable the `Workload`'s Pods to have the necessary AppSSO-generated credentials available as a Service Binding. Learn more about how this works here.

It takes some minutes for the workload to become available as a URL.

To query the latest status of the Workload, run:

```
tanzu apps workload get appsso-starter-java --namespace workloads
```

⚠ You may see the status of the workload at first:

message: waiting to read value [.status.latestImage] from resource [image.kpack.io/appsso-starter-java] in namespace [workloads]

reason: MissingValueAtPath

status: Unknown

This is NOT an error, this is normal operation of a pending workload. Watch the status for changes.

Follow the `Workload` logs:

```
tanzu apps workload tail appsso-starter-java --namespace workloads
```

Once the status of the workload reaches the `Ready` state, you may navigate to the URL provided, which should look similar to:

```
http://appsso-starter-java.workloads.127.0.0.1.nip.io
```

Navigate to the URL in your favorite browser, and observe a large login button tailored for logging with AppSSO.

Once you have explored the accelerator and its operation, head on to the next section for uninstall instructions.

Cleaning up

You may delete the running accelerator by running the following:

Delete the sample application workload

```
tanzu apps workload delete appsso-starter-java --namespace workloads
```

Delete the service resource claim for the ClientRegistration

```
tanzu services claim delete appsso-starter-java --namespace workloads
```

Disconnect the accelerator from AppSSO

```
kubectl delete clientregistration appsso-starter-java --namespace workloads
```

Custom Resource Definitions

- AuthServer
- ClientRegistration

ClientRegistration

ClientRegistration is the request for client credentials for an AuthServer.

It implements the Service Bindings' ProvisionedService. The credentials are returned as a Service Bindings Secret.

A ClientRegistration needs to uniquely identify an AuthServer via spec.authServerSelector. If it matches none, too many or a disallowed AuthServer it won't get credentials. The other fields are for the configuration of the client on the AuthServer.

Spec

```
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: ClientRegistration
metadata:
  name: ""
  namespace: ""
spec:
  authServerSelector: # required
    matchLabels: { }
  redirectURIs: # required
    - ""
  scopes: # optional
    - name: ""
      description: ""
  authorizationGrantTypes: # optional
    - client_credentials
    - authorization_code
    - refresh_token
  clientAuthenticationMethod: basic # or "post", optional
  requireUserConsent: false # optional
status:
  authServerRef:
    apiVersion: ""
    issuerURI: ""
    kind: ""
    name: ""
    namespace: ""
  binding:
    name: ""
  clientID: ""
  clientSecretHelp: ""
  conditions:
    - lastTransitionTime: ""
      message: ""
      reason: ""
      status: "True" # or "False"
      type: ""
  observedGeneration: 0
```

Alternatively, you can interactively discover the spec with:

```
kubectl explain clientregistrations.sso.apps.tanzu.vmware.com
```

Status & conditions

The `.status` subresource helps you to learn about your client credentials, the matched `AuthServer` and to troubleshoot issues.

`.status.authServerRef` identifies the successfully matched `AuthServer` and its issuer URI.

`.status.binding.name` is the name of the Service Bindings `Secret` which contains the client credentials.

`.status.conditions` documents each step in the reconciliation:

- `Valid`: Is the spec valid?
- `AuthServerResolved`: Has the targeted `AuthServer` been resolved?
- `ClientSecretResolved`: Has the client secret been resolved?
- `ServiceBindingSecretApplied`: Has the Service Bindings Secret with the client credentials been applied?
- `AuthServerConfigured`: Has the resolved `AuthServer` been configured with the client?
- `Ready`: whether all the previous conditions are “True”

The super condition `Ready` denotes a fully successful reconciliation of a given `ClientRegistration`.

If everything goes well you will see something like this:

```
status:
  authServerRef:
    apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
    issuerURI: http://authserver-sample.default
    kind: AuthServer
    name: authserver-sample
    namespace: default
  binding:
    name: clientregistration-sample
  clientID: default_clientregistration-sample
  clientSecretHelp: 'Find your clientSecret: ''kubectl get secret clientregistration-s
ample --namespace default'''
  conditions:
    - lastTransitionTime: "2022-05-13T07:56:41Z"
      message: ""
      reason: Updated
      status: "True"
      type: AuthServerConfigured
    - lastTransitionTime: "2022-05-13T07:56:40Z"
      message: ""
      reason: Resolved
      status: "True"
      type: AuthServerResolved
    - lastTransitionTime: "2022-05-13T07:56:40Z"
      message: ""
      reason: ResolvedFromBindingSecret
      status: "True"
      type: ClientSecretResolved
    - lastTransitionTime: "2022-05-13T07:56:41Z"
      message: ""
      reason: Ready
      status: "True"
      type: Ready
    - lastTransitionTime: "2022-05-13T07:56:40Z"
      message: ""
      reason: Applied
      status: "True"
      type: ServiceBindingSecretApplied
    - lastTransitionTime: "2022-05-13T07:56:40Z"
      message: ""
      reason: Valid
      status: "True"
      type: Valid
  observedGeneration: 1
```

Example

```
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: ClientRegistration
metadata:
  name: my-client-registration
  namespace: app-team
spec:
  authServerSelector:
    matchLabels:
      for: app-team
      ldap: "true"
  redirectURIs:
    - "https://127.0.0.1:8080/authorized"
    - "https://my-application.com/authorized"
  requireUserConsent: false
  clientAuthenticationMethod: basic
  authorizationGrantTypes:
    - "client_credentials"
    - "refresh_token"
  scopes:
    - name: "openid"
      description: "To indicate that the application intends to use OIDC to verify the
user's identity"
    - name: "email"
      description: "The user's email"
    - name: "profile"
      description: "The user's profile information"
```

The client is being registered with the authorization server with the given specs. The resulting client credentials are available in a Secret that's owned by the ClientRegistration.

```
apiVersion: v1
kind: Secret
type: servicebinding.io/oauth2
data: # fields below are base64-decoded for display purposes only
  type: oauth2
  provider: appsso
  client-id: default_my_client_registration
  client-secret: c2VjcmV0 # auto-generated
  issuer-uri: https://appsso.example.com
  client-authentication-method: basic
  scope: openid,email,profile
  authorization-grant-types: client_credentials,refresh_token
```

AuthServer

`AuthServer` represents the request for an OIDC authorization server. It results in the deployment of an authorization server backed by Redis over mTLS.

An `AuthServer` should have labels which allow to uniquely match it amongst others. `ClientRegistration` selects an `AuthServer` by label selector and needs a unique match to be successful.

To allow `ClientRegistrations` from all or a restricted set of Namespaces, the annotation `sso.apps.tanzu.vmware.com/allow-client-namespaces` must be set. Its value is a comma-separated list of allowed Namespaces, e.g. `"app-team-red,app-team-green"`, or `"*"` if it should allow clients from all namespaces. If the annotation is missing, no clients are allowed.

An `AuthServer` has a `spec.issuerURI` which is the entry point for clients and end-users. A form of Ingress needs to be configured for this issuer URI.

Token signature keys are configured through `spec.tokenSignature`. If no keys are configured, no tokens can be minted.

Identity providers are configured under `spec.identityProviders`. If there are none, end-users won't be able to log in.

The deployment can be further customized by configuring replicas, resources, http server and logging properties.

An `AuthServer` reconciles into the following resources in its namespace:

```
AuthServer/my-authserver
├─Certificate/my-authserver-redis-client
├─Certificate/my-authserver-redis-server
├─Certificate/my-authserver-root
├─ConfigMap/my-authserver-ca-cert
├─Deployment/my-authserver-auth-server
├─Deployment/my-authserver-redis
├─Issuer/my-authserver-bootstrap
├─Issuer/my-authserver-root
├─Role/my-authserver-auth-server
├─RoleBinding/my-authserver-auth-server
├─Secret/my-authserver-auth-server-clients
```

```
├─Secret/my-authserver-auth-server-keys
├─Secret/my-authserver-auth-server-properties
├─Secret/my-authserver-redis-client-cert-keystore-password
├─Secret/my-authserver-registry-credentials
├─Service/my-authserver-redis
└─ServiceAccount/my-authserver-auth-server
```

Spec

```
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
metadata:
  name: ""
  namespace: ""
  labels: { } # required, must uniquely identify this AuthServer
  annotations:
    sso.apps.tanzu.vmware.com/allow-client-namespaces: "" # required, must be "*" or a
comma-separated list of allowed client namespaces
    sso.apps.tanzu.vmware.com/allow-unsafe-issuer-uri: "" # optional
    sso.apps.tanzu.vmware.com/allow-unsafe-identity-provider: "" # optional
spec:
  issuerURI: "" # required
  tokenSignature: # optional
    signAndVerifyKeyRef:
      name: ""
    extraVerifyKeyRefs:
      - name: ""
  identityProviders: # optional
    # each must be one and only one of internalUnsafe, ldap, openID or saml
    - name: "" # must be unique
      internalUnsafe: # requires annotation `sso.apps.tanzu.vmware.com/allow-unsafe-id
entity-provider: ""`
        users:
          - username: ""
            password: ""
            givenName: ""
            familyName: ""
            email: ""
            emailVerified: false
            roles:
              - ""
    - name: "" # must be unique
      ldap:
        server:
          scheme: ""
          host: ""
          port: 0
          base: ""
        bind:
          dn: ""
          passwordRef:
            name: ldap-password
        user:
          searchFilter: ""
          searchBase: ""
        group:
          searchFilter: ""
          searchBase: ""
          searchSubTree: false
          searchDepth: 0
          roleAttribute: ""
    - name: "" # must be unique
      openID:
        issuerURI: ""
        clientID: ""
        clientSecretRef:
          name: ""
        scopes:
          - ""
    - name: "" # must be unique
      saml:
        metadataURI: ""
        claimMappings: { }
  replicas: 1 # optional, default 2
  logging: "" # optional, must be valid YAML
  server: "" # optional, must be valid YAML
  resources: # optional, default {requests: {cpu: "256m", memory: "300Mi"}, limits: {c
pu: "2", memory: "768Mi"}}
    requests:
      cpu: ""
      mem: ""
    limits:
      cpu: ""
      mem: ""
  redisResources: # optional, default {requests: {cpu: "50m", memory: "100Mi"}, limit
s: {cpu: "100m", memory: "256Mi"}}
    requests:
      cpu: ""
      mem: ""
    limits:
      cpu: ""
      mem: ""
status:
  observedGeneration: 0
  clientRegistrationCount: 1
  tokenSignatureKeyCount: 0
  deployments:
    authServer:
      LastParentGenerationWithRestart: 0
      configHash: ""
      image: ""
      replicas: 0
    redis:
      image: ""
  conditions:
    - lastTransitionTime:
      message: ""
      reason: ""
      status: "True" # or "False"
      type: ""
```

Alternatively, you can interactively discover the spec with:

```
kubectl explain authservers.sso.apps.tanzu.vmware.com
```

Status & conditions

The `.status` subresource helps you to learn the `AuthServer`'s readiness, resulting deployments, attached clients and to troubleshoot issues.

`.status.tokenSignatureKeyCount` is the number of currently configured token signature keys.

`.status.clientRegistrationCount` is the number of currently registered clients.

`.status.deployments.authServer` describes the current authorization server deployment by listing the running image, its replicas, the hash of the current configuration and the generation which has last resulted in a restart.

`.status.deployments.redis` describes the current Redis deployment by listing its running image.

`.status.conditions` documents each step in the reconciliation:

- `Valid`: Is the spec valid?
- `ImagePullSecretApplied`: Has the image pull secret been applied?
- `SignAndVerifyKeyResolved`: Has the single sign-and-verify key been resolved?
- `ExtraVerifyKeysResolved`: Have the single extra verify keys been resolved?
- `IdentityProvidersResolved`: Has all identity provider configuration been resolved?
- `ConfigResolved`: Has the complete configuration for the authorization server been resolved?
- `AuthServerConfigured`: Has the complete configuration for the authorization server been applied?
- `IssuerURIReady`: Is the authorization server yet responding to `{spec.issuerURI}/.well-known/openid-configuration`?
- `Ready`: whether all the previous conditions are "True"

The super condition `Ready` denotes a fully successful reconciliation of a given `ClientRegistration`.

If everything goes well you will see something like this:

```
status:
  observedGeneration: 1
  tokenSignatureKeyCount: 3
  clientRegistrationCount: 1
  deployments:
    authServer:
      LastParentGenerationWithRestart: 1
      configHash: "13146309071473757471"
      image: dev.registry.tanzu.vmware.com/sso-for-kubernetes/authserver@sha256:9c761d
d21bdd54cf8bf0de3cb23e04d75dcdedbbeee82bb78f6d3419c1c748ea
      replicas: 1
    redis:
      image: dev.registry.tanzu.vmware.com/sso-for-kubernetes/redis@sha256:3906dfa3d49
b340ffc85c05890ddca7e5a9c775344c9b9d3bacda9bb6efac191
  conditions:
    - lastTransitionTime: "2022-05-13T08:29:55Z"
      message: ""
      reason: KeysConfigSecretUpdated
      status: "True"
      type: AuthServerConfigured
    - lastTransitionTime: "2022-05-13T08:29:54Z"
      message: ""
      reason: Resolved
      status: "True"
      type: ConfigResolved
    - lastTransitionTime: "2022-05-13T08:29:54Z"
      message: ""
      reason: ExtraVerifyKeysResolved
      status: "True"
      type: ExtraVerifyKeysResolved
    - lastTransitionTime: "2022-05-13T08:29:54Z"
      message: ""
      reason: Resolved
      status: "True"
      type: IdentityProvidersResolved
    - lastTransitionTime: "2022-05-13T08:29:54Z"
      message: ""
      reason: ImagePullSecretApplied
      status: "True"
      type: ImagePullSecretApplied
    - lastTransitionTime: "2022-05-13T09:04:22Z"
      message: ""
      reason: Ready
      status: "True"
      type: IssuerURIReady
    - lastTransitionTime: "2022-05-13T09:04:22Z"
      message: ""
      reason: Ready
      status: "True"
      type: Ready
    - lastTransitionTime: "2022-05-13T08:29:54Z"
      message: ""
      reason: SignAndVerifyKeyResolved
      status: "True"
      type: SignAndVerifyKeyResolved
    - lastTransitionTime: "2022-05-13T08:29:54Z"
      message: ""
      reason: Valid
      status: "True"
      type: Valid
```

RBAC

The `ServiceAccount` of the authorization server has a `Role` with the following permissions:

```
- apiGroups:
    - ""
  resources:
    - secrets
  verbs:
    - get
    - list
    - watch
  resourceNames:
    - { name }-auth-server-keys
    - { name }-auth-server-clients
```

Example

This example requests an authorization server with the issuer URI `http://authserver-sample.default`, two token signature keys and two identity providers. It also configures ingress as you would on a local Kind cluster.

```
---
apiVersion: sso.apps.tanzu.vmware.com/v1alpha1
kind: AuthServer
metadata:
  name: authserver-sample
  namespace: default
  labels:
    name: authserver-sample
    sample: "true"
  annotations:
    sso.apps.tanzu.vmware.com/allow-client-namespaces: "*"
    sso.apps.tanzu.vmware.com/allow-unsafe-identity-provider: ""
    sso.apps.tanzu.vmware.com/allow-unsafe-issuer-uri: ""
spec:
  replicas: 1
  issuerURI: http://authserver-sample.default
  tokenSignature:
    signAndVerifyKeyRef:
      name: sample-token-signing-key
    extraVerifyKeyRefs:
      - name: sample-token-verification-key
  identityProviders:
    - name: internal
      internalUnsafe:
        users:
          - username: test-user-1
            password: $2a$10$201z9o/tHlocFsHFTo0plukh03ApBYe4dRiXcqeyRQH6CNNtS8jWK #!
bcrypt-encoded "password"
            roles:
              - message.write
          - username: test-user-2
            password: $2a$10$201z9o/tHlocFsHFTo0plukh03ApBYe4dRiXcqeyRQH6CNNtS8jWK #!
bcrypt-encoded "password"
            roles:
              - message.read
    - name: okta
      openID:
        issuerURI: https://dev-xxxxxx.okta.com
        clientID: xxxxxxxxxxxxx
        clientSecretRef:
          name: okta-client-secret
        authorizationUri: https://dev-xxxxxx.okta.com/oauth2/v1/authorize
        tokenUri: https://dev-xxxxxx.okta.com/oauth2/v1/token
        jwksUri: https://dev-xxxxxx.okta.com/oauth2/v1/keys
        scopes:
          - openid
        claimMappings:
          roles: my_custom_okta_roles_claim

---
apiVersion: secretgen.k14s.io/v1alpha1
kind: RSAKey
```

```
metadata:
  name: sample-token-signing-key
  namespace: default
spec:
  secretTemplate:
    type: Opaque
    stringData:
      key.pem: $(privateKey)
      pub.pem: $(publicKey)

---
apiVersion: secretgen.k14s.io/v1alpha1
kind: RSAKey
metadata:
  name: sample-token-verification-key
  namespace: default
spec:
  secretTemplate:
    type: Opaque
    stringData:
      key.pem: $(privateKey)
      pub.pem: $(publicKey)

---
apiVersion: v1
kind: Secret
metadata:
  name: okta-client-secret
  namespace: default
stringData:
  clientSecret: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

---
apiVersion: v1
kind: Service
metadata:
  name: authserver-sample
  namespace: default
spec:
  selector:
    app.kubernetes.io/part-of: authserver-sample
    app.kubernetes.io/component: authorization-server
  ports:
    - port: 80
      targetPort: 8080

---
apiVersion: projectcontour.io/v1
kind: HTTPProxy
metadata:
  name: authserver-sample
  namespace: default
spec:
  virtualhost:
    fqdn: authserver-sample.default
  routes:
    - conditions:
        - prefix: /
      services:
        - name: authserver-sample
          port: 80
```

Known Issues

- AppSSO makes requests to external identity providers with `http` rather than `https`.

Appendix

- TAP developer namespace setup example

TAP developer namespace setup example

Applies to TAP v1.2.0

The following is an example setup of a TAP developer namespace (for use with Workloads) as per **TAP v1.2.0** guidelines. You may use this document as a reference guide to expedite the process of creating the necessary space to deploy your workloads.

Installing the namespace configuration

Add container image registry credentials

```
# The namespace in which your workloads will be applied to
export WORKLOADS_NAMESPACE="workloads"

# The container image registry to which your workloads will be published to. This example uses Google Container Registry.
export CONTAINER_IMAGE_REGISTRY="https://gcr.io"
export CONTAINER_IMAGE_REGISTRY_USERNAME="<username>"
export CONTAINER_IMAGE_REGISTRY_PASSWORD="<password>"

# Apply registry credentials for access to container image registry
tanzu secret registry add registry-credentials \
   --server "${CONTAINER_IMAGE_REGISTRY}" \
   --username "${CONTAINER_IMAGE_REGISTRY_USERNAME}" \
   --password "${CONTAINER_IMAGE_REGISTRY_PASSWORD}" \
   --namespace "${WORKLOADS_NAMESPACE}"
```

To verify secret creation, run:

```
tanzu secret registry list -n ${WORKLOADS_NAMESPACE}
```

Output will be similar to:

```
NAME                  REGISTRY        EXPORTED      AGE
registry-credentials  https://gcr.io  not exported  20s
```

Apply namespace configurations

Save ytt-templated yaml configuration for workloads namespace here (found on this page below) – select all and save to local filesystem. In the example below, the file is saved to the default

MacOS `Downloads` folder.

Apply the namespace configurations using `kapp`:

```
# The namespace in which your workloads will be applied to
export WORKLOADS_NAMESPACE="workloads"

# The Git repository hostname (without https prefix) where your workload source code lives. This example uses GitHub.
export GIT_REPOSITORY_HOSTNAME="github.com"

# Private/public key pair that allows read/write access to GIT_REPOSITORY_HOSTNAME
export GIT_REPOSITORY_SSH_PRIVATE_KEY="<private-key-string>"
export GIT_REPOSITORY_SSH_PUBLIC_KEY="<public-key-string>"

# Set known hosts string
export GIT_REPOSITORY_KNOWN_HOSTS="$(ssh-keyscan github.com)"

# Deploy kapp as per namespace spec yaml
ytt \
   --data-value-file=git_repository_hostname=<(echo "${GIT_REPOSITORY_HOSTNAME}") \
   --data-value-file=ssh_private_key=<(echo "${GIT_REPOSITORY_SSH_PRIVATE_KEY}") \
   --data-value-file=ssh_public_key=<(echo "${GIT_REPOSITORY_SSH_PUBLIC_KEY}") \
   --data-value-file=known_hosts=<(echo "${GIT_REPOSITORY_KNOWN_HOSTS}") \
   --data-value=namespace="${WORKLOADS_NAMESPACE}" \
   --file ~/Downloads/tap-dev-ns-setup.yaml |
   kapp deploy \
     --namespace ${WORKLOADS_NAMESPACE} \
     --app workload-prerequisites \
     --wait \
     --wait-timeout=120s \
     --diff-changes \
     --yes \
     --file -
```

By deploying with `kapp`, you have the power to *cleanly* uninstall the configuration once you are done with the demonstration or if there is an issue with the configuration.

Your cluster is now ready to host Workloads.

Uninstalling namespace configurations

To uninstall the above configurations, run:

```
# The namespace in which your workloads will be applied to
export WORKLOADS_NAMESPACE="workloads"

tanzu secret registry delete registry-credentials \
    --namespace "${WORKLOADS_NAMESPACE}" \
    --yes
kapp delete \
    --namespace ${WORKLOADS_NAMESPACE} \
    --app "workload-prerequisites" \
    --yes \
    --diff-changes
```

Developer namespace configuration ytt template

```
#@ load("@ytt:data", "data")
---
apiVersion: v1
kind: Namespace
metadata:
  name: #@ data.values.namespace
---
#! see: https://fluxcd.io/docs/components/source/gitrepositories/#ssh-authentication
apiVersion: v1
kind: Secret
metadata:
  name: git-ssh
  namespace: #@ data.values.namespace
  annotations:
    tekton.dev/git-0: #@ data.values.git_repository_hostname
type: kubernetes.io/ssh-auth
stringData:
  ssh-privatekey: #@ data.values.ssh_private_key
  identity: #@ data.values.ssh_private_key
  identity.pub: #@ data.values.ssh_public_key
  known_hosts: #@ data.values.known_hosts
---
apiVersion: v1
kind: Secret
metadata:
  name: tap-registry
  namespace: #@ data.values.namespace
  annotations:
    secretgen.carvel.dev/image-pull-secret: ""
type: kubernetes.io/dockerconfigjson
data:
  .dockerconfigjson: e30K
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: default
  namespace: #@ data.values.namespace
  annotations:
    kapp.k14s.io/create-strategy: fallback-on-update
secrets:
  - name: git-ssh
  - name: registry-credentials
imagePullSecrets:
  - name: registry-credentials
  - name: tap-registry
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: default-permit-deliverable
  namespace: #@ data.values.namespace
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: deliverable
```

```
subjects:
  - kind: ServiceAccount
    name: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: default-permit-workload
  namespace: #@ data.values.namespace
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: workload
subjects:
  - kind: ServiceAccount
    name: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: dev-permit-app-editor
  namespace: #@ data.values.namespace
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: app-editor
subjects:
  - kind: Group
    name: "namespace-developers"
    apiGroup: rbac.authorization.k8s.io
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: namespace-dev-permit-app-editor
  namespace: #@ data.values.namespace
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: app-editor-cluster-access
subjects:
  - kind: Group
    name: "namespace-developers"
    apiGroup: rbac.authorization.k8s.io
```