App Builder

The *app-builder* tool assists us in developing and building a containerized App that can be installed using the App Center. In summary, the tool takes an *App definition file* as input and from this builds a package that the App Center can use to install the App. How to build the actual container images used by the App is not in the scope of this document and basic knowledge about how to work with container images is expected.

Prerequisites

To run the app-builder, we must have Docker installed. If running on Windows, we can use Docker Desktop and for Ubuntu or other Linux variants we can use the standard channels to install it. The actual app-builder tool is available as both a power-shell script (`app-builder.ps1`) for Windows and a bash script (`app-builder.sh`) for Linux. You can download it from here: https://doc.developer.milestonesys.com/appen/?path=App-Builder

A simple example

To introduce the app-builder, it is best to look at a simple example. Building an App always starts with creating an App definition file. It is a YAML file and below is shown an example of how a very simple App can be defined.

```yaml
name: "most-simple-app"
realm: "my-realm"
version: 1.0.0
description: "The most simple App you can make"

prerequisites:
  systemVersion: ">=0.3.9"

services:
- name: pub-my-realm-my-service
  routes:
  - name: my-route
    port: 80
    targetPort: 80
  containers:
  - name: my-container
    image: sandbox.io/hello-world/webserver:1.2.2
```

In summary, we define an App named `most-simple-app` which has one service called `pub-my-realm-my-service`. The service will run the container image `sandbox.io/hello-world/webserver:1.2.2` and expose port 80 through which other services in the system can access it using hostname `pub-my-realm-my-service`. It is worth mentioning here that the service names are governed by a strict naming convention which you can read more about in the section describing the app-definition file.

Given the above App definition file, we can now build the App using the app-builder tool.

```
./app-builder.sh -f most-simple-app.yaml
```

This will generate a file in the current directory named `most-simple-app-1.0.0.tgz`. To make the App show up in the App Center, this file must be published to one of the repositories that the App Center is configured to use. The public repositories should not be used until the App is tested and approved so we need some other repository where we can put it while we are developing and testing the App. This is what the *Sandbox* repository is for.

The Sandbox repository

The sandbox provides an internal repository deployed on the same system as the App Center itself. It is only available if enabled specifically and should not be so for any production environment. In this section we will look at how to push an App to the sandbox repository and thereby making it visible to the App Center.

Before being able to push an App to the sandbox repository, we must first login to the system. This is done by running the app-builder with the `login` command.

```
./app-builder.sh login
```

With this in place, we can now push the App we built earlier with the `push` command

```
./app-builder.sh -f most-simple-app.yaml push
```

This will only push the App itself (its definition) and not the containers it is referring to. In the example above, the App refers to one container image named `sandbox.io/hello-world/webserver:1.2.2`. To push this container to the sandbox, we must tag our locally built container image with `[system ip / hostname]:5000/sandbox.io/hello-world/webserver:1.2.2` and then run the `docker push` command. By default, Docker does not accept insecure registries, and we therefore have to add the following lines to the Docker Daemon configuration and restart the service.

```
{
  "insecure-registries": [
    "[system ip / hostname]:5000",
  ]
}
```

At this point, we should be able to go to the App Center, find the App and install it.

Detailed command line usage

Let us take a look at some of the other commands that the app-builder supports. Below is listed the complete usage of the app-builder tool. Here we have already seen the `build`, `login` and `push` commands in action.

```
App Builder - Building your own App for the App Center

Usage: app-builder [options] <command>

Options:
  -h               This message
  -f <file>        App YAML definition file to use; default is app-definition.yaml
  -c <directory>   Directory with custom definitions; default is the YAML definition
file without file extension
  -o <directory>   Directory to store packaged Helm Chart of App; default is current
directory
  -r <url>         Repository from which to fetch app-registation helm chart;
default is https://horizonsystem.azurewebsites.net/system
  -n               Non-interactive mode for build pipelines; default is false. When
true, no interactive prompts are shown

Commands:
  build                   Build App as Helm Chart (the default command)
  login                   Login to to system and remember credentials for later use
  logout                  Logout from system and forget stored credentials
  dashboard               Expose Kubernetes Dashboard through localhost
  dashboard-create-token  Create new token allowing you to login to the Kubernetes
Dashboard
  list                    List Apps currently in sandbox repository
  push                    Push App to sandbox repository
```

```
  remove                  Remove App from sandbox repository
  install-from-file       Install App from local file
  install-from-repo       Install App from sandbox repository
  uninstall               Uninstall App
  restart                 Restart all containers deployed by App and pull new images
before starting them again
  status                  Show status of all containers deployed by App
  events                  Show events related to deployment of App
  volumes                 Show volumes used by App
  bundle                  Bundle App including container images together in one
archive file
```

Commands for updating the App repository

Above, we saw how to use the `push` command to upload an App to the sandbox repository. If the App already exist in the sandbox repository, it will be removed first. Pushing an already existing App will thus produce output as shown below.

```
{"deleted":true}
{"saved":true}
```

To explicitly remove an App from the repository, we can use the `remove` command and we will see output like this.

```
{"deleted":true}
```

Commands for running Apps

Instead of using the App Center to install an App, we can also do it directly with the app-builder tool. There are two commands available for this. The first one is `install-from-file` which will install the App from the generated output file directly. Here is example output from running this command.

```
Installing /root/out/most-simple-app-1.0.0.tgz to system at 10.10.16.34
Release "most-simple-app" does not exist. Installing it now.
NAME: most-simple-app
LAST DEPLOYED: Mon Mar  3 12:04:40 2025
NAMESPACE: most-simple-app
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Here, the input file `most-simple-app-1.0.0.tgz` is the App that was built using the `build` command. By default it is saved to the current directory and is also read from here by default when using the `install-from-file` command. We can change the default directory by specifying the `-o` option.

The second way to install an App using the app-builder is to use the `install-from-repo` command. This one will install the App from the sandbox repository and thus requires it to be uploaded first with the `push` command. It will generate similar output as shown above.

To get a list of all Apps that has been pushed to the sandbox repository, we can use the `list` command. The output will be YAML formatted with each global key being the name of an App.

```
most-simple-app:
- apiVersion: v2
  appVersion: 1.0.0
  created: '2025-03-03T13:08:17.355513209Z'
  dependencies:
  - name: app-registration
    repository: https://horizonsystem.azurewebsites.net/system
    version: 1.7.0
  description: The most simple App you can make
  digest: a382ea13b546f859bace3ba74e4e63554cc6e6ff289cf2d181f7459de47ffac9
  name: most-simple-app
  type: application
  urls:
  - charts/most-simple-app-1.0.0.tgz
  version: 1.0.0
```

When an App is installed, it will instruct the system to pull the needed images from the container registry and run them. We can at any time restart all running containers using the `restart` command. For every service provided by the App, we will see an output line similar to the following

```
pod "my-service-857dc79c5b-csghm" deleted
```

Behind the scenes, the container (pod) running the service is deleted, but since the App definition declares that it must be running, it is automatically restarted by the container orchestration framework. One useful side-effect of running the `restart` command is that all container images that have changed will be re-pulled from the container registry.

Finally, uninstalling the App, is also possible with the `uninstall` command.

Commands for getting status about Apps

If things are not working as expected, then it can be useful to get more detailed information about the current status of an App. The `status` command will for each service of the App show if it is currently running, and if so, for how long and how many restarts have been made. The output could look like this.

```
NAME                          READY   STATUS    RESTARTS   AGE   IP          NODE
my-service-857dc79c5b-zbjcc   1/1     Running   0          12m   10.1.27.8   dkws-
vt02-05
```

If a service is not starting as expected, then the `events` command can be useful. It will show all events related to getting the App up and running. If e.g., a container image could not be pulled, then this will show up here. Below we see output from the `events` command notifying us that the image `sandbox.io/hello-world/webserver:1.2.3` could not be pulled.

```
LAST SEEN   TYPE      REASON    OBJECT                            MESSAGE
4m41s       Normal    BackOff   pod/my-service-55c6f6f459-sj5fs   Back-off pulling
```

```
image "sandbox.io/hello-world/webserver:1.2.3"
4m41s       Warning   Failed    pod/my-service-55c6f6f459-sj5fs   Error:
ImagePullBackOff
```

For services using volumes for persistent storage, the `volumes` command will list all active volumes and where they are physicially located on disk. The example output below shows that `my-volume` is bound to the directory `/pool/media/2` on server with hostname `dkws-vt02-06`.

```
{
  "volume": "my-volume-pvc",
  "hostname": "dkws-vt02-06",
  "path": "/pool/media/2",
  "class": "media",
  "phase": "Bound"
}
```

If more details are needed about the status of an App (like log files) or the system as a whole, we can use the `dashboard` command to run the Kubernetes Dashboard. When running this command, the dashboard will be made available through https://localhost:10443 until we press Ctrl+C to break the command. Note that this only works if the system is setup with the developer option enabled. If not, the Kubernetes Dashboard is not installed in the system.

Commands for bundling an App

Once an App is working and it is ready for going into a Milestone hosted repository, the `bundle` command can be used to create one file containing both the App definition and all referred container images. This bundle can then be sent to Milestone for verification and if approved it will be made available in the approriate Milestone hosted repository.

App definition file

In this section we will take a closer look at the App definition file. We already saw a simple example earlier that looked like this

```
name: "most-simple-app"
realm: "my-realm"
version: 1.0.0
description: "The most simple App you can make"

prerequisites:
  systemVersion: ">=0.3.9"

services:
- name: pub-my-realm-my-service
  routes:
  - name: my-route
    port: 80
    targetPort: 80
  containers:
```

```
  - name: my-container
    image: sandbox.io/hello-world/webserver:1.2.2
```

This is a very minimal App definition file which exposes a service named `pub-my-realm-my-service`. There is a strict naming convention for service names, which the app-builder enforces. The name can be divided into three parts following this pattern `[pub-]<realm>-<name>`. In the above example the parts would be `pub`, `my-realm` and `my-service`. The first part is optional and will if present always be `pub`. This signifies that the service is *public* and can be reached by any other service in the system. Had we changed the name in the example above to `my-realm-my-service`, then this service would only be accessible from other services within the same *realm* as the App itself. The realm of an App is specified using the top-level property named `realm` and this value must always match the realm part of the service name. The third part, which in the example is `my-service`, is a descriptive but unique name of the service within the given realm.

In the following sections, we will now look at how we can extend this simple example in different ways.

App metadata

We have already seen how we can specify a `name`, `realm`, `version` and `description` of an App. Each of these are mandatory fields and the app-builder will fail if they are not explicitly defined. Optional metadata fields are `license`, `readme` and `changelog` as illustrated below.

```
license: "MIT License"
readme: |
  ## Overview
  This is an app that shows the hello world message
  in a web package.
changelog: |
  ## [1.0.0] - 2025-02-25
  First version.
```

The `license` field should be set to the name of the license that applies to the App. We can use any text that identifies the license in use whether it is open source or commercial. The `readme` field is text which in details explain why the App is useful, what can be done with the App, and how it is used. This is in contrast to the `description` field that we saw earlier, which is expected to be a single line describing the App. The `changelog` field must be a curated, chronologically ordered list of notable changes for each version of the App. Both the `readme` and `changelog` fields can be specified using Markdown syntax.

Specifying prerequisites

In the simple example, we saw how to specify a prerequisite on the system version using the `systemVersion` field. When the App is installed, it will be checked that the running system fulfills the version requirement and fail if not. Thus, it might be necessary to upgrade our system before a given App can be installed.

While the `systemVersion` is a required field, there are also two optional prerequisites that we can provide. These are for system services and other Apps. Below is shown an example that illustrates both.

```
prerequisites:
  systemVersion: ">=0.3.1"
```

```
  systemServices: [ "kafka", "fuseki" ]
  apps:
  - name: "another-app"
    version: ">=1.6.2"
  - name: "yet-another-app"
    version: ">=1.2.0"
```

In this example, we use the `systemServices` field to specify that the App must have both Kafka and Fuseki installed to function. We don't provide a specific version for these since this is indirectly implied by the given `systemVersion`. Another possible prerequisite that the App can define is for other Apps to be already installed. In the example above we e.g., require `another-app` in at least version `1.6.2` to be already installed. When installing an App through the App Center, the required system services and App dependencies will be automatically installed.

Consuming and providing APIs from contracts

An API contract serves as a formal agreement outlining how different services should interact with each other. The agreement defines rules, specifications, and expectations for data exchange, functionality, and communication. When an App depends on the APIs of a contract, we say that it consumes the APIs of the contract. Likewise, when an App is implementing the APIs of a contract, we say that is provides the APIs of the contract. We express this relationship by adding an `apis` section to the app definition file. An example is shown below.

```
apis:
  consumes:
  - name: my-realm.other-api-contract
    version: ">=1.1.2"
  provides:
  - name: my-realm.my-api-contract
    version: "1.4.7"
```

Here the App consumes APIs from the contract named `other-api-contract` in version `=>1.1.2`. The example also specifies that the App provides the APIs of the contract named `my-api-contract` in version `1.4.7`. The version scheme used is Semantic Versioning 2.0.0. For comparing versions, the `semverCompare` function is used which you can read more about here.

Note, that before installing an App, it will be checked if the consumed APIs are made available by an already installed App. If not, the installation will fail.

Using the client credentials flow

Often, an App will implement a service which will need access to other services in the system without a user being directly involved. For authorizing this kind of service to service communication, the recommended approach is to use the *OAuth 2 Client Credentials Flow*. The first step in this flow is to get our App registered with the *XProtect Identity Provider*. This is done by adding the following top-level section to the App definition file

```
credentials:
  clientCredentialsFlow:
    clientName: "app-using-client-credentials-flow"
    clientScopes: [ "managementserver" ]
```

For the registration we must provide a descriptive `clientName` and a list of `clientScopes` identifying which scopes our App will need access to. When the App is installed, it will now automatically be registered as a client with the XProtect Identity Provider. Below we will refer to the XProtect Identity Provider as simply the IDP.

As the App gets registered with the IDP, a *client id* and *client secret* are generated. Using these as credentials, we can now ask the IDP for an *access token* with a given scope and using this token our App can now connect to other services and be properly authorized.

For all running containers, the URL of the IDP is available as an environment variable named `SYSTEM_IDENTITY_PROVIDER` and requesting an access token is as simple as sending a HTTP POST request corresponding to running the following `curl` command.

```
curl --request POST \
     --url ${SYSTEM_IDENTITY_PROVIDER}/connect/token \
     --header 'content-type: application/x-www-form-urlencoded' \
     --data grant_type=client_credentials \
     --data scope=managementserver \
     --data client_id=${CCF_CLIENT_ID} \
     --data client_secret=${CCF_CLIENT_SECRET}
```

The `client_id` and `client_secret` are provided inside each running container through two environment variables named respectively `CCF_CLIENT_ID` and `CCF_CLIENT_SECRET`.

The body of the response will look something like this

```
{
  "access_token": "eyJhbGc...",
  "expires_in": 3600,
  "token_type": "Bearer",
  "scope": "managementserver"
}
```

Registering a client for the authorization code flow

The *OAuth 2 Authorization Code Flow* is the standard and most secure method for web applications to obtain authorized access to protected resources on behalf of a user. The process involves several interactions between the user, the App (e.g., a website), the authorization server (the XProtect Identity Provider), and the resource server (the API that holds the user's data).

Before using the flow, our App must be registered with the *XProtect Identity Provider* (IDP). This is done by adding the following top-level section to the App definition file

```
credentials:
  authorizationCodeFlow:
    clientName: "app-using-authorization-code-flow"
    clientScopes: [ "openid", "profile", "managementserver", "offline_access" ]
    clientRedirectURIs: [ "https://my-app/sign-in" ]
    postLogoutRedirectURIs: [ "https://my-app/sign-out" ]
```

For the registration we must provide a descriptive `clientName` and a list of `clientScopes` identifying which scopes our App will need access to. Also, we must specify a set of approved URIs that the authorization server (the IDP) can use to redirect the user's browser to. The `clientRedirectURIs` are the allowed redirects after having granted or denied permission. Likewise, the `postLogoutRedirectURIs` are the allowed redirects after a logout.

Below are listed the typical steps involed in the authorization code flow.

1. The App initiates the flow by redirecting the user's browser to the authorization server's authorization endpoint.
2. The authorization server displays a consent screen, asking the user if they grant the App the requested permissions.
3. If the user approves, the authorization server generates a temporary, single-use authorization code and redirects the user's browser back to the App's pre-registered redirect URI.
4. The App sends a request to exchanges the code for a token using the back-channel without involving the user's browser. Part of the request will be the client id and client secret of the App.
5. The authorization server validates the code, client id, and client secret. If everything is correct, it responds with an access and refresh token.
6. The App uses the access token to call the resource server's API endpoints.

The authorization server URL, client id and client secret are all available to the App through the environment variables named respecitively `SYSTEM_IDENTITY_PROVIDER`, `ACF_CLIENT_ID` and `ACF_CLIENT_SECRET`.

Working with Kafka topics

Kafka can be used to decouple services by allowing messages to be produced and consumed through topics in a robust, reliable and fault-tolerant maner. However, topics must be created before they can be used and this part can be handled automatically when installing an App, so we don't have to do it from code.

To have Kafka topics created automatically when installing an App, we must specify a section similar to the one shown below. Here we are listing those Kafka topics from which we plan to consume messages and also those topics to which we plan to produce messages.

```
messaging:
  kafka:
    consumerTopics:
    - name: "my-realm.my-topic"
    - name: "pub.another-realm.another-topic"
    producerTopics:
    - name: "my-realm.my-topic"
    - name: "pub.my-realm.my-new-topic"
```

Only the producer topics are created when the App is installed. The consumer topics are instead expected to already exist and if it turns out that they do not, then the installation of the App will fail.

The naming convention of topics follow the one used for services very closely, except here we use the `.` character to more clearly separate the parts. For services this was not possible because those names end up being used in contexts where `.` is an illegal character. A topic name consist of three parts following the pattern `[pub.]<realm>.<name>`, of which the first one is optional. For the topic named `pub.my-realm.my-new-topic`, the three parts are thus `pub`, `my-realm` and `my-new-`

`topic`. Here `pub` means that the topic is *public* and thus accessible for all other services in the system. If the `pub` part is left out, like if the name is `my-realm.my-topic`, then the topic is private and only accessible to services belonging to the realm named `my-realm`. The last and third part of the topic name is a descriptive and unique name identifying the topic within the realm it belongs to.

All producer topics must have a realm matching the realm of the App itself. For consumer topics, it is possible to use a topic with a different realm, but it then must be public.

There are several configuration parameters that can be applied when a producer topic is created. Here is an example listing all the parameters we can specify.

```
messaging:
  kafka:
    producerTopics:
    - name: "my-realm.my-topic"
      partitions: 1
      replicationFactor: 1
      cleanupPolicy: "delete"
      retention:
        milliseconds: 300000
        bytes: 1073741824
      segment:
        milliseconds: 60000
        bytes: 134217728
```

Configuring the topics in an optimal way is a fairly advanced topic. An intro to the core concepts like *partitions* and *replication factor* can be found here: https://kafka.apache.org/documentation/#intro_concepts_and_terms. Both `partitions` and `replicationFactor` use a default value of 1 if not specified.

The remaining parameters are related to how messages get deleted. Here is a list mapping each to their respecitive counterpart parameter in the Kafka documentation. If not specified (or set to zero), the values will default to global values configured for the Kafka cluster.

- `cleanupPolicy` : https://kafka.apache.org/documentation/#topicconfigs_cleanup.policy
- `retention.milliseconds` : https://kafka.apache.org/documentation/#topicconfigs_retention.ms
- `retention.bytes` : https://kafka.apache.org/documentation/#topicconfigs_retention.bytes
- `segment.milliseconds` : https://kafka.apache.org/documentation/#topicconfigs_segment.ms
- `segment.bytes` : https://kafka.apache.org/documentation/#topicconfigs_segment.bytes

When the App is uninstalled, all producer topics will be deleted.

Accessing topics from code is done using a language specific library that implements a Kafka client. There are many different libraries available, but they all have in common that they need something called a bootstrap server. This is the server (actually a comma separated list of servers) that allows the client to bootstrap the communication with the Kafka cluster. The bootstrap server will be made available to every container through the environment variable named `KAFKA_BOOTSTRAP_SERVER`.

Using a PostgreSQL database

Often an App will need to store some data (e.g., configuration data) which it can query at a later point. To help serve this very common need, a PostgreSQL database can be automatically created as part of

installing an App. We simply add a databases section to our app definition file like shown below. All we need to provide is just the name of the database to create.

```
databases:
  postgresql:
    name: "my-realm.my-db"
```

The prefix `my-realm` must match the value of the `realm` top-level property. This naming scheme will be enforced by the app-builder and has the purpose of avoiding naming conflicts between databases created within different realms. The connection details will be available to every container deployed by the App through the following environment variables.

```
PGDB_HOST      : The hostname of the PostgreSQL database cluster service.
PGDB_PORT      : The port number of the PostgreSQL database cluster service.
PGDB_DBNAME    : The name of the database.
PGDB_USERNAME  : The username needed when connecting to the database.
PGDB_PASSWORD  : The password needed when connecting to the database.
PGDB_URI       : The complete PostgreSQL connection string needed when working with
the database.
```

Optionally, we can also provide a schema (an array of SQL scripts) to apply on the newly created database. We can do this by providing the schema inline as shown below using the `schema` field.

```
databases:
  postgresql:
    name: "my-realm.my-app-db"
    schema:
    - |-
      create table if not exists tasks (
        id serial primary key,
        description text not null
      );
```

Or we can use the `schemaFiles` field, if it is preferred to have the schema in separate files. Note, that the files must be placed inside `files` in the custom definitions directory (see the section on *Custom Definition*).

```
databases:
  postgresql:
    name: "my-realm.my-app-db"
    schemaFiles:
    - "files/my-first-script.sql"
    - "files/my-second-script.sql"
```

When upgrading our App to a new version, the schema will be re-applied to the database and we can thus here change our database layout as needed. All containers will be stopped before the schema is applied and started again afterwards. Note that the provided SQL scripts are always executed in the order they are listed.

When the App is uninstalled, the database is automatically deleted.

Working with persistent storage

Most Apps that need to store state should use an actual database (e.g., a PostgreSQL database) for such a purpose. However, there are special cases where an App might need a persistent file storage for storing data. For such use cases the app definition file allows us to define a persistent volume which we can then mount into a container. If the system or container is restarted, the data will not be lost.

Here is an example app definition file setting up a volume to be mounted into a container.

```
name: "app-using-storage"
realm: "my-realm"
version: 1.0.0
description: "An App that uses persistent storage"

prerequisites:
  systemVersion: ">=0.3.9"

volumes:
- name: my-volume
  size: 30Gi
  class: media

services:
- name: my-realm-my-service
  routes:
  - name: my-route
    port: 80
    targetPort: 80

  containers:
  - name: my-container
    image: sandbox.io/hello-world/webserver:1.2.2
    volumeMounts:
    - name: my-volume
      path: /mnt/data
```

Here we first define a volume named `my-volume` instructing the system to find an available disk with a total capacity of at least 30Gi and which has been assigned the label `media`. Labels are assigned to disks as part of the system configuration and provides a way to group disks (actually partitions) by what kind of storage they are suitable for. The media label can as an example be assigned disks that are suitable for storing media data (video and audio data). If no disk is found, the installation of the App will fail.

In the example above, we mount the volume inside `my-container` at `/mnt/data`. Whatever the container stores here will continue to exist also after a restart of the container. Note that two containers can mount the same volume. Also note that a container which mounts a volume will be locked to run on the server which has the physical disk backing up the storage. Thus, such a container cannot be rescheduled automatically to run on another server in the system.

As shown earlier, we can use the `volumes` command to list the active volumes of an App and also what physical disk they are mounted from.

Replicating service workloads

Pods are the smallest deployable units of computing that we can create. All containers of one service are deployed as one pod. This makes them share storage and network resources. Specificially, all containers of a service have the same IP address.

Below is an example `service` specification in an app definition file. Here the service will deploy two containers together in one pod. Both containers are assigned the same IP address and thus must listen on two different ports for incoming requests. In the example, we route both port 80 and port 81 to the pod and we expect the two containers to serve these.

```
services:
- name: pub-my-realm-my-service
  replicas: 5
  routes:
  - name: my-route-1
    port: 80
    targetPort: 80
  - name: my-route-2
    port: 81
    targetPort: 81
  containers:
  - name: my-container-1
    image: sandbox.io/first-container:1.0.0
    ports:
    - name: web
      containerPort: 80
  - name: my-container-2
    image: sandbox.io/second-container:1.0.0
    ports:
    - name: api
      containerPort: 81
```

In the above example, we also specify the `replicas` field with a value of 5. This instructs the scheduler that we want the pod with the two containers to be replicated 5 times. So, in all, we will have 10 containers running in 5 different pods. When connecting to the service using the hostname `pub-my-realm-my-service`, the traffic will be routed to a random one of the 5 pods.

When scheduling multiple pods, you can control how they are spread across the cluster by adding a `topologySpreadConstraints` section to the service. Below is an example that will cause the pods to be equally spread across all nodes in the cluster.

```
topologySpreadConstraints:
- maxSkew: 1
  topologyKey: kubernetes.io/hostname
  whenUnsatisfiable: ScheduleAnyway
```

The `topologyKey` defines what domain you want your pods to be spread across. It could be regions (`topology.kubernetes.io/region`), zones (`topology.kubernetes.io/zone`), nodes (`kubernetes.io/hostname`), or even custom labels, depending on your infrastructure needs. Nodes that have a label with this key and identical values are considered to be in the same topology. We call

each instance of a topology a domain. The scheduler will try to put a balanced number of pods into each domain. The `maxSkew` parameter controls the allowed imbalance between topology domains. For instance, if you set `maxSkew` to 2, the difference in the number of pods between any two domains should not be more than 2. It gives you some flexibility in distribution while still ensuring a reasonable balance. Finally `whenUnsatisfiable` defines what should be done when the pod distribution rules cannot be satisfied. There are two options: `ScheduleAnyway` and `DoNotSchedule`.

Sometimes, it is important that we have exactly one pod per node in the cluster. If a node is added, a new pod should automatically be scheduled. Likewise, if a node is removed, the pod running on that node should be stopped and removed. To support this usecase, we can set the `replicas` field to `OnePerNode`.

When replicating with `OnePerNode`, it can be useful to enable traffic restrictions to only route traffic to endpoints within the node the traffic originated from. This will avoid a round trip via the cluster network and thus can help improve reliability and performance (network latency and throughput). We do this by setting the `internalTrafficPolicy` to `Local`.

```
services:
- name: pub-my-realm-my-service
  replicas: OnePerNode
  internalTrafficPolicy: Local
  routes:
  - name: my-route
    port: 80
    targetPort: 80
  containers:
  - name: my-container
    image: sandbox.io/my-container:1.0.0
```

The default value of `internalTrafficPolicy` is `Cluster`.

Specifying App resources

App resources specify the compute resources (CPU, memory, GPU) that each container in your App will request and be limited to when running in Kubernetes. This allows you to control how much of the cluster's resources your App can use, and helps ensure fair scheduling and stability.

You can define resource requests and limits for each container in your App definition file under the `resources` section:

```
services:
- name: my-service
  containers:
  - name: my-container
    image: sandbox.io/hello-world/webserver:1.2.2
    resources:
      limits:
        cpu: "1"
        memory: "1Gi"
        gpu: "1"
      requests:
        cpu: "500m"
```

```
        memory: "512Mi"
        gpu: "1"
```

Requests are the minimum amount of resources a container is guaranteed, used for scheduling, while limits are the maximum amount a container can use, preventing it from consuming excessive resources and affecting others.

CPU

- Format: `"500m"` (500 millicores, or 0.5 CPU), `"1"` (1 CPU core)
- If not specified, defaults to `500m` (limit) and `250m` (request).

Memory

- Format: `"512Mi"`, `"1Gi"`, `"2Gi"` (must use binary units: Ki, Mi, Gi, Ti, Pi, Ei)
- If not specified, defaults to `512Mi` (limit) and `256Mi` (request).

GPU

- Format: `"1"`, `"2"` (number of GPUs, optional)
- Only set if your workload requires GPU resources.

Note that if you omit the `resources` section, the default values will be used. It is important that appropriate requests and limits are configured to help the scheduler run your containers efficiently and preventing resource contention.

GPU resources are only available if your cluster supports them (e.g., with NVIDIA GPUs).

For more details, see the Kubernetes documentation on resource management

Routing ingress to service

We have seen how an App can define a service and expose it so that other services inside the system can connect to it. However, if we want to access one of these services from outside the system, then we need set up a specific rule that allows this. We use the term *ingress* for traffic coming from the outside and entering the system. Ingress will always arrive through the system IP address and then be routed to a specific service. Exposing a Web page or REST API to the outside world are typical examples for which ingress is needed.

Below is shown an example expanding a given route to also allow ingress traffic to reach a service. Specifically, all HTTP requests sent to the system IP address (or system host name) with root path `/xp/hello-world` will here be routed to `my-realm-my-service`.

```
name: "app-exposing-endpoint"
realm: "my-realm"
version: 1.0.0
description: "An App that exposes an endpoint at /my-webpage"

prerequisites:
  systemVersion: ">=0.3.9"

services:
- name: my-realm-my-service
  routes:
  - name: my-route
    port: 80
```

```
    targetPort: 80
    ingress:
      rootPath: "/xp/hello-world"
      rules:
      - path: "/"
        scopes: [ "managementserver" ]

  containers:
  - name: my-container
    image: sandbox.io/hello-world/webserver:1.2.2
```

We define a rule that will forward all sub paths of `/xp/hello-world` to `my-realm-my-service`. The path will be rewritten in the process to strip away the specified `rootPath`. So, a request to `/xp/hello-world/index.html` will reach the service with path `/index.html` and have the `X-Forwarded-Prefix` header field set to original request path. Also, in the example above, we require that all requests are authorized to have access to the `managementserver` scope. The request must thus in the authorization header provide an access token that has been issued by the XProtect Identity Provider and that has access to the scope named `managementserver`. If this is not the case, the request will be denied before routed to the service. If we leave out `scopes`, then authorization is disabled and no access token is required. Also note, that we above expose a service that is not public (it's name does not start with `pub-`). This means that even though we exposed the service outside the cluster, it cannot be accessed directly inside the cluster from services in other realms.

We structure ingress traffic and how to specify the `rootPath` by three different usecases; an *experience*, an *MFE* and an *API*. An *experience* is a webpage that an end-user might be interested in visiting, perpaphs by simply writing the URL in the browser. The `rootPath` for an experience is for this reason kept fairly open only requiring it not to start with `/mfe`, `/api` or `/legacy`.

An *MFE* (or microfrontend) is a UI component that can be re-used by multiple experiences. Here the `rootPath` must match the pattern `/mfe/<realm>/<name>`, where the `<realm>` is that of the App exposing the microfrontend and the `<name>` is free to choose. Note, that we can expose both the MFE itself (`main.js`) and an associated BFF (backend for frontend) through the same service by setting up individual rules.

```
ingress:
  rootPath: "/mfe/my-realm/my-mfe"
  rules:
  - path: "/main.js"
  - path: "/api/"
    scopes: [ "managementserver" ]
```

Finally, an *API* (e.g., a REST API) offered by a service can be exposed by having the `rootPath` match the pattern `/api/<realm>/<name>`. Here, the `<realm>` is that of the App exposing the service and the `<name>` is free to choose but recommended to follow that of the service. So, if we have a service named `pub-my-realm-my-service` then it is good pratice to expose it using the path `/api/my-realm/my-service`.

Importing a Grafana dashboard

A Grafana dashboard is a visual interface that displays data from various sources in a customizable way. It's essentially a collection of panels, each showing a specific piece of information, like graphs, charts, or tables, arranged to provide a comprehensive view of your system's health and performance.

We can import one or more dashboards to Grafana by adding a `dashboards` section simliar to the one shown below.

```
dashboards:
  grafana:
    import:
    - "files/my-first-dashboard.json"
    - "files/my-second-dashboard.json"
```

When the App is installed (and upgraded), the two dashboards will automatically be imported into Grafana. Note, that the files must be placed inside `files` in the custom definitions directory (see the section on *Custom Definition*). When the App is uninstalled, the dashboards are likewise deleted from Grafana.

The specified files are subject to Jinja template rendering. This means you can use values from the App definition file inside the dashboards. As an example, `{{ name }}` will expand to the name of the App. Since Grafana dashboards use the same notation for variables, this can cause problems. To avoid any template rendering, prefix and suffix you files with respecitively `{%- raw -%}` and `{%- endraw -%}`.

Collecting metrics from a container using Prometheus

Prometheus is an open-source monitoring and alerting toolkit that collects and stores time-series data. It's widely used in cloud-native environments, especially with Kubernetes, for monitoring applications and infrastructure. Prometheus works by scraping metrics from target systems and storing them with timestamps, allowing for powerful querying and alerting.

We also use Prometheus to collect metrics from Apps. Prometheus will connect to each App with regular intervals and ask the App for the current value of its metrics. Typically, your App would implement an HTTP endpoint hosting its metrics with path `/metrics`. There are several client libraries that can help you implement such an endpoint; e.g., for golang, you can use the following library https://github.com/prometheus/client_golang.

To let Prometheus know that it should scrape such an endpoint, you must name the relevant container port `app-metrics`. Here is an example of how to specify this for a given container. Specificially, with this example, Prometheus will start collecting metrics from the container using HTTP requests for `http://<cluster-ip-of-pod>:9100/metrics`.

```
containers:
- name: my-container
  image: sandbox.io/hello-world/webserver:1.2.2
  ports:
  - name: app-metrics
    containerPort: 9100
```

Extending client applications with Microfrontends

A microfrontend is a web development architectural approach where a large frontend application is broken down into smaller, independent, and deployable modules or *microfrontends*. These modules are often developed and maintained by separate teams. The shell (or container) is the overarching application that loads and orchestrates these microfrontends, providing a unified user interface and handling navigation and shared functionality.

An App can expose a microfrontend (MFE) to the shell by adding a registration similar to what is shown below.

```
microfrontends:
- id: "6933780b-0b1f-457c-9e62-232958ecd1d3"
  path: "/mfe/my-realm/maps/main.js"
  version: 2.0.2
  namespace: "my-realm-maps"
  name: "my-realm-maps"
  audience:
    requiredShellApi: ">=1.0.2"
    requiredPermissions: [ "maps.view", "maps.manage" ]
    targetExperiences: [ "administrator" ]
    targetApplications: [ "management-client" ]
  metadata:
    module: "main"
    bundler: "webpack"
    framework: "react"
```

The `path` must locate the `main.js` file of the microfrontend and it must start with `/mfe/<realm>/`, where the `<realm>` is that of the App exposing the microfrontend. The entire path `/mfe/my-realm/maps/main.js` must be exposed for ingress, so that a request for this file from the shell will reach the container serving the file. Here is an example of how such a service definition could look like.

```
services:
- name: my-realm-my-service
  routes:
  - name: my-route
    port: 80
    targetPort: 80
    ingress:
      rootPath: "/mfe/my-realm/maps"
      redirectPath: "/"
      rules:
      - path: "/main.js"
      - path: "/api/"
  containers:
  - name: my-container
    image: sandbox.io/my-app/my-service:1.0.0
```

Note, that we also here allow requests on `/mfe/my-realm/maps/api/...` to reach the container. Basically, this is to allow the microfrontend to make API calls to its BFF (backend-for-frontend). Instead of a single, general-purpose backend serving all clients, BFFs tailor the backend logic and data to the unique needs of each frontend. This approach optimizes performance, simplifies frontend development, and enhances the overall user experience.

Going back to the `microfrontends` section, we have for each microfrontend an `audience` section that identifies for whom it is designed and in which context it is intended to be used. The `requiredShellAPI` and `requiredPermissions` specify respectively the minimum shell API version and the permissions of the audience that are required for the microfrontend to be loaded. With

`targetExperiences` and `targetApplications`, the audience of the microfrontend can be made more specific. Possible values of an experience are `operator`, `administrator`, `integrator`, `supporter` and `developer`. For applications, possible values are `management-client`, `smart-client` and `web-client`.

Finally, there is a `metadata` section that the shell will use for deciding where and how to render the microfrontend. The ownership of what can go into the `metadata` section belongs to the shell and will evolve as newer versions of the shell becomes available. See the shell api documentation for more details.

Environment variables and command line arguments

In the App definition file, we can define environment variables for each container. We do this by expanding each container definition with an `env` section as shown below.

```
containers:
- name: my-container
  image: sandbox.io/hello-world/webserver:1.2.2
  env:
  - name: MY_FIRST_ENVIRONMENT_VARIABLE
    value: "The value of my first environment variable"
  - name: MY_SECOND_ENVIRONMENT_VARIABLE
    value: "The value of my second environment variable"
```

All containers have the following set of predefined environment variables that we use.

```
APP_NAME                  : The name of the App
APP_REALM                 : The realm of the App
APP_VERSION               : The version of the App
SYSTEM_IP                 : The external IP address of the System
SYSTEM_NAME               : The display name of the System
SYSTEM_UUID               : The unqiue ID of the System
SYSTEM_IDENTITY_PROVIDER  : The URL of the Identity Provider of the System
LEGACY_MANAGEMENT_SERVER  : The hostname of the Management Server in XProtect
LEGACY_USE_TLS            : Whether legacy system is configured to use TLS
(true/false)
```

For setting command line arguments, the App definition file can be extended with an `args` section which is basically a list of command line arguments that will be passed on to the container.

```
containers:
- name: my-container
  image: sandbox.io/hello-world/webserver:1.2.2
  args: [ "my-first-command-line-argument", "my-second-command-line-argument" ]
```

App values

App values are named values that we can define in the App definition file, which can be changed at the time of installation. Since they cannot be set when installing an App from the App Center, they are mostly useful for testing and debugging purposes. Below is shown a simple example using an App value to control the log level of a web server.

```yaml
name: "app-using-app-values"
realm: "my-realm"
version: 1.0.0
description: "An App that allows you to control the log level with an App value"

prerequisites:
  systemVersion: ">=0.3.9"

values:
  logLevel: error

services:
- name: my-realm-my-service
  routes:
  - name: my-route
    port: 80
    targetPort: 80
  containers:
  - name: my-container
    image: sandbox.io/hello-world/webserver:1.2.2
    args: [ "httpd-foreground", "-e", "{{ .Values.logLevel }}" ]
```

We define a new section named `values` which is a dictionary of named values. Specifically, we here have one value named `logLevel` which by default is set to `error`. When running the web server (`httpd-foreground`) in the container, we provide the command line arguments `-e` and `{{ .Values.logLevel }}`. At the time of installation, the argument `{{ .Values.logLevel }}` will expand to the value that we specified for `logLevel` in the `values` section. When installing the App through the App Center, the `error` log level will thus be used. However, this can be overwritten if installing the App using the app-builder. As an example, if we want to instead use the `debug` level, we can run the following command (the `install-from-file` command works the same way)

```
./app-builder.sh -f app-using-app-values.yaml install-from-repo --set logLevel=debug
```

Note that the we don't need to uninstall the App first if it is already installed. We can thus keep running the above command with different values and the container will automatically be restarted using the new value.

Custom definitions

Sometimes, the features supported by the App definition file is not enough for our App. In this case, we can provide our own custom definitions and control in much more detail how the App gets deployed. However, this also requires a much greater understanding of Kubernetes and Helm Charts.

Custom definitions are provided through a special directory that we can specify on the command line with the `-c` option.

```
App Builder - Building your own App for the App Center

Usage: app-builder [options] <command>
```

```
Options:
  -h               This message
  -f <file>        App YAML definition file to use; default is app-definition.yaml
  -c <directory>   Directory with custom definitions; default is the YAML definition
file without file extension
  -o <directory>   Directory to store packaged Helm Chart of App; default is current
directory
```

The structure of the folder must match that of a Helm Chart and the files will be merged in on top of the files generated by the App builder before the final package is built. All files with the extension `.jinja` will be rendered by the *jinja2 template engine* before copied and the resulting file will have its `.jinja` extension removed from its name. The directory named `files` is treated in a special way in that it is not being copied / rendered into the final package. This directory can be used for files that are only needed during the template rendering phase and not directly in the final package.