Kyma

In a nutshell

Kyma allows you to connect applications and third-party services in a cloud-native environment. Use it to create extensions for the existing systems, regardless of the language they are written in. Customize extensions with minimum effort and time devoted to learning their configuration details.

With Kyma in hand, you can focus purely on coding since it ensures the following out-of-the-box functionalities:

- Service-to-service communication and proxying (Istio Service Mesh)
- In-built monitoring, tracing, and logging (Grafana, Prometheus, Jaeger, Logspout, OK Log)
- Secure authentication and authorization (Dex, Service Identity, TLS, Role Based Access Control)
- The catalog of services to choose from (Service Catalog, Service Brokers)
- The development platform to run lightweight functions in a cost-efficient and scalable way (Serverless, Kubeless)
- The endpoint to register Events and APIs of external applications (Application Connector)

- The messaging channel to receive Events, enrich them, and trigger business flows using lambdas or services (Event Bus, NATS)
- CLI supported by the intuitive UI (Console)

Main features

Major open-source and cloud-native projects, such as Istio, NATS, Kubeless, and Prometheus, constitute the cornerstone of Kyma. Its uniqueness, however, lies in the "glue" that holds these components together. Kyma collects those cutting-edge solutions in one place and combines them with the in-house developed features that allow you to connect and extend your enterprise applications easily and intuitively.

Kyma allows you to extend and customize the functionality of your products in a quick and modern way, using serverless computing or microservice architecture. The extensions and customizations you create are decoupled from the core applications, which means that:

- Deployments are quick.
- Scaling is independent from the core applications.
- The changes you make can be easily reverted without causing downtime of the production system.

Last but not least, Kyma is highly cost-efficient. All Kyma native components and the connected open-source tools are written in Go. It ensures low memory consumption and reduced maintenance costs compared to applications written in other programming languages such as Java.

Technology stack

The entire solution is containerized and runs on a Kubernetes cluster. Customers can access it easily using a single sign on solution based on the Dex identity provider integrated with any OpenID Connect-compliant identity provider or a SAML2-based enterprise authentication server.

The communication between services is handled by the Istio Service Mesh component which enables security, traffic management, routing, resilience (retry, circuit breaker, timeouts),

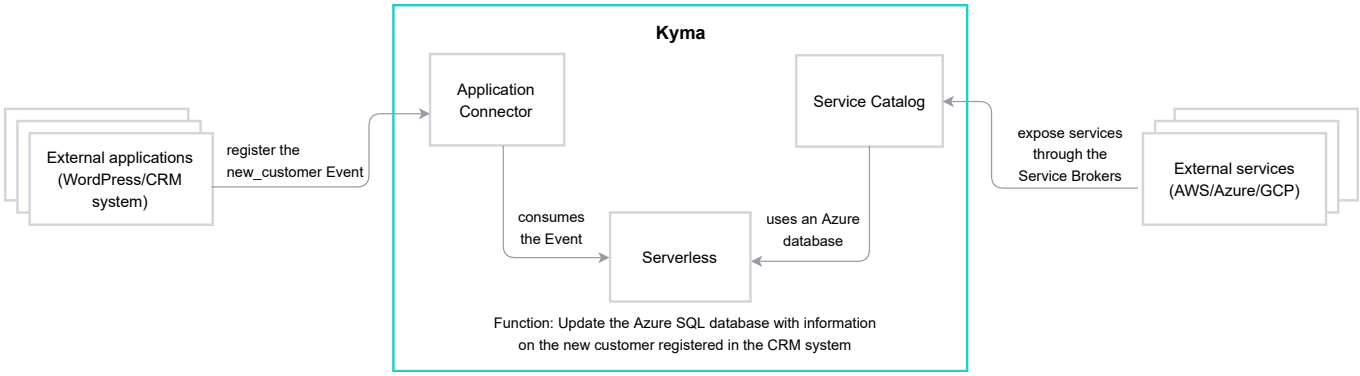
monitoring, and tracing without the need to change the application code. Build your applications using services provisioned by one of the many Service Brokers compatible with the Open Service Broker API, and monitor the speed and efficiency of your solutions using Prometheus, which gives you the most accurate and up-to-date monitoring data.

Key components

Kyma is built of numerous components but these three drive it forward:

- **Application Connector**:
  - Simplifies and secures the connection between external systems and Kyma
  - Registers external Events and APIs in the Service Catalog and simplifies the API usage
  - Provides asynchronous communication with services and lambdas deployed in Kyma through Events
  - Manages secure access to external systems
  - Provides monitoring and tracing capabilities to facilitate operational aspects
- **Serverless**:
  - Ensures quick deployments following a lambda function approach
  - Enables scaling independent of the core applications
  - Gives a possibility to revert changes without causing production system downtime
  - Supports the complete asynchronous programming model
  - Offers loose coupling of Event providers and consumers
  - Enables flexible application scalability and availability
- **Service Catalog**:
  - Connects services from external sources
  - Unifies the consumption of internal and external services thanks to compliance with the Open Service Broker standard
  - Provides a standardized approach to managing the API consumption and access
  - Eases the development effort by providing a catalog of API and Event documentation to support automatic client code generation

This basic use case shows how the three components work together in Kyma:

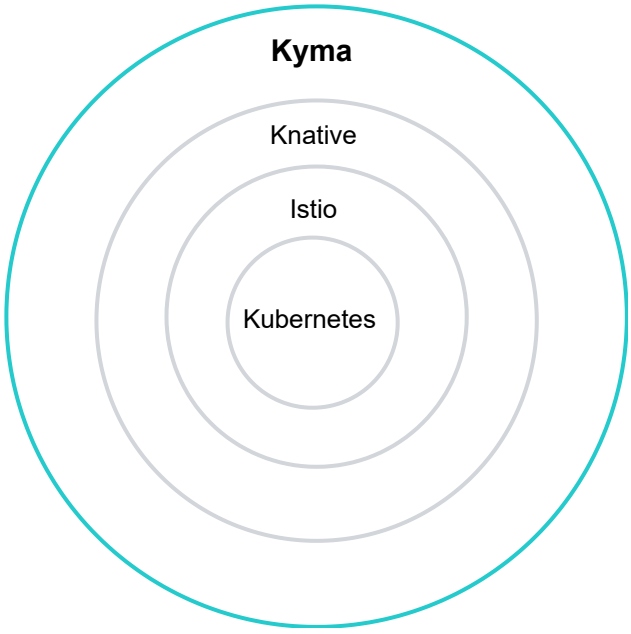


Kyma and Knative - brothers in arms

Integration with Knative is a step towards Kyma modularization and the "slimming" approach which aims to extract some out-of-the-box components and provide you with a more flexible choice of tools to use in Kyma.

Both Kyma and Knative are Kubernetes and Istio-based systems that offer development and eventing platforms. The main difference, however, is their focus. While Knative concentrates more on providing the building blocks for running serverless workloads, Kyma focuses on integrating those blocks with external services and applications.

The diagram shows dependencies between the components:



The nearest plan for Kyma and Knative cooperation is to replace Serverless in Kyma with the Knative technology. Other planned changes concerning Kyma and Knative cooperation involve providing configuration options to allow Istio deployed with Knative to work on

Kyma, and extracting Kyma eventing to fully integrate it with Knative eventing. The eventing integration will provide more flexibility on deciding which messaging implementation to use (NATS, Kafka, or any other).

How to start

When you already know what Kyma is and what components it consists of, you can start using it. Minikube allows you to run Kyma locally, develop, and test your solutions on a small scale before you push them to a cluster. With the Installation and Getting Started guides in hand, you can start developing in a matter of minutes.

Read, learn, and try on your own to:

- Install Kyma locally from the release
- Install Kyma locally from sources
- Install Kyma on a cluster
- Deploy a sample service locally
- Deploy a service on a cluster
- Develop a service locally without using Docker
- Publish a service Docker image and deploy it to Kyma
- Configure the Installer with override values for Helm charts
- Register a ClusterServiceBroker
- Create a new Application
- Get the client certificate
- Register a service
- Bind an Application to an Environment
- Trigger a lambda with events
- Call a registered external service from Kyma
- Expose custom metrics in Kyma

Components

Kyma is built on the foundation of the best and most advanced open-source projects which make up the components readily available for customers to use. This section describes the Kyma components.

Service Catalog

The Service Catalog lists all of the services available to Kyma users through the registered Service Brokers. Using the Service Catalog, you can provision new services in the Kyma Kubernetes cluster and create bindings between the provisioned service and an application.

Service Mesh

The Service Mesh is an infrastructure layer that handles service-to-service communication, proxying, service discovery, traceability, and security independent of the code of the services. Kyma uses the Istio Service Mesh that is customized for the specific needs of the implementation.

Security

Kyma security enforces RBAC (Role Based Access Control) in the cluster. Dex handles the identity management and identity provider integration. It allows you to integrate any OpenID Connect or SAML2-compliant identity provider with Kyma using connectors. Additionally, Dex provides a static user store which gives you more flexibility when managing access to your cluster.

Service Brokers

Service Brokers are Open Service Broker API-compatible servers that manage the lifecycle of one or more services. Each Service Broker registered in Kyma presents the services it offers to the Service Catalog. You can provision these services on a cluster level through the Service Catalog. Out of the box, Kyma comes with three Service Brokers. You can register more Open Service Broker API-compatible Service Brokers in Kyma and provision the services they offer using the Service Catalog.

Application Connector

The Application Connector is a proprietary Kyma solution. This endpoint is the Kyma side of the connection between Kyma and the external solutions. The Application Connector allows you to register the APIs and the Event Catalog, which lists all of the available events, of the connected solution. Additionally, the Application Connector proxies the calls from Kyma to external APIs in a secure way.

Event Bus

Kyma Event Bus receives Events from external solutions and triggers the business logic created with lambda functions and services in Kyma. The Event Bus is based on the NATS Streaming open source messaging system for cloud-native applications.

Serverless

The Kyma Serverless component allows you to reduce the implementation and operation effort of an application to the absolute minimum. Kyma Serverless provides a platform to run lightweight functions in a cost-efficient and scalable way using JavaScript and Node.js. Kyma Serverless is built on the Kubeless framework, which allows you to deploy lambda functions, and uses the NATS messaging system that monitors business events and triggers functions accordingly.

Monitoring

Kyma comes bundled with tools that give you the most accurate and up-to-date monitoring data. Prometheus open source monitoring and alerting toolkit provides this data, which is consumed by different add-ons, including Grafana for analytics and monitoring, and Alertmanager for handling alerts.

Tracing

The tracing in Kyma uses the Jaeger distributed tracing system. Use it to analyze performance by scrutinizing the path of the requests sent to and from your service. This information helps you optimize the latency and performance of your solution.

Logging

Logging in Kyma uses Logspout and OK Log. Use a plaintext or a regular expression to fetch logs from Pods using the OK Log UI.

Environments

An Environment is a custom Kyma security and organizational unit based on the concept of Kubernetes Namespaces. Kyma Environments allow you to divide the cluster into smaller units to use for different purposes, such as development and testing.

Kyma Environment is a user-created Namespace marked with the `env: "true"` label. The Kyma UI only displays the Namespaces marked with the `env: "true"` label.

Default Kyma Namespaces

Kyma comes configured with default Namespaces dedicated for system-related purposes. The user cannot modify or remove any of these Namespaces.

- `kyma-system` - This Namespace contains all of the Kyma Core components.
- `kyma-integration` - This Namespace contains all of the Application Connector components responsible for the integration of Kyma and external solutions.
- `kyma-installer` - This Namespace contains all of the Kyma Installer components, objects, and Secrets.
- `istio-system` - This Namespace contains all of the Istio-related components.

Environments in Kyma

Kyma comes with three Environments ready for you to use. These environments are:

- `production`
- `qa`
- `stage`

Create a new Environment

To create a new Environment, create a Namespace and mark it with the `env: "true"` label. Use this command to do that in a single step:

```
$ cat <<EOF | kubectl create -f -
apiVersion: v1
kind: Namespace
metadata:
  name: my-environment
  labels:
    env: "true"
EOF
```

Initially, the system deploys two template roles: `kyma-reader-role` and `kyma-admin-role`. The controller finds the template roles by filtering available roles in the namespace `kyma-system` by the label `env: "true"`. The controller copies these roles into the Environment.

Testing Kyma

For testing, the Kyma components use the Helm test concept. Place your test under the `templates` directory as a Pod definition that specifies a container with a given command to run.

Add a new test

The system bases tests on the Helm broker concept with one modification: adding a Pod label. Before you create a test, see the official Chart Tests documentation. Then, add the `"helm-chart-test": "true"` label to your Pod template.

See the following example of a test prepared for Dex:

```
# Chart tree
dex
├── Chart.yaml
├── README.md
├── templates
│   ├── tests
│   │   └── test-dex-connection.yaml
│   ├── dex-deployment.yaml
│   ├── dex-ingress.yaml
│   ├── dex-rbac-role.yaml
```

```
│   ├── dex-service.yaml
│   ├── pre-install-dex-account.yaml
│   ├── pre-install-dex-config-map.yaml
│   └── pre-install-dex-secrets.yaml
└── values.yaml
```

The test adds a new **test-dex-connection.yaml** under the `templates/tests` directory. This simple test calls the `Dex` endpoint with cURL, defined as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: "test-{{ template "fullname" . }}-connection-dex"
  annotations:
    "helm.sh/hook": test-success
  labels:
      "helm-chart-test": "true" # ! Our customization
spec:
  hostNetwork: true
  containers:
  - name: "test-{{ template "fullname" . }}-connection-dex"
    image: tutum/curl:alpine
    command: ["/usr/bin/curl"]
    args: [
      "--fail",
      "http://dex-service.{{ .Release.Namespace }}.svc.cluster.local:5556/.well-known
    ]
  restartPolicy: Never
```

Test execution

All tests created for charts under `/resources/core/` run automatically after starting Kyma. If any of the tests fail, the system prints the Pod logs in the terminal, then deletes all the Pods.

> **NOTE:** If you run Kyma locally, by default, the system does not take into account the test's exit code. As a result, the system does not terminate Kyma Docker container, and you can still access it. To force a termination in case of failing tests, use `--exit-on-test-fail` flag when executing `run.sh` script.

CI propagates the exit status of tests. If any test fails, the whole CI job fails as well.

Follow the same guidelines to add a test which is not a part of any `core` component. However, for test execution, see the **Run a test manually** section in this document.

Run a test manually

To run a test manually, use the `testing.sh` script located in the `/installation/scripts/` directory which runs all tests defined for `core` releases. If any of the tests fail, the system prints the Pod logs in the terminal, then deletes all the Pods.

Another option is to run a Helm test directly on your release.

```
$ helm test {your_release_name}
```

You can also run your test on custom releases. If you do this, remember to always delete the Pods after a test ends.

Charts

Kyma uses Helm charts to deliver single components and extensions, as well as the core components. This document contains information about the chart-related technical concepts, dependency management to use with Helm charts, and chart examples.

Manage dependencies with Init Containers

The **ADR 003: Init Containers for dependency management** document declares the use of Init Containers as the primary dependency mechanism.

Init Containers present a set of distinctive behaviors:

- They always run to completion.
- They start sequentially, only after the preceding Init Container completes successfully. If any of the Init Containers fails, the Pod restarts. This is always true, unless the `restartPolicy` equals `never`.

Readiness Probes ensure that the essential containers are ready to handle requests before you expose them. At a minimum, probes are defined for every container accessible from outside of the Pod. It is recommended to pair the Init Containers with readiness probes to provide a basic dependency management solution.

Examples

Here are some examples:

1. Generic

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
        readinessProbe:
          httpGet:
            path: /healthz
            port: 80
          initialDelaySeconds: 30
          timeoutSeconds: 1
```

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  initContainers:
  - name: init-myservice
    image: busybox
    command: ['sh', '-c', 'until nslookup nginx; do echo waiting for nginx; sleep 2;
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
```

2. Kyma

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: helm-broker
  labels:
    app: helm-broker
spec:
  replicas: 1
  selector:
    matchLabels:
      app: helm-broker
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
  template:
    metadata:
      labels:
        app: helm-broker
    spec:

      initContainers:
      - name: init-helm-broker
        image: eu.gcr.io/kyma-project/alpine-net:0.2.74
        command: ['sh', '-c', 'until nc -zv service-catalog-controller-manager.kyma-s

      containers:
      - name: helm-broker
        ports:
        - containerPort: 6699
        readinessProbe:
          tcpSocket:
            port: 6699
          failureThreshold: 3
          initialDelaySeconds: 10
          periodSeconds: 3
          successThreshold: 1
          timeoutSeconds: 2
```

Support for the Helm wait flag

High level Kyma components, such as **core**, come as Helm charts. These charts are installed as part of a single Helm release. To provide ordering for these core components, the Helm client runs with the `--wait` flag. As a result, Tiller waits for the readiness of all of the components, and then evaluates the readiness.

For `Deployments`, set the strategy to `RollingUpdate` and set the `MaxUnavailable` value to a number lower than the number of replicas. This setting is necessary, as readiness in Helm v2.10.0 is fulfilled if the number of replicas in ready state is not lower than the expected number of replicas:

```
ReadyReplicas >= TotalReplicas - MaxUnavailable
```

Chart installation details

The Tiller server performs the chart installation process. This is the order of operations that happen during the chart installation:

- resolve values
- recursively gather all templates with the corresponding values
- sort all templates
- render all templates
- separate hooks and manifests from files into sorted lists
- aggregate all valid manifests from all sub-charts into a single manifest file
- execute PreInstall hooks
- create a release using the ReleaseModule API and, if requested, wait for the actual readiness of the resources
- execute PostInstall hooks

Notes

All notes are based on Helm v2.10.0 implementation and are subject to change in feature releases.

- Regardless of how complex a chart is, and regardless of the number of sub-charts it references or consists of, it's always evaluated as one. This means that each Helm release is compiled into a single Kubernetes manifest file when applied on API server.
- Hooks are parsed in the same order as manifest files and returned as a single, global list for the entire chart. For each hook the weight is calculated as a part of this sort.
- Manifests are sorted by `Kind`. You can find the list and the order of the resources on the Kubernetes Tiller website.

Glossary

- **resource** is any document in a chart recognized by Helm or Tiller. This includes manifests, hooks, and notes.
- **template** is a valid Go template. Many of the resources are also Go templates.

Deploy with a private Docker registry

Docker is a free tool to deploy applications and servers. To run an application on Kyma, provide the application binary file as a Docker image located in a Docker registry. Use the `DockerHub` public registry to upload your Docker images for free access to the public. Use a private Docker registry to ensure privacy, increased security, and better availability.

This document shows how to deploy a Docker image from your private Docker registry to the Kyma cluster.

Details

The deployment to Kyma from a private registry differs from the deployment from a public registry. You must provide Secrets accessible in Kyma, and referenced in the `.yaml` deployment file. This section describes how to deploy an image from a private Docker registry to Kyma. Follow the deployment steps:

1. Create a Secret resource.
2. Write your deployment file.
3. Submit the file to the Kyma cluster.

Create a Secret for your private registry

A Secret resource passes your Docker registry credentials to the Kyma cluster in an encrypted form. For more information on Secrets, refer to the Kubernetes documentation.

To create a Secret resource for your Docker registry, run the following command:

```
kubectl create secret docker-registry {secret-name} --docker-server={registry FQN} --
```

Refer to the following example:

```
kubectl create secret docker-registry docker-registry-secret --docker-server=myregist
```

The Secret is associated with a specific Namespace. In the example, the Namespace is `production`. However, you can modify the Secret to point to any desired Namespace.

Write your deployment file

1. Create the deployment file with the `.yml` extension and name it `deployment.yml`.
2. Describe your deployment in the `.yml` file. Refer to the following example:

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  namespace: production # {production/stage/qa}
  name: my-deployment # Specify the deployment name.
  annotations:
    sidecar.istio.io/inject: true
spec:
  replicas: 3 # Specify your replica - how many instances you want from that deployme
  selector:
    matchLabels:
      app: app-name # Specify the app label. It is optional but it is a good practice
  template:
    metadata:
      labels:
        app: app-name # Specify app label. It is optional but it is a good practice.
        version: v1 # Specify your version.
    spec:
      containers:
      - name: container-name # Specify a meaningful container name.
        image: myregistry:5000/user-name/image-name:latest # Specify your image {regi
        ports:
          - containerPort: 80 # Specify the port to your image.
      imagePullSecrets:
        - name: docker-registry-secret # Specify the same Secret name you generated i
        - name: example-secret-name # Specify your Namespace Secret, named `example-s
```

3. Submit you deployment file using this command:

```
kubectl apply -f deployment.yml
```

Your deployment is now running on the Kyma cluster.

Install Kyma locally from the release

This Installation guide shows developers how to quickly deploy Kyma locally on a Mac or Linux from the latest release. Kyma installs locally using a proprietary installer based on a Kubernetes operator. The document provides prerequisites, instructions on how to install Kyma locally and verify the deployment, as well as the troubleshooting tips.

Prerequisites

To run Kyma locally, clone this Git repository to your machine and check out the latest release.

Additionally, download these tools:

- Minikube 0.28.2
- kubectl 1.10.0
- Helm 2.10.0
- jq

Virtualization:

- Hyperkit driver - Mac only
- VirtualBox - Linux only

> **NOTE:** To work with Kyma, use only the provided scripts and commands. Kyma does not work on a basic Minikube cluster that you can start using the `minikube start` command.

Set up certificates

Kyma comes with a local wildcard self-signed `server.crt` certificate that you can find under the `/installation/certs/workspace/raw/` directory of the `kyma` repository. Trust it on the OS

level for convenience.

Follow these steps to "always trust" the Kyma certificate on Mac:

1. Change the working directory to `installation`:

```
cd installation
```

2. Run this command:

```
sudo security add-trusted-cert -d -r trustRoot -k /Library/Keychains/System.keych
```

> **NOTE:** "Always trusting" the certificate does not work with Mozilla Firefox.

To access the Application Connector and connect an external solution to the local deployment of Kyma, you must add the certificate to the trusted certificate storage of your programming environment. Read the **Access Application Connector on local Kyma** in the **Application Connector** topic to learn more.

Install Kyma on Minikube

You can install Kyma either with all core subcomponents or only with the selected ones. This section describes how to install Kyma with all core subcomponents. To learn how to install only the specific ones, see the **Install subcomponents** document for details.

> **NOTE:** Running the installation script deletes any previously existing cluster from your Minikube.

> **NOTE:** Logging and Monitoring subcomponents are not included by default when you install Kyma on Minikube. You can install them using the instructions provided here.

To install Kyma, follow these steps:

1. Change the working directory to `installation`:

```
cd installation
```

2. Use the following command to run Kubernetes locally using Minikube:

```
./scripts/minikube.sh --domain "kyma.local" --vm-driver "hyperkit"
```

3. Wait until the `kube-dns` Pod is ready. Run this script to setup Tiller:

```
./scripts/install-tiller.sh
```

4. Go to this page and choose the release you want to use.
5. Export the version you chose as an environment variable. Run:

```
export LATEST={KYMA_RELEASE_VERSION}
```

6. Configure the Kyma installation using the local configuration file from the $LATEST release:

```
kubectl apply -f https://github.com/kyma-project/kyma/releases/download/$LATEST/k
```

7. To trigger the installation process, label the `kyma-installation` custom resource:

```
kubectl label installation/kyma-installation action=install
```

8. By default, the Kyma installation is a background process, which allows you to perform other tasks in the terminal window. Nevertheless, you can track the progress of the installation by running this script:

```
./scripts/is-installed.sh
```

Read the **Reinstall Kyma** document to learn how to reinstall Kyma without deleting the cluster from Minikube. To learn how to test Kyma, see the **Testing Kyma** document.

Verify the deployment

Follow the guidelines in the subsections to confirm that your Kubernetes API Server is up and running as expected.

Verify the installation status using the is-installed.sh script

The `is-installed.sh` script is designed to give you clear information about the Kyma installation. Run it at any point to get the current installation status, or to find out whether the installation is successful.

If the script indicates that the installation failed, try to install Kyma again by re-running the `run.sh` script.

If the installation fails in a reproducible manner, don't hesitate to create a GitHub issue in the project or reach out to the "installation" Slack channel to get direct support from the community.

Access the Kyma console

Access your local Kyma instance through this link.

- Click **Login with Email** and sign in with the **admin@kyma.cx** email address. Use the password contained in the `admin-user` Secret located in the `kyma-system` Namespace. To get the password, run:

```
kubectl get secret admin-user -n kyma-system -o jsonpath="{.data.password}" | base64
```

- Click the **Environments** section and select an Environment from the drop-down menu to explore Kyma further.

Access the Kubernetes Dashboard

Additionally, confirm that you can access your Kubernetes Dashboard. Run the following command to check the IP address on which Minikube is running:

```
minikube ip
```

The address of your Kubernetes Dashboard looks similar to this:

```
http://{ip-address}:30000
```

See the example of the website address:

```
http://192.168.64.44:30000
```

Enable Horizontal Pod Autoscaler (HPA)

By default, the Horizontal Pod Autoscaler is not enabled in your local Kyma installation, so you need to enable it manually.

Kyma uses the autoscaling/v1 stable version, which only provides support for CPU autoscaling. Once enabled, HPA automatically scales the number of lambda function Pods based on observed CPU utilization.

> **NOTE:** The autoscaling/v1 version does not support custom metrics. To use such metrics, you need the autoscaling/v2beta2 version.

To enable Horizontal Pod Autoscaler, follow these steps:

1. Enable the metrics server for resource metrics by running the following command:

   ```
   minikube addons enable metrics-server
   ```

2. Verify if the metrics server is active by checking the list of addons:

   ```
   minikube addons list
   ```

Stop and restart Kyma without reinstalling

Use the `minikube.sh` script to restart the Minikube cluster without reinstalling Kyma. Follow these steps to stop and restart your cluster:

1. Stop the Minikube cluster with Kyma installed. Run:

   ```
   minikube stop
   ```

2. Restart the cluster without reinstalling Kyma. Run:

   ```
   ./scripts/minikube.sh --domain "kyma.local" --vm-driver "hyperkit"
   ```

The script discovers that a minikube cluster is initialized and asks if you want to delete it. Answering `no` causes the script to start the Minikube cluster and restarts all of the previously installed components. Even though this procedure takes some time, it is faster than a clean installation as you don't download all of the required Docker images.

To verify that the restart is successful, run this command and check if all Pods have the `RUNNING` status:

```
kubectl get pods --all-namespaces
```

Troubleshooting

If the Installer does not respond as expected, check the installation status using the `is-installed.sh` script with the `--verbose` flag added. Run:

```
scripts/is-installed.sh --verbose
```

If the installation is successful but a component does not behave in an expected way, see if all deployed Pods are running. Run this command:

```
kubectl get pods --all-namespaces
```

The command retrieves all Pods from all Namespaces, the status of the Pods, and their instance numbers. Check if the STATUS column shows Running for all Pods. If any of the Pods that you require do not start successfully, perform the installation again. If the problem persists, don't hesitate to create a GitHub issue or reach out to the "installation" Slack channel to get direct support from the community.

Install Kyma locally from sources

This Installation guide shows developers how to quickly deploy Kyma on a Mac or Linux from local sources. Follow it if you want to use Kyma for development purposes.

Kyma installs locally using a proprietary installer based on a Kubernetes operator. The document describes only the installation part. For prerequisites, certificates setup, deployment validation, and troubleshooting steps, see the **Install Kyma locally from the release** document.

Install Kyma

To run Kyma locally, clone this Git repository to your machine.

To start the local installation, run the following command:

```
./installation/cmd/run.sh
```

This script sets up default parameters, starts Minikube, builds the Kyma-Installer, generates local configuration, creates the Installation custom resource, and sets up the Installer.

> **NOTE:** See the **Local installation scripts** document for a detailed explanation of the `run.sh` script and the subscripts it triggers.

You can execute the `installation/cmd/run.sh` script with the following parameters:

- `--skip-minikube-start` which skips the execution of the `installation/scripts/minikube.sh` script.
- `--vm-driver` which points to either `virtualbox` or `hyperkit`, depending on your operating system.

Read the **Reinstall Kyma** document to learn how to reinstall Kyma without deleting the cluster from Minikube. To learn how to test Kyma, see the **Testing Kyma** document.

Install Kyma on a GKE cluster

This Installation guide shows developers how to quickly deploy Kyma on a Google Kubernetes Engine (GKE) cluster. Kyma installs on a cluster using a proprietary installer based on a Kubernetes operator.

Prerequisites

- A domain for your GKE cluster
- Google Cloud Platform (GCP) project
- Docker
- Docker Hub account
- gcloud

> **NOTE:** If you don't own a domain which you can use or you don't want to assign a domain to a cluster, see the **Install Kyma on a GKE cluster with wildcard DNS** document which shows you how to create a cluster-based playground environment using a wildcard DNS provided by xip.io.

DNS setup

Delegate the management of your domain to Google Cloud DNS. Follow these steps:

1. Export the domain name, project name, and DNS zone name as environment variables. Run the commands listed below:

```
export DOMAIN={YOUR_SUBDOMAIN}
export DNS_NAME={YOUR_DOMAIN}.
export PROJECT={YOUR_GOOGLE_PROJECT}
export DNS_ZONE={YOUR_DNS_ZONE}
```

2. Create a DNS-managed zone in your Google project. Run:

```
gcloud dns --project=$PROJECT managed-zones create $DNS_ZONE --description= --dn
```

   Alternatively, create it through the GCP UI. Navigate go to **Network Services** in the **Network** section, click **Cloud DNS** and select **Create Zone**.

3. Delegate your domain to Google name servers.
   - Get the list of the name servers from the zone details. This is a sample list:

```
ns-cloud-b1.googledomains.com.
ns-cloud-b2.googledomains.com.
ns-cloud-b3.googledomains.com.
ns-cloud-b4.googledomains.com.
```

- Set up your domain to use these name servers.

4. Check if everything is set up correctly and your domain is managed by Google name servers. Run:

```
host -t ns $DNS_NAME
```

A successful response returns the list of the name servers you fetched from GCP.

Get the TLS certificate

1. Create a folder for certificates. Run:

```
mkdir letsencrypt
```

2. Create a new service account and assign it to the `dns.admin` role. Run these commands:

```
gcloud iam service-accounts create dnsmanager --display-name "dnsmanager"
```

```
gcloud projects add-iam-policy-binding $PROJECT         --member serviceAccount:
```

3. Generate an access key for this account in the `letsencrypt` folder. Run:

```
gcloud iam service-accounts keys create ./letsencrypt/key.json --iam-account dns
```

4. Run the Certbot Docker image with the `letsencrypt` folder mounted. Certbot uses the key to apply DNS challenge for the certificate request and stores the TLS certificates in that folder. Run:

```
docker run -it --name certbot --rm         -v "$(pwd)/letsencrypt:/etc/letsencry
```

5. Export the certificate and key as environment variables. Run these commands:

```
export TLS_CERT=$(cat ./letsencrypt/live/$DOMAIN/fullchain.pem | base64 | sed 's
export TLS_KEY=$(cat ./letsencrypt/live/$DOMAIN/privkey.pem | base64 | sed 's/ /
```

Prepare the GKE cluster

1. Select a name for your cluster and set it as an environment variable. Run:

```
export CLUSTER_NAME={CLUSTER_NAME_YOU_WANT}
```

2. Create a cluster in the `europe-west1` region. Run:

```
gcloud container --project "$PROJECT" clusters     create "$CLUSTER_NAME" --zone
```

3. Install Tiller on your GKE cluster. Run:

```
kubectl apply -f installation/resources/tiller.yaml
```

Prepare the installation configuration file

Using the latest GitHub release

1. Go to this page and choose the release you want to use.
2. Export the version you chose as an environment variable. Run:

```
export LATEST={KYMA_RELEASE_VERSION}
```

3. Download the `kyma-config-cluster` file from the release you chose. Run:

```
wget https://github.com/kyma-project/kyma/releases/download/$LATEST/kyma-config-c
```

4. Update the file with the values from your environment variables. Run:

```
cat kyma-config-cluster.yaml | sed -e "s/__DOMAIN__/$DOMAIN/g" |sed -e "s/__TLS_
```

5. The output of this operation is the `my_kyma.yaml` file. Use it to deploy Kyma on your GKE cluster.

Using your own image

1. Checkout kyma-project and enter the root folder.
2. Build an image that is based on the current Installer image and includes the current installation and resources charts. Run:

```
docker build -t kyma-installer:latest -f tools/kyma-installer/kyma.Dockerfile .
```

3. Push the image to your Docker Hub:

```
docker tag kyma-installer:latest [YOUR_DOCKER_LOGIN]/kyma-installer:latest
```

```
docker push [YOUR_DOCKER_LOGIN]/kyma-installer:latest
```

4. Prepare the deployment file:

```
cat installation/resources/installer.yaml <(echo -e "
---") installation/resources/installer-config-cluster.yaml.tpl  <(echo -e "
---") installation/resources/installer-cr-cluster.yaml.tpl | sed -e "s/__DOMAIN__
```

5. The output of this operation is the `my_kyma.yaml` file. Modify it to fetch the proper image with the changes you made ([YOUR_DOCKER_LOGIN]/kyma-installer:latest). Use the modified file to deploy Kyma on your GKE cluster.

Deploy Kyma

1. Configure kubectl to use your new cluster. Run: add yourself as the cluster admin, and deploy Kyma Installer with your configuration.

```
gcloud container clusters get-credentials $CLUSTER_NAME --zone europe-west1-b --
```

2. Add your account as the cluster administrator:

```
kubectl create clusterrolebinding cluster-admin-binding --clusterrole=cluster-ad
```

3. Deploy Kyma using the `my-kyma` custom configuration file you created. Run:

```
kubectl apply -f my-kyma.yaml
```

4. Check if the Pods of Tiller and the Kyma Installer are running:

```
kubectl get pods --all-namespaces
```

5. Start Kyma installation:

```
kubectl label installation/kyma-installation action=install
```

6. To watch the installation progress, run:

```
kubectl get pods --all-namespaces -w
```

Configure DNS for the cluster load balancer

Run these commands:

```
export EXTERNAL_PUBLIC_IP=$(kubectl get service -n istio-system istio-ingressgateway

export REMOTE_ENV_IP=$(kubectl get service -n kyma-system application-connector-nginx

gcloud dns --project=$PROJECT record-sets transaction start --zone=$DNS_ZONE

gcloud dns --project=$PROJECT record-sets transaction add $EXTERNAL_PUBLIC_IP --name=

gcloud dns --project=$PROJECT record-sets transaction add $REMOTE_ENV_IP --name=gatew
```

```
gcloud dns --project=$PROJECT record-sets transaction execute --zone=$DNS_ZONE
```

Prepare your Kyma deployment for production use

To use the cluster in a production environment, it is recommended you configure a new server-side certificate for the Application Connector and replace the placeholder certificate it installs with. If you don't generate a new certificate, the system uses the placeholder certificate. As a result, the security of your implementation is compromised.

Follow this steps to configure a new, more secure certificate suitable for production use.

1. Generate a new certificate and key. Run:

```
openssl req -new -newkey rsa:4096 -nodes -keyout ca.key -out ca.csr -subj "/C=PL

openssl x509 -req -sha256 -days 365 -in ca.csr -signkey ca.key -out ca.pem
```

2. Export the certificate and key to environment variables:

```
export AC_CRT=$(cat ./ca.pem | base64 | base64)
export AC_KEY=$(cat ./ca.key | base64 | base64)
```

3. Prepare installation file with the following command:

```
cat kyma-config-cluster.yaml | sed -e "s/__DOMAIN__/$DOMAIN/g" |sed -e "s/__TLS_
```

Local installation scripts

This document extends the **Install Kyma locally from sources** guide with a detailed breakdown of the alternative local installation method which is the `run.sh` script.

The following snippet is the main element of the `run.sh` script:

```
if [[ ! $SKIP_MINIKUBE_START ]]; then
    bash $CURRENT_DIR/../scripts/minikube.sh --domain "$DOMAIN" --vm-driver "$VM_DRIV
fi

bash $CURRENT_DIR/../scripts/build-kyma-installer.sh --vm-driver "$VM_DRIVER"

if [ -z "$CR_PATH" ]; then

    TMPDIR=`mktemp -d "$CURRENT_DIR/../../temp-XXXXXXXXXX"`
    CR_PATH="$TMPDIR/installer-cr-local.yaml"

    bash $CURRENT_DIR/../scripts/create-cr.sh --output "$CR_PATH" --domain "$DOMAIN"
    bash $CURRENT_DIR/../scripts/installer.sh --local --cr "$CR_PATH"

    rm -rf $TMPDIR
else
    bash $CURRENT_DIR/../scripts/installer.sh --cr "$CR_PATH"
fi
```

Subsequent sections provide details of all involved subscripts, in the order in which the `run.sh` script triggers them.

The minikube.sh script

> **NOTE:** To work with Kyma, use only the provided scripts and commands. Kyma does not work on a basic Minikube cluster that you can start using the `minikube start` command.

The purpose of the `installation/scripts/minikube.sh` script is to configure and start Minikube. The script also checks if your development environment is configured to handle the Kyma installation. This includes checking Minikube and kubectl versions. If Minikube is already initialized, the system prompts you to agree to remove the previous Minikube cluster.

- If you plan to perform a clean installation, answer `yes`.
- If you installed Kyma to your Minikube cluster and then stopped the cluster using the `minikube stop` command, answer `no`. This allows you to start the cluster again without reinstalling Kyma.

Minikube is configured to disable the default Nginx Ingress Controller.

> **NOTE:** For the complete list of parameters passed to the `minikube start` command, refer to the `installation/scripts/minikube.sh` script.

Once Minikube is up and running, the script adds local installation entries to `/etc/hosts`.

The build-kyma-installer.sh script

The Installer is an application based on a Kubernetes operator. Its purpose is to install Helm charts defined in the Installation custom resource. The Kyma-Installer is a Docker image that bundles the Installer binary with Kyma charts.

The `installation/scripts/build-kyma-installer.sh` script extracts the Kyma-Installer image name from the `installer.yaml` deployment file and uses it to build a Docker image inside Minikube. This image contains local Kyma sources from the `resources` folder.

> **NOTE:** For the Kyma-Installer Docker image details, refer to the `tools/kyma-installer/kyma.Dockerfile` file.

The configure-azure-broker.sh script

The `configure-azure-broker.sh` script configures Azure Broker, an optional subcomponent of the `core` deployment.

The Azure Broker subcomponent is part of the `core` deployment that provisions managed services in the Microsoft Azure cloud. To enable the Azure Broker, export the following environment variables:

- AZURE_BROKER_SUBSCRIPTION_ID
- AZURE_BROKER_TENANT_ID
- AZURE_BROKER_CLIENT_ID
- AZURE_BROKER_CLIENT_SECRET

> **NOTE:** You need to export above environment variables before executing the `installation/cmd/run.sh` script. As the Azure credentials are converted to a Kubernetes Secret, make sure the exported values are base64-encoded.

The create-cr.sh script

The `installation/scripts/create-cr.sh` script prepares the Installation custom resource from the `installation/resources/installer-cr.yaml.tpl` template. The local installation scenario

uses the default Installation custom resource. The Kyma-Installer already contains local Kyma resources bundled, thus `url` is ignored by the Installer component.

> **NOTE:** For the Installation custom resource details, refer to the **Installation** document.

The installer.sh script

The `installation/scripts/installer.sh` script creates the default RBAC role, installs Tiller, and deploys the Kyma-Installer component.

> **NOTE:** For the Kyma Installer deployment details, refer to the `installation/resources/installer.yaml` file.

The script applies the Installation custom resource and marks it with the `action=install` label, which triggers the Kyma installation.

> **NOTE:** The Kyma installation runs in the background. Execute the `./installation/scripts/is-installed.sh` script to follow the installation process.

The is-installed.sh script

The `installation/scripts/is-installed.sh` script shows the status of Kyma installation in real time. The script checks the status of the Installation custom resource. When it detects that the status changed to `Installed`, the script exits. If you define a timeout period and the status doesn't change to `Installed` within that period, the script fetches the installer logs. If you don't set a timeout period, the script waits for the change of the status until you terminate it.

Install subcomponents

It is up to you to decide which subcomponents you install as part of the `core` release. By default, most of the core subcomponents are enabled. If you want to install only specific subcomponents, follow the steps that you need to perform before the local and cluster installation.

Install subcomponents locally

To specify whether to install a given core subcomponent on Minikube, use the `manage-component.sh` script before you trigger the Kyma installation. The script consumes two parameters:

- the name of the core subcomponent
- a Boolean value that determines whether to install the subcomponent (`true`) or not (`false`)

Example:

To enable the Azure Broker subcomponent, run the following command:

```
scripts/manage-component.sh azure-broker true
```

Alternatively, to disable the Azure Broker subcomponent, run this command:

```
scripts/manage-component.sh azure-broker false
```

Install subcomponents on a cluster

Install subcomponents on a cluster based on Helm conditions described in the `requirements.yaml` file. Read more about the fields in the `requirements.yaml` file here.

To specify whether to install a given core subcomponent, provide override values before you trigger the installation.

Example:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kyma-sub-components
  namespace: kyma-installer
  labels:
    installer: overrides
data:
  azure-broker.enabled: "true"
```

> **NOTE:** Some subcomponents can require additional configuration to work properly.

Specify subcomponents versions

Versions of the Kyma components are specified in the `values.yaml` file in charts. Two properties, `version` and `dir`, describe each component version. The first one defines the actual docker image tag. The second property describes the directory under which the tagged image is pushed. It is optional and is followed by a forward slash (/).

Possible values of the `dir` property:

- `pr/` contains images built from the pull request
- `develop/` contains images built from the `master` branch
- `rc/` contains images built for a pre-release
- (empty) contains images built for a release

To override subcomponents versions during Kyma startup, create the `versions-overrides.env` file in the `installation` directory.

The example overrides the Namespace Controller component and sets the image version to `0.0.1`, based on the version from the `develop` directory.

Example:

```
global.namespace_controller.dir=develop/
global.namespace_controller.version=0.0.1
```

Reinstall Kyma

The custom scripts allow you to remove Kyma from a Minikube cluster and reinstall Kyma without removing the cluster.

> **NOTE:** These scripts do not delete the cluster from your Minikube. This allows you to quickly reinstall Kyma.

1. Use the `clean-up.sh` script to uninstall Kyma from the cluster. Run:

```
scripts/clean-up.sh
```

2. Run this script to reinstall Kyma on an existing cluster:

```
cmd/run.sh --skip-minikube-start
```

Install Kyma on a GKE cluster with wildcard DNS

If you want to try Kyma in a cluster environment without assigning the cluster to a domain you own, you can use `xip.io` which provides a wildcard DNS for any IP address. Such a scenario requires using a self-signed TLS certificate.

This solution is not suitable for a production environment but makes for a great playground which allows you to get to know the product better.

Prerequisites

The prerequisites match these listed in this document. However, you don't need to prepare a domain for your cluster as it is replaced by a wildcard DNS provided by `xip.io` .

> **NOTE:** This feature requires Kyma version 0.6 or higher.

Installation

The installation process follows the steps outlined in the Install Kyma on a GKE cluster document. Follow this section to prepare your cluster.

In addition to exporting the desired cluster name as an environment variable, make sure to export your GCP project name. Run:

```
export PROJECT={YOUR_GCP_PROJECT_NAME}
```

When you install Kyma with the wildcard DNS, you can use one of two approaches to allocating the required IP addresses for your cluster:

- Dynamic IP allocation - can be used with Knative eventing and serverless, but disables the Application Connector.

- Manual IP allocation - cannot be used with Knative eventing and serverless, but leaves the Application Connector functional.

Follow the respective instructions to deploy a cluster Kyma cluster with wildcard DNS which uses the IP allocation approach of your choice.

Dynamic IP allocation

1. Use this command to prepare a configuration file that deploys Kyma with `xip.io` providing a wildcard DNS:

```
(cat installation/resources/installer.yaml ; echo "
---" ; cat installation/resources/installer-config-cluster.yaml.tpl ; echo "
---" ; cat installation/resources/installer-cr-cluster-xip-io.yaml.tpl) | sed -e
```

> **NOTE:** Using this approach disables the Application Connector.

2. Follow these instructions to install Kyma using the configuration file you prepared.

Manual IP allocation

1. Get public IP addresses for the load balancer of the GKE cluster to which you deploy Kyma and for the load balancer of the Application Connector.
   - Export the `PUBLIC_IP_ADDRESS_NAME` and the `APP_CONNECTOR_IP_ADDRESS_NAME` environment variables. This defines the names of the reserved public IP addresses in your GCP project. Run:

```
export PUBLIC_IP_ADDRESS_NAME={GCP_COMPLIANT_PUBLIC_IP_ADDRESS_NAME}
export APP_CONNECTOR_IP_ADDRESS_NAME={GCP_COMPLIANT_APP_CONNECTOR_IP_ADDRESS_
```

> **NOTE:** The name you set for the reserved public IP address must start with a lowercase letter followed by up to 62 lowercase letters, numbers, or hyphens, and cannot end with a hyphen.

   - Run these commands to reserve public IP addresses for the load balancer of your cluster and the load balancer of the Application Connector.

```
gcloud beta compute --project=$PROJECT addresses create $PUBLIC_IP_ADDRESS_NA
gcloud beta compute --project=$PROJECT addresses create $APP_CONNECTOR_IP_ADI
```

> **NOTE:** The region in which you reserve IP addresses must match the region of your GKE cluster.

- Set the reserved IP addresses as `EXTERNAL_PUBLIC_IP` and `CONNCETOR_IP` environment variables. Run:

```
export EXTERNAL_PUBLIC_IP=$(gcloud compute addresses list --project=$PROJECT
export CONNECTOR_IP=$(gcloud compute addresses list --project=$PROJECT --filt
```

2. Use this command to prepare a configuration file that deploys Kyma with `xip.io` providing a wildcard DNS:

```
(cat installation/resources/installer.yaml ; echo "
---" ; cat installation/resources/installer-config-cluster.yaml.tpl ; echo "
---" ; cat installation/resources/installer-cr-cluster-xip-io.yaml.tpl) | sed -e
```

3. Follow these instructions to install Kyma using the configuration file you prepared.

Add the xip.io self-signed certificate to your OS trusted certificates

After the installation, add the custom Kyma `xip.io` self-signed certificate to the trusted certificates of your OS. For MacOS run:

```
tmpfile=$(mktemp /tmp/temp-cert.XXXXXX) && kubectl get configmap cluster-certificate-
```

Access the cluster

To access your cluster, use the wildcard DNS provided by `xip.io` as the domain of the cluster. To get this information, run:

```
kubectl get cm installation-config-overrides -n kyma-installer -o jsonpath='{.data.gl
```

A successful response returns the cluster domain following this format:

```
{WILDCARD_DNS}.xip.io
```

Access your cluster under this address:

```
https://console.{WILDCARD_DNS}.xip.io
```

> **NOTE:** To log in to your cluster, use the default `admin` static user. To learn how to get the login details for this user, see this document.

Installation with custom Istio deployment

You can use Kyma with a custom deployment of Istio that you installed in the target environment. To enable such implementation, remove Istio from the list of components that install with Kyma. The version of your Istio deployment must match the version that Kyma currently supports.

In the installation process, the Installer applies a custom patch to every Istio deployment. This is a mandatory step.

> **NOTE:** To learn more, read the **Istio patch** document in the **Service Mesh** documentation topic.

Prerequisites

- A live Istio version compatible with the version currently supported by Kyma. To check the supported version, see the value of the `REQUIRED_ISTIO_VERSION` environmental variable in the `resources/istio-kyma-patch/templates/job.yaml` file.

  > **NOTE:** Follow this quick start guide to learn how to install and configure Istio on a Kubernetes cluster.

- Security enabled in your Istio deployment. To verify if security is enabled, check if the `policies.authentication.istio.io` custom resource exists in the cluster.
- Mutual TLS (mTLS) disabled in your Istio deployment.
- Kyma downloaded from the latest release.

Local installation

1. Remove these lines from the `kyma-config-local.yaml` file:

```
name: "istio"
namespace: "istio-system"
```

2. Follow the installation steps described in the **Install Kyma locally from the release** document.

Cluster installation

1. Remove these lines from the `kyma-config-cluster.yaml` file:

```
name: "istio"
namespace: "istio-system"
```

2. Follow the installation steps described in the **Install Kyma on a GKE cluster** document.

Verify the installation

1. Check if all Pods are running in the `kyma-system` Namespace:

```
kubectl get pods -n kyma-system
```

2. Sign in to the Kyma Console using the `admin@kyma.cx` as described in the **Install Kyma locally from the release** document.

Installation with Knative

You can install Kyma with Knative and use its solutions for handling events and serverless functions.

> **NOTE:** You can't install Kyma with Knative on clusters with a pre-allocated ingress gateway IP address.

> **NOTE:** Knative intagration requires Kyma 0.6 or higher.

Knative with local deployment from release

When you install Kyma locally from a release, follow this guide and run the following command after you complete step 6:

```
kubectl -n kyma-installer patch configmap installation-config-overrides -p '{"data":
```

Knative with local deployment from sources

When you install Kyma locally from sources, add the `--knative` argument to the `run.sh` script. Run this command:

```
./run.sh --knative
```

Knative with a GKE cluster deployment from release

To install Kyma with Knative when deploying on a GKE cluster from release, follow the instructions outlined in the **Install Kyma on a GKE cluster** installation guide.

To prepare the `my-kyma.yaml` configuration file that installs Kyma with Knative on a GKE cluster, run:

```
cat kyma-config-cluster.yaml | sed -e "s/__DOMAIN__/$DOMAIN/g" | sed -e "s/__TLS_CERT
```

Knative with a GKE cluster deployment from sources

To install Kyma with Knative when deploying on a GKE cluster from sources, follow the instructions outlined in the **Install Kyma on a GKE cluster** installation guide.

To prepare the `my-kyma.yaml` configuration file that installs Kyma with Knative on a GKE cluster, run:

```
cat installation/resources/installer.yaml <(echo -e "
---") installation/resources/installer-config-cluster.yaml.tpl  <(echo -e "
---") installation/resources/installer-cr-cluster.yaml.tpl | sed -e "s/global.knative
```

Installation with custom Service Catalog deployment

You can use Kyma with a custom Service Catalog deployment. To enable such implementation, remove the Service Catalog from the list of components that install with Kyma.

Prerequisites

- The Service Catalog in the Kyma-supported version . To check the currently supported version of the Service Catalog, see the value of the **image** parameter in this file.

  > **NOTE:** Follow this guide to learn how to install and configure Service Catalog on a Kubernetes cluster.

- Kyma latest release.

Local installation

1. Remove these lines from the installer-cr.yaml.tpl file:

   ```
   name: "service-catalog"
   namespace: "kyma-system"
   ```

2. Follow the installation steps described in the **Install Kyma locally from the release** document.

Cluster installation

1. Remove these lines from the installer-cr-cluster.yaml.tpl file:

```
name: "service-catalog"
namespace: "kyma-system"
```

2. Follow the installation steps described in the **Install Kyma on a GKE cluster** document.

Verify the installation

1. Check if all Pods are running in the `kyma-system` Namespace:

```
kubectl get pods -n kyma-system
```

2. Sign in to the Kyma Console using the `admin@kyma.cx` login as described in the **Install Kyma locally from the release** document.

Sample service deployment on local

This Getting Started guide is intended for the developers who want to quickly learn how to deploy a sample service and test it with Kyma installed locally on Mac.

This guide uses a standalone sample service written in the Go language .

Prerequisites

To use the Kyma cluster and install the example, download these tools:

- kubectl 1.10.0
- curl

Steps

Deploy and expose a sample standalone service

Follow these steps:

1. Deploy the sample service to any of your Environments. Use the `stage` Environment for this guide:

```
kubectl create -n stage -f https://raw.githubusercontent.com/kyma-project/example
```

2. Create an unsecured API for your example service:

```
kubectl apply -n stage -f https://raw.githubusercontent.com/kyma-project/examples
```

3. Add the IP address of Minikube to the `hosts` file on your local machine for your APIs:

```
$ echo "$(minikube ip) http-db-service.kyma.local" | sudo tee -a /etc/hosts
```

4. Access the service using the following call:

```
curl -ik https://http-db-service.kyma.local/orders
```

The system returns a response similar to the following:

```
HTTP/2 200
content-type: application/json;charset=UTF-8
vary: Origin
date: Mon, 01 Jun 2018 00:00:00 GMT
content-length: 2
x-envoy-upstream-service-time: 131
server: envoy

[]
```

Update your service's API to secure it

Run the following command:

```
kubectl apply -n stage -f https://raw.githubusercontent.com/kyma-project/examples/
```

After you apply this update, you must include a valid bearer ID token in the Authorization header to access the service.

> **NOTE:** The update might take some time.

Sample service deployment on a cluster

This Getting Started guide is intended for the developers who want to quickly learn how to deploy a sample service and test it with the Kyma cluster.

This guide uses a standalone sample service written in the Go language.

Prerequisites

To use the Kyma cluster and install the example, download these tools:

- kubectl 1.10.0
- curl

Steps

Download configuration for kubectl

Follow these steps to download **kubeconfig** and configure kubectl to access the Kyma cluster:

1. Access the Console UI and download the **kubectl** file from the settings page.
2. Place downloaded file in the following location: `$HOME/.kube/kubeconfig` .
3. Point **kubectl** to the configuration file using the terminal: `export KUBECONFIG=$HOME/.kube/kubeconfig` .
4. Confirm **kubectl** is configured to use your cluster: `kubectl cluster-info` .

Set the cluster domain variable

The commands throughout this guide use URLs that require you to provide the domain of the cluster which you are using. To complete this configuration, set the variable `yourClusterDomain` to the domain of your cluster.

For example, if your cluster's domain is `demo.cluster.kyma.cx`, run the following command:

```
export yourClusterDomain='demo.cluster.kyma.cx'
```

Deploy and expose a sample standalone service

Follow these steps:

1. Deploy the sample service to any of your Environments. Use the `stage` Environment for this guide:

   ```
   kubectl create -n stage -f https://raw.githubusercontent.com/kyma-project/example
   ```

2. Create an unsecured API for your service:

   ```
   curl -k https://raw.githubusercontent.com/kyma-project/examples/master/gateway/se
   ```

3. Access the service using the following call:

   ```
   curl -ik https://http-db-service.$yourClusterDomain/orders
   ```

   The system returns a response similar to the following:

   ```
   HTTP/2 200
   content-type: application/json;charset=UTF-8
   vary: Origin
   date: Mon, 01 Jun 2018 00:00:00 GMT
   content-length: 2
   x-envoy-upstream-service-time: 131
   server: envoy

   []
   ```

Update your service's API to secure it

Run the following command:

```
curl -k https://raw.githubusercontent.com/kyma-project/examples/master/gateway/ser
```

After you apply this update, you must include a valid bearer ID token in the Authorization header to access the service.

> **NOTE:** The update might take some time.

Develop a service locally without using Docker

You can develop services in the local Kyma installation without extensive Docker knowledge or a need to build and publish a Docker image. The `minikube mount` feature allows you to mount a directory from your local disk into the local Kubernetes cluster.

This guide shows how to use this feature, using the service example implemented in Golang.

Prerequisites

Install Golang.

Steps

Install the example on your local machine

1. Install the example:

```
go get -insecure github.com/kyma-project/examples/http-db-service
```

2. Navigate to installed example and the `http-db-service` folder inside it:

```
cd ~/go/src/github.com/kyma-project/examples/http-db-service
```

3. Build the executable to run the application:

```
CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .
```

Mount the example directory into Minikube

For this step, you need a running local Kyma instance. Read the **Install Kyma locally from the release** Installation guide to learn how to install Kyma locally.

1. Open the terminal window. Do not close it until the development finishes.
2. Mount your local drive into Minikube:

```
# Use the following pattern:
minikube mount {LOCAL_DIR_PATH}:{CLUSTER_DIR_PATH}`
# To follow this guide, call:
minikube mount ~/go/src/github.com/kyma-project/examples/http-db-service:/go/src/
```

See the example and expected result:

```
# Terminal 1
$ minikube mount ~/go/src/github.com/kyma-project/examples/http-db-service:/go/src/gi

Mounting /Users/{USERNAME}/go/src/github.com/kyma-project/examples/http-db-service in
This daemon process must stay alive for the mount to still be accessible...
ufs starting
```

Run your local service inside Minikube

1. Create Pod that uses the base Golang image to run your executable located on your local machine:

```
# Terminal 2
kubectl run mydevpod --image=golang:1.9.2-alpine --restart=Never -n stage --overr
{
"spec":{
   "containers":[
      {
```

```
            "name":"mydevpod",
            "image":"golang:1.9.2-alpine",
            "command": ["./main"],
            "workingDir":"/go/src/github.com/kyma-project/examples/http-db-service",
            "volumeMounts":[
               {
                  "mountPath":"/go/src/github.com/kyma-project/examples/http-db-serv
                  "name":"local-disk-mount"
               }
            ]
         }
      ],
      "volumes":[
         {
            "name":"local-disk-mount",
            "hostPath":{
               "path":"/go/src/github.com/kyma-project/examples/http-db-service"
            }
         }
      ]
   }
   }
   '
```

2. Expose the Pod as a service from Minikube to verify it:

```
kubectl expose pod mydevpod --name=mypodservice --port=8017 --type=NodePort -n st
```

3. Check the Minikube IP address and Port, and use them to access your service.

```
# Get the IP address.
minikube ip
# See the example result: 192.168.64.44
# Check the Port.
kubectl get services -n stage
# See the example result: mypodservice  NodePort 10.104.164.115  <none>  8017:322
```

4. Call the service from your terminal.

```
curl {minikube ip}:{port}/orders -v
# See the example: curl http://192.168.64.44:32226/orders -v
# The command returns an empty array.
```

Modify the code locally and see the results immediately in Minikube

1. Edit the `main.go` file by adding a new `test` endpoint to the `startService` function

```
router.HandleFunc("/test", func (w http.ResponseWriter, r *http.Request) {
 w.Write([]byte("test"))
})
```

2. Build a new executable to run the application inside Minikube:

```
CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .
```

3. Replace the existing Pod with the new version:

```
kubectl get pod mydevpod -n stage -o yaml | kubectl replace --force -f -
```

4. Call the new `test` endpoint of the service from your terminal. The command returns the `Test` string:

```
curl http://192.168.64.44:32226/test -v
```

Publish a service Docker image and deploy it to Kyma

In the Getting Started guide for local development of a service, you can learn how to develop a service locally. You can immediately see all the changes made in the local Kyma installation based on Minikube, without building a Docker image and publishing it to a Docker registry, such as the Docker Hub.

Using the same example service, this guide explains how to build a Docker image for your service, publish it to the Docker registry, and deploy it to the local Kyma installation. The instructions base on Minikube, but you can also use the image that you create, and the Kubernetes resource definitions that you use on the Kyma cluster.

> **NOTE:** The deployment works both on local Kyma installation and on the Kyma cluster.

Steps

Build a Docker image

The `http-db-service` example used in this guide provides you with the `Dockerfile` necessary for building Docker images. Examine the `Dockerfile` to learn how it looks and how it uses the Docker Multistaging feature, but do not use it one-to-one for production. There might be custom `LABEL` attributes with values to override.

1. In your terminal, go to the `examples/http-db-service` directory. If you did not follow the **Sample service deployment on local** guide and you do not have this directory locally, get the `http-db-service` example from the `examples` repository.
2. Run the build with `./build.sh`.

> **NOTE:** Ensure that the new image builds and is available in your local Docker registry by calling `docker images`. Find an image called `example-http-db-service` and tagged as `latest`.

Register the image in the Docker Hub

This guide bases on Docker Hub. However, there are many other Docker registries available. You can use a private Docker registry, but it must be available in the Internet. For more details about using a private Docker registry, see the **How to deploy a Docker image from a private registry** document.

1. Open the Docker Hub webpage.
2. Provide all of the required details and sign up.

Sign in to the Docker Hub registry in the terminal

1. Call `docker login`.
2. Provide the username and password, and select the `ENTER` key.

Push the image to the Docker Hub

1. Tag the local image with a proper name required in the registry: `docker tag example-http-db-service {username}/example-http-db-service:0.0.1`.
2. Push the image to the registry: `docker push {username}/example-http-db-service:0.0.1`.

  ` shell #This is how it looks in the terminal

```
The push refers to repository [docker.io/{username}/example-http-db-service] 4302273b9e11: Pushed 5835bd463c0e: Pushed 0.0.1: digest: sha256:9ec28342806f50b92c9b42fa36d979c0454aafcdda6845b362e2efb9816d1439 size: 734
```

> **NOTE:** To verify if the image is successfully published, check if it is available online at the following address: `https://hub.docker.com/r/{username}/example-http-db-service/`

Deploy to Kyma

The `http-db-service` example contains sample Kubernetes resource definitions needed for the basic Kyma deployment. Find them in the `deployment` folder. Perform the following modifications to use your newly-published image in the local Kyma installation:

1. Go to the `deployment` directory.
2. Edit the `deployment.yaml` file. Change the **image** attribute to `{username}/example-http-db-service:0.0.1`.
3. Create the new resources in local Kyma using these commands: `kubectl create -f deployment.yaml -n stage && kubectl create -f ingress.yaml -n stage`.
4. Edit your `/etc/hosts` to add the new `http-db-service.kyma.local` host to the list of hosts associated with your `minikube ip`. Follow these steps:
   - Open a terminal window and run: `sudo vim /etc/hosts`
   - Select the **i** key to insert a new line at the top of the file.
   - Add this line: `{YOUR.MINIKUBE.IP} http-db-service.kyma.local`
   - Type `:wq` and select the **Enter** key to save the changes.
5. Run this command to check if you can access the service: `curl https://http-db-service.kyma.local/orders`. The response should return an empty array.

Helm overrides for Kyma installation

Kyma packages its components into Helm charts that the Installer uses. This document describes how to configure the Installer with override values for Helm charts.

Overview

The Installer is a Kubernetes Operator that uses Helm to install Kyma components. Helm provides an overrides feature to customize the installation of charts, such as to configure environment-specific values. When using Installer for Kyma installation, users can't interact with Helm directly. The installation is not an interactive process.

To customize the Kyma installation, the Installer exposes a generic mechanism to configure Helm overrides called **user-defined** overrides.

User-defined overrides

The Installer finds user-defined overrides by reading the ConfigMaps and Secrets deployed in the `kyma-installer` Namespace and marked with the `installer:overrides` Label.

The Installer constructs a single override by inspecting the ConfigMap or Secret entry key name. The key name should be a dot-separated sequence of strings corresponding to the structure of keys in the chart's `values.yaml` file or the entry in chart's template. See the examples below.

Installer merges all overrides recursively into a single YAML stream and passes it to Helm during the Kyma installation/upgrade operation.

Common vs component overides

The Installer looks for available overrides each time a component installation or update operation is due. Overrides for the component are composed from two sets: **common** overrides and **component-specific** overrides.

Kyma uses common overrides for the installation of all components. ConfigMaps and Secrets marked with the label `installer:overrides` , contain the definition. They require no additional label.

Kyma uses component-specific overrides only for the installation of specific components. ConfigMaps and Secrets marked with both `installer:overrides` and `component: <name>` Labels, where `<name>` is the component name, contain the definition. Component-specific overrides have precedence over Common ones in case of conflicting entries.

Overrides Examples

Top-level charts overrides

Overrides for top-level charts are straightforward. Just use the template value from the chart (without leading ".Values." prefix) as the entry key in the ConfigMap or Secret.

Example:

The Installer uses a `core` top-level chart that contains a template with the following value reference:

```
memory: {{ .Values.test.acceptance.ui.requests.memory }}
```

The chart's default value `test.acceptance.ui.requests.memory` in the `values.yaml` file resolves the template. The following fragment of `values.yaml` shows this definition:

```
test:
  acceptance:
    ui:
      requests:
        memory: "1Gi"
```

To override this value, for example to "2Gi", proceed as follows:

- Create a ConfigMap in the `kyma-installer` Namespace, labelled with: `installer:overrides` (or reuse an existing one).
- Add an entry `test.acceptance.ui.requests.memory: 2Gi` to the map.

Once the installation starts, the Installer generates overrides based on the map entries. The system uses the value of "2Gi" instead of the default "1Gi" from the chart `values.yaml` file.

For overrides that the system should keep in Secrets, just define a Secret object instead of a ConfigMap with the same key and a base64-encoded value. Be sure to label the Secret with `installer:overrides`.

Sub-chart overrides

Overrides for sub-charts follow the same convention as top-level charts. However, overrides require additional information about sub-chart location.

When a sub-chart contains the `values.yaml` file, the information about the chart location is not necessary because the chart and it's `values.yaml` file are on the same level in the directory hierarchy.

The situation is different when the Installer installs a chart with sub-charts. All template values for a sub-chart must be prefixed with a sub-chart "path" that is relative to the top-level "parent" chart.

This is not an Installer-specific requirement. The same considerations apply when you provide overrides manually using the `helm` command-line tool.

Here is an example. There's a `core` top-level chart, that the Installer installs. There's an `application-connector` sub-chart in `core` with another nested sub-chart: `connector-service` . In one of its templates there's a following fragment (shortened):

```
spec:
  containers:
  - name: {{ .Chart.Name }}
    args:
      - "/connectorservice"
      - '--appName={{ .Chart.Name }}'
      - "--domainName={{ .Values.global.domainName }}"
      - "--tokenExpirationMinutes={{ .Values.deployment.args.tokenExpirationMinutes }
```

The following fragment of the `values.yaml` file in `connector-service` chart defines the default value for `tokenExpirationMinutes` :

```
deployment:
  args:
    tokenExpirationMinutes: 60
```

To override this value, such as to change "60 to "90", do the following:

- Create a ConfigMap in the `kyma-installer` Namespace labeled with `installer:overrides` or reuse existing one.
- Add an entry `application-connector.connector-service.deployment.args.tokenExpirationMinutes: 90` to the map.

Notice that the user-provided override key now contains two parts:

- The chart "path" inside the top-level `core` chart: `application-connector.connector-service`
- The original template value reference from the chart without the .Values. prefix: `deployment.args.tokenExpirationMinutes` .

Once the installation starts, the Installer generates overrides based on the map entries. The system uses the value of "90" instead of the default value of "60" from the `values.yaml` chart file.

Global overrides

There are several important parameters usually shared across the charts. Helm convention to provide these requires the use of the `global` override key. For example, to define the `global.domain` override, just use "global.domain" as the name of the key in ConfigMap or Secret for the Installer.

Once the installation starts, the Installer merges all of the map entries and collects all of the global entries under the `global` top-level key to use for installation.

Values and types

Installer generally recognizes all override values as strings. It internally renders overrides to Helm as a YAML stream with only string values.

There is one exception to this rule with respect to handling booleans: The system converts "true" or "false" strings that it encounters to a corresponding boolean value (true/false).

Merging and conflicting entries

When the Installer encounters two overrides with the same key prefix, it tries to merge them. If both of them represent a map (they have nested sub-keys), their nested keys are recursively merged. If at least one of keys points to a final value, the Installer performs the merge in a non-deterministic order, so either one of the overrides is rendered in the final YAML data.

It is important to avoid overrides having the same keys for final values.

Example of non-conflicting merge:

Two overrides with a common key prefix ("a.b"):

```
"a.b.c": "first"
"a.b.d": "second"
```

The Installer yields correct output:

```
a:
  b:
    c: first
    d: second
```

Example of conflicting merge:

Two overrides with the same key ("a.b"):

```
"a.b": "first"
"a.b": "second"
```

The Installer yields either:

```
a:
  b: "first"
```

Or (due to non-deterministic merge order):

```
a:
  b: "second"
```

Installation

The `installations.installer.kyma-project.io` CustomResourceDefinition (CRD) is a detailed description of the kind of data and the format used to control the Kyma Installer, a

proprietary solution based on the Kubernetes operator principles. To get the up-to-date CRD and show the output in the `yaml` format, run this command:

```
kubectl get crd installations.installer.kyma-project.io -o yaml
```

Sample custom resource

This is a sample CR that controls the Kyma Installer. This example has the **action** label set to `install`, which means that it triggers the installation of Kyma. The **name** and **namespace** fields in the `components` array define which components you install and Namespaces in which you install them. This example shows that you install the `hmc-default` release of the `remote-environments` component in the `kyma-integration` Namespace.

> **NOTE:** See the `installer-cr.yaml.tpl` file in the `/installation/resources` directory for the complete list of Kyma components.

```
apiVersion: "installer.kyma-project.io/v1alpha1"
kind: Installation
metadata:
  name: kyma-installation
  labels:
    action: install
  finalizers:
    - finalizer.installer.kyma-project.io
spec:
  version: "1.0.0"
  url: "https://sample.url.com/kyma_release.tar.gz"
  components:
    - name: "cluster-essentials"
      namespace: "kyma-system"
    - name: "istio"
      namespace: "istio-system"
    - name: "prometheus-operator"
      namespace: "kyma-system"
    - name: "provision-bundles"
    - name: "dex"
      namespace: "kyma-system"
    - name: "core"
      namespace: "kyma-system"
```

Custom resource parameters

This table lists all the possible parameters of a given resource together with their descriptions:

Field	Mandatory	Description
metadata.name	**YES**	Specifies the name of the CR.
metadata.labels.action	**YES**	Defines the behavior of the Kyma Installer. Available options are `install` and `uninstall`.
metadata.finalizers	**NO**	Protects the CR from deletion. Read this Kubernetes document to learn more about finalizers.
spec.version	**NO**	When manually installing Kyma on a cluster, specify any valid SemVer notation string.
spec.url	**YES**	Specifies the location of the Kyma sources `tar.gz` package. For example, for the `master` branch of Kyma, the address is `https://github.com/kyma-project/kyma/archive/master.tar.gz`
spec.components	**YES**	Lists which components of Helm chart components to install or update.
spec.components.name	**YES**	Specifies the name of the component which is the same as the name of the component subdirectory in the `resources` directory.
spec.components.namespace	**YES**	Defines the Namespace in which you want the Installer to install, or update the component.
spec.components.release	**NO**	Provides the name of the Helm release. The default parameter is the component name.

Related resources and components

These components use this CR:

Component	Description
Installer	The CR triggers the Installer to install, update or delete of the specified components.

Kyma features and concepts in practice

The table contains a list of examples that demonstrate Kyma functionalities. You can run all of them locally or on a cluster. Examples are organized by a feature or concept they showcase. Each of them contains ready-to-use code snippets and the instructions in `README.md` documents.

Follow the links to examples' code and content sources, and try them on your own.

Example	Description	Technology
HTTP DB Service	Test the service that exposes an HTTP API to access a database on the cluster.	Go, MSSQL
Event Service Subscription	Test the example that demonstrates the `publish` and `consume` features of the Event Bus.	Go
Event Lambda Subscription	Create functions, trigger them on Events, and bind them to services.	Kubeless
Gateway	Expose APIs for functions or services.	Kubeless
Service Binding	Bind a Redis service to a lambda function.	Kubeless, Redis, NodeJS
Call SAP Commerce	Call SAP Commerce in the context of the end user.	Kubeless, NodeJS
Alert Rules	Configure alert rules in Kyma.	Prometheus

Example	Description	Technology
Custom Metrics in Kyma	Expose custom metrics in Kyma.	Go, Prometheus
Event Email Service	Send an automated email upon receiving an Event.	NodeJS
Tracing	Configure tracing for a service in Kyma.	Go