Kumori PaaS Quick Start

Kumori Systems v1.0.0, June 2018

Table of Contents

This document describes the basic elements of Kumori PaaS, and guides the developer ('you') through the implementation and deployment of two example Node.js applications.
For sure, this is **not** a complete manual or a full reference guide of Kumori PaaS. In fact, what we will show you here is only a small part of the full potential of the platform.

1. Introducing Kumori

Kumori is a Platform as a Service (PaaS) that eases the development and deployment of an application, and the management of its lifecycle.

Kumori runs on top of an IaaS, from where it obtains and manages resources in a transparent way to application developer, so you don’t have to worry about infrastructure.

Kumori mission is to support the creation of elastic applications and manage to automatically scale them based on load situation, fulfilling a SLA and minimizing the incurred cost in the underlying IAAS.

Access to Kumori PaaS is currently done via Kumori Dashboard, Kumori CLI, and the underlying Admission REST API. The two first ways are covered, although not thoroughtly, in this guide.

1.1. The Kumori PaaS Service Model

In order for a service to be managed by Kumori PaaS, it must follow a very specific service model. This model is defined by the following elements:

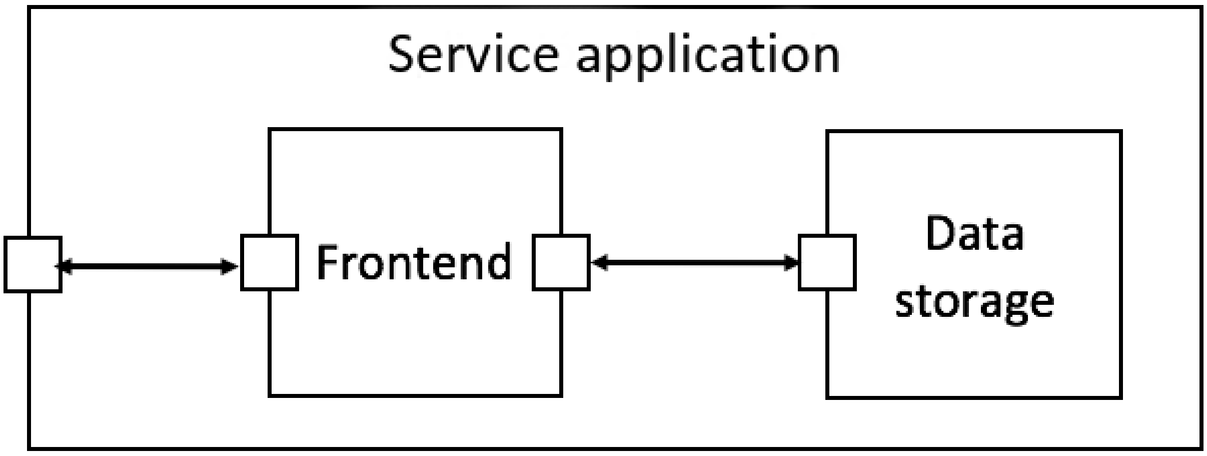
- Component: basic unit of execution. It is a runnable and self-contained piece of code that tipically implements a certain API. A component can interact with other components through channels, which are provided by the platform, but defined by component developer. A component can also have some configuration parameters and require some resources to work (e.g. volatile or persistent volumes).
- Service application: set of components that work together to provide a final service. Each component can carry out one or more roles in the service. A service application declares a specific topology connecting different role channels through defined connectors.
- Service: it is the result of deploying a service application. It is composed by a number of running instances of each role. The number of instances may vary during service lifecycle.

1.1.1. An example

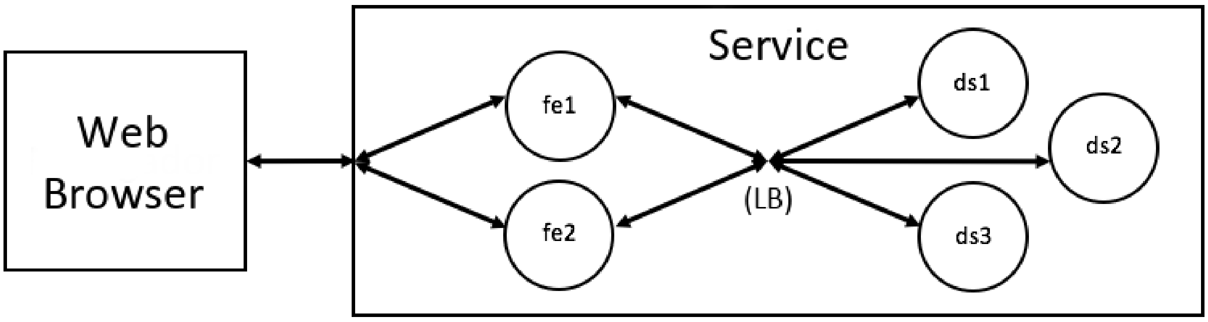
Let’s say we want to provide a key-value datastore as a service that can be accessed through a REST API (this example is covered in detail in further sections). Our service application will be composed by two roles carried out by two components:

- Data Storage: deals with data storing and retrieving. It requires a single channel through which it handles requests to store and retrieve data.
- Front End (FE): provides a REST API to interact with the datastore. This component requires

two channels. One to attend to REST petitions and the other to issue requests to the data storage role.



Once deployed, Kumori PaaS will launch several instances of each role.
In this example, instances of FE role will be connected by a channel to Data Storage role instances reciprocal channel, the routing element in the middle being a Load Balancer (LB) connector.



2. Setting up tools and environment

2.1. Environment requisites

- Linux OS (current LTS version of Ubuntu or equivalent)
- Docker CE
- Node.js v8
- Npm
- Git
- curl

2.2. Kumori CLI install

Kumori CLI (command-line interface) is a tool intended to boost the process of developing elements for Kumori PaaS. It is available in npm.

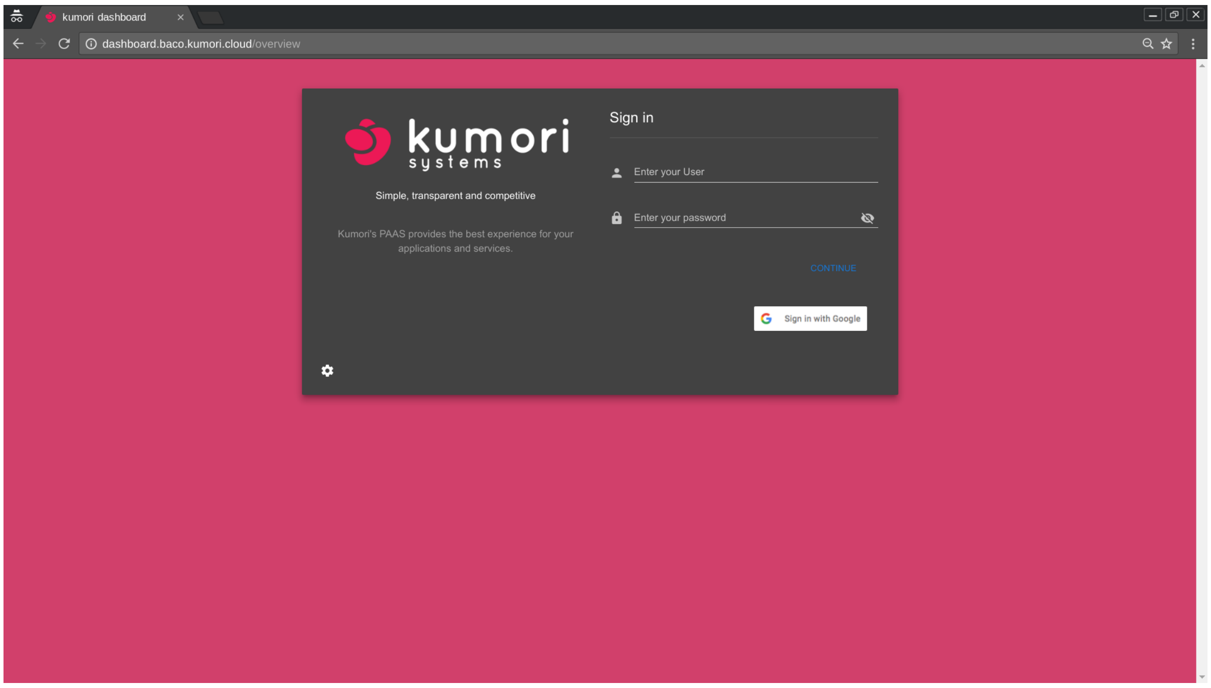
Run the following shell command to install Kumori CLI:

```
npm install -g @kumori/cli
```

2.3. Kumori Dashboard

Kumori Dashboard is available in https://dashboard.baco.kumori.cloud

Be sure you already have a Kumori account, because you will need to access Dashboard in further sections. Otherwise, sign up and get a free account here.


kumori dashboard
dashboard.baco.kumori.cloud/overview
kumori systems
Simple, transparent and competitive
Kumori's PAAS provides the best experience for your applications and services.
Sign in
Enter your User
Enter your password
CONTINUE
Sign in with Google

2.4. Create a workspace

Kumori CLI uses the concept of workspace, where all the elements related to a project are placed. To create a new workspace, switch to an empty directory and run `kumori init` command:

```
mkdir workspace
cd workspace
kumori init
## Initializing workspace following standard Kumori project
hierarchy...
## Initializing kumoriConfig.json for this workspace...
```

`kumoriConfig.json` file contains the configuration used in this workspace, including the API access token used to interact with the platform.

2.5. Set your workspace domain

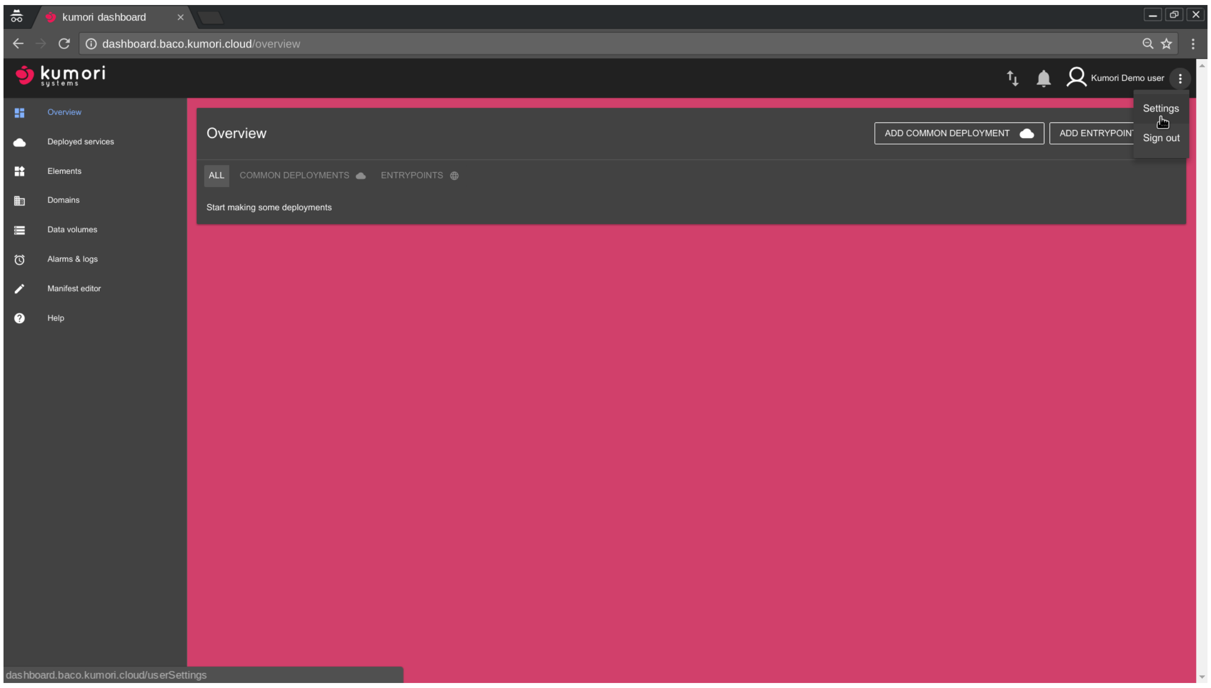
In Kumori PaaS, every element has a unique Uniform Resource Name (URN) that is used to identify it. This URN is defined when registering an element, and as part of that URN a domain name is used. In order to avoid collisions with other customers that would prevent you from registering parts of your project, you need to set a domain of your own choice, which is done at workspace level (although it could be overrided when adding particular elements):

```
kumori set domain YOUR_CHOSEN_DOMAIN
```

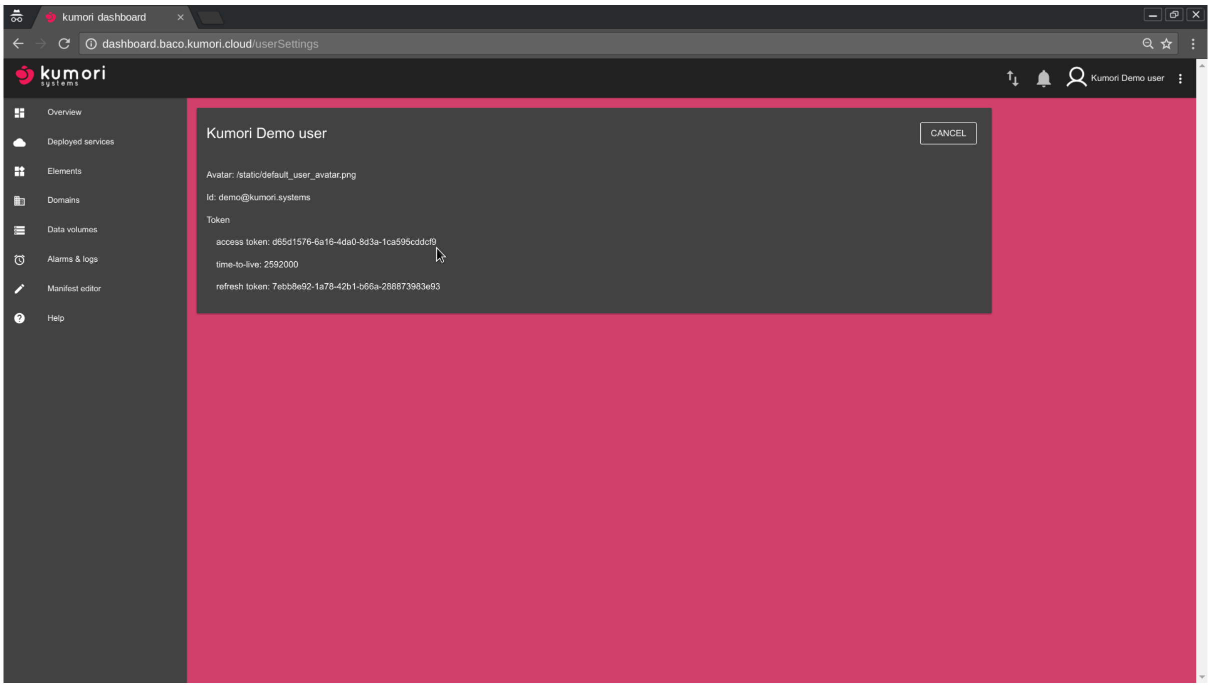
2.6. Obtain your API access token

Go to Kumori Dashboard and sign in using the same authorization system (Google, GitHub...) you used during sign up. This is important, as the different systems are not linked, even if two accounts share the same email.

Open the three-dot menu at the top right of the page, next to user name, and click on Settings.



The main view will change to the one below:



Kumori Demo user

Avatar: /static/default_user_avatar.png

Id: demo@kumori.systems

Token

access token: d65d1576-6a16-4da0-8d3a-1ca595cddcf9

time-to-live: 2592000

refresh token: 7ebb8e92-1a78-42b1-b66a-288873983e93

Copy the API access token.

Then, run the following Kumori CLI commands:

```
kumori stamp update -t YOUR_API_ACCESS_TOKEN baco
kumori stamp use baco
```

BACO is the current production version of Kumori PaaS.

Now, you will be able to interact with the platform from your workspace.

For security reasons, API access tokens expire after 30 days. In the future, you will be able to issue and revoke permament tokens in Dashboard, but at the moment, you will need to repeat this operation every month.

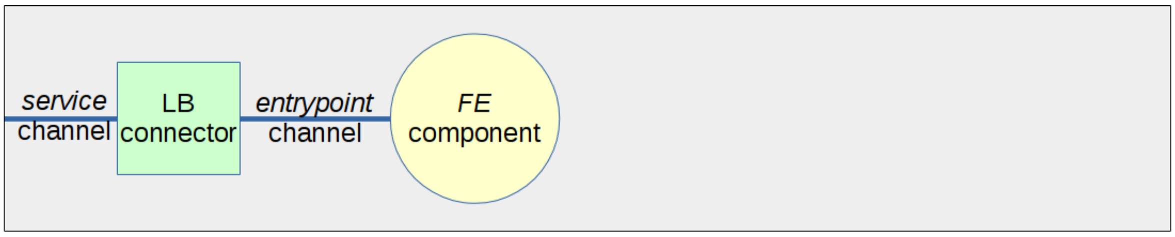
3. Get your first service up and running in Kumori PaaS (Hello World example)

We are going to start with a very simple service application, composed by a single component named FE (Front End), developed using Node.js and Express.

FE serves a static sample page in `/` (so you can quickly test the service in your browser) as well as exposing a simple REST API in `/api/`, with two routes: `/api/sayhello`, which always returns "Hello World!" message, and `/api/echo/:msg`, which returns the message passed as part of the URL.

Not surprisingly, the topology of this service lacks channels and connectors for interconnecting components.
The component is going to have a single channel that allows to access the service from the

outside, through a domain name.



3.1. Add FE component to workspace

First, we need to add a new component to our workspace using Kumori CLI. For this first example, we are going to use the `hello-world` template, that provides a fully implemented component, and we are going to name the component as `hello_world` (note the `_` separator difference, the use of `-` is forbidden for component names):

```
kumori component add -t hello-world hello_world
```

This will create the component under `components/YOUR_CHOSEN_DOMAIN/hello_world/` workspace path.

3.1.1. Implementation

The source code of this example is extensively commented and we encourage you to review it for implementation details.

A component is nothing more than a class that inherits from Component (more on this later), and that implements the necessary methods for the Kumori runtime (in this example, the one corresponding to Node.js) to manage its lifecycle.

In the case of this example we have:

package.json

The usual in a project for Node.js. It is worth noting that it includes dependencies to two Kumori modules:

- component: class that our component must extend.
- http-message: equivalent to the Node.js http module, but that listens on a Kumori channel instead of IP+port.
  Apart from that, it includes some custom NPM scripts.

taskfile.js

Contains Taskr configuration and targets. Taskr is a task automation tool, much like Gulp or Grunt, but with some nice features. In our templates, we provide targets for installing dependencies, building and generating distributable versions of the components, the ones that Kumori CLI uses under the hood.

Manifest.json

It is the descriptor of the component. We will explain its content later on. For the moment, just mentioning that it defines component channels and parameters.

src/restapi.coffee

REST API implementation. It uses Express, with only a single thing that is exclusively related to Kumori PaaS: instead of using Node.js http module, it employs Kumori httpMessage module.

src/index.coffee

FE component implementation. Its methods *constructor*, *run*, *shutdown* y *configure* are invoked by Kumori PaaS, managing instance lifecycle. In its *constructor* the RestAPI object is created, which is started in *run* method.

static/

Contains the files used for serving the static sample page, i.e. a bunch of HTML, JS, image and font files.

3.2. Build FE component

Once developed the component, we must install its dependencies, before registering it on the platform.

We cannot simply run `npm install` on our computer, since we may have some dependency on the operating system libraries (e.g when compiling some module, using `node-gyp` in this case) that may cause troubles later.

Therefore, we must perform the installation of dependencies using Kumori CLI, which internally runs some scripts and targets defined in `package.json` and `taskfile.js`, respectively, that essentially execute `npm install` in the same runtime environment (i.e. same Docker image) that the one that will run the instance of the component. The specific runtime is defined in component manifest, but more on that later.

To build the component, run the following Kumori CLI command:

```
kumori component build hello_world
```

3.3. Register FE component

Now it's time to register our previosly built component in Kumori PaaS.

To do so, run the following Kumori CLI command:

```
kumori component register hello_world
```

This requires that you have configured a valid API access token. If that's not the case, please revisit preceding sections.

3.4. Add Hello World service aplication to workspace

At this point, all the components of the service application are ready to use, so we can define a service application that makes use of them, assign them roles, define connectors, and service channels, and declare a specific topology connecting role channels with other role channels or service channels through defined connectors. To do so, we need to add a new service application to our workspace using Kumori CLI.

For this first example, we are going to use the `hello-world` template, that provides a fully defined service application, and (due to limitations in template system) we need to name it with the same name as the component: `hello_world`:

```
kumori service add -t hello-world hello_world
```

This will create the service application under `services/YOUR_CHOSEN_DOMAIN/hello_world/` workspace path.

The `Manifest.json` file, that we will explain later on, contains all service application definition.

3.5. Register Hello World service application

Once defined, a service application only needs to be registered.

To register it in Kumori PaaS, run the following Kumori CLI command:

```
kumori service register hello_world
```

3.6. Add Hello World service application deployment to workspace

We are one step away from deploying our first service. What lasts is to add a new deployment of the previous service application to our workspace using Kumori CLI. Among other things, a deployment sets initial values for parameters and resource allocation.

We are going to name this first deployment (e.g.) as `hello-world-service`:

```
kumori deployment add hello-world-service hello_world
```

This will create the service application deployment under `deployments/YOUR_CHOSEN_DOMAIN/hello-world-service/` workspace path.

The `Manifest.json` file, that we will explain later on, contains all service application deployment definition. The default values for resource allocation are good enough for this example.

3.6.1. Configure logging system (set `logzioToken` parameter)

The following is not mandatory (our examples work), but we encourage you to do it, especially if you introduce modifications in the code.

In this example, we have used a logging system in the component code. This is especially useful when running things on a managed environment, i.e. when running your service applications in Kumori PaaS. There are many solutions available, we have just chosen the simplest one to our knowledge.

Go to https://logz.io/freetrial/ and create an account.

You will be immediately logged in, you don't even need to click on any confirmation email.

Then, go to https://app.logz.io/#/dashboard/settings/general and copy account token.

Finally, edit `deployments/YOUR_CHOSEN_DOMAIN/hello-world-service/Manifest.json` and set `logzioToken` value.

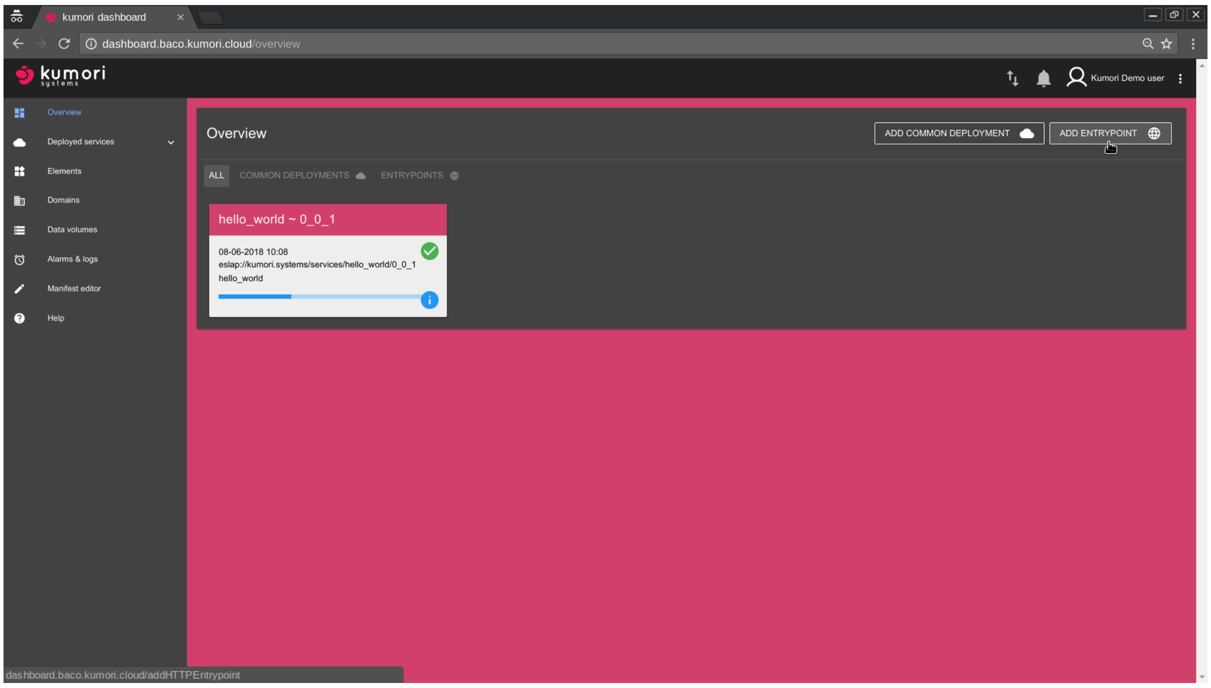
You will be able to check logs in https://app.logz.io/#/dashboard/kibana once you have deployed the service.

3.7. Deploy Hello World service

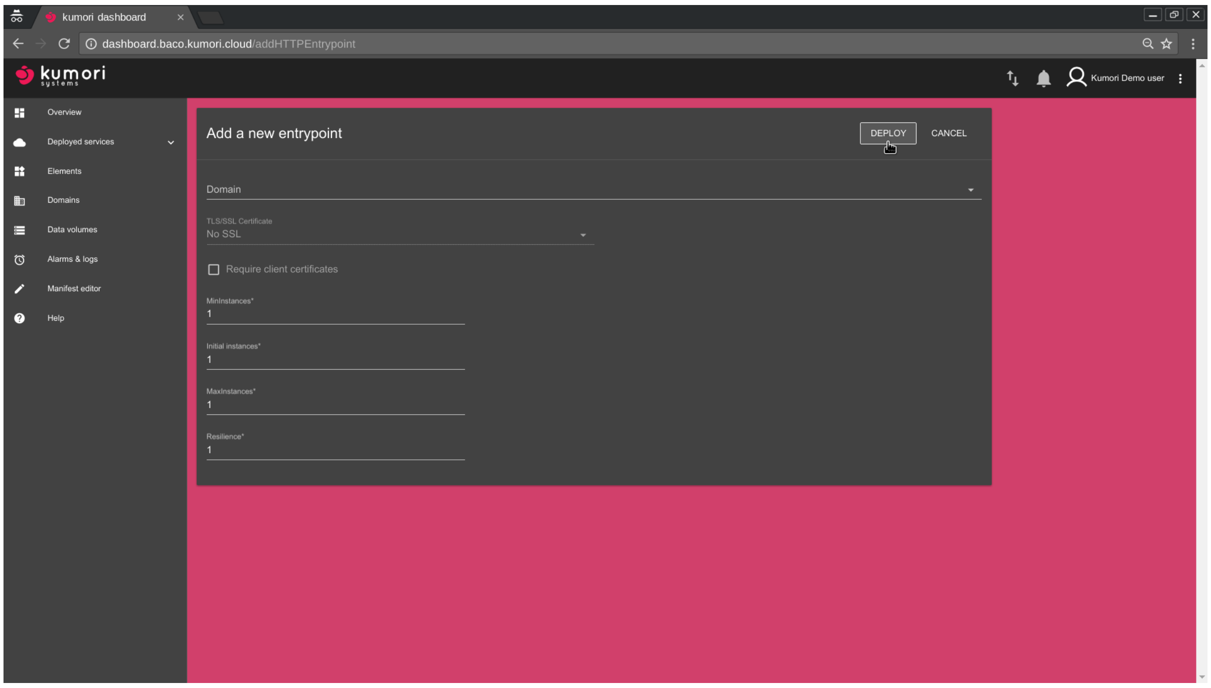
It's time to deploy! Once again, you can do it using Kumori CLI:

```
kumori deployment deploy hello-world-service
```

At this point, you have deployed your service, but it is not accesible from outside. Go to Kumori Dashboard - Overview, and press Add Entrypoint button.



Do not select any domain, so a random domain is generated, and press Deploy button.



dashboard.baco.kumori.cloud/addHTTPEntrypoint

Add a new entrypoint

Domain

TLS/SSL Certificate
No SSL

Require client certificates

MinInstances*
1

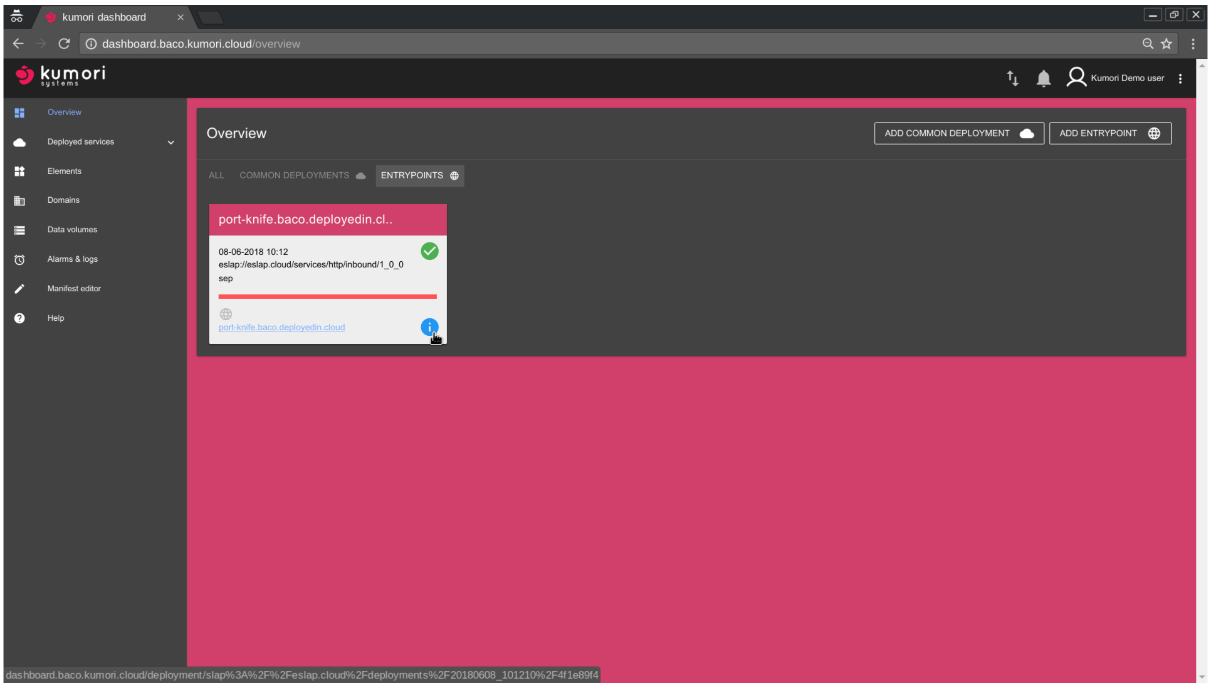
Initial instances*
1

MaxInstances*
1

Resilience*
1

This deploys an HTTP inbound service.

Then, click on the blue Info button of HTTP inbound service.



dashboard.baco.kumori.cloud/overview

Overview

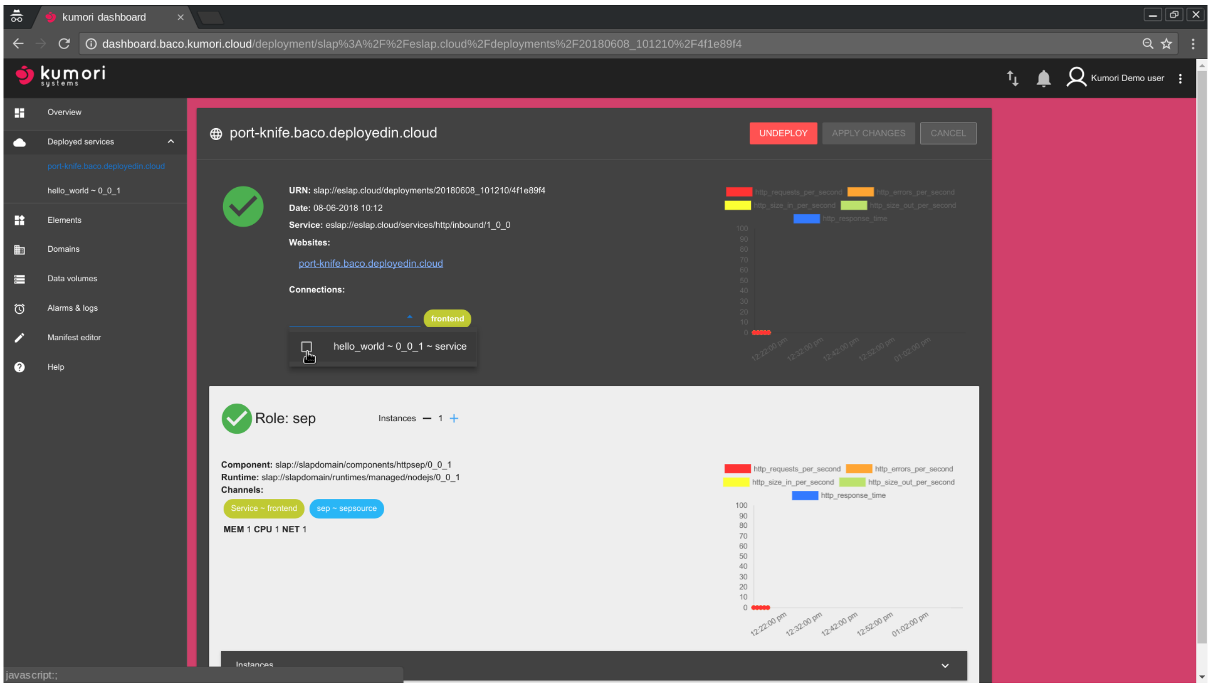
port-knife.baco.deployedin.cl..

08-06-2018 10:12
eslap://eslap.cloud/services/http/inbound/1_0_0
sep

port-knife.baco.deployedin.cloud

dashboard.baco.kumori.cloud/deployment/slap%3A%2F%2Feslap.cloud%2Fdeployments%2F20180608_101210%2F4f1e89f4

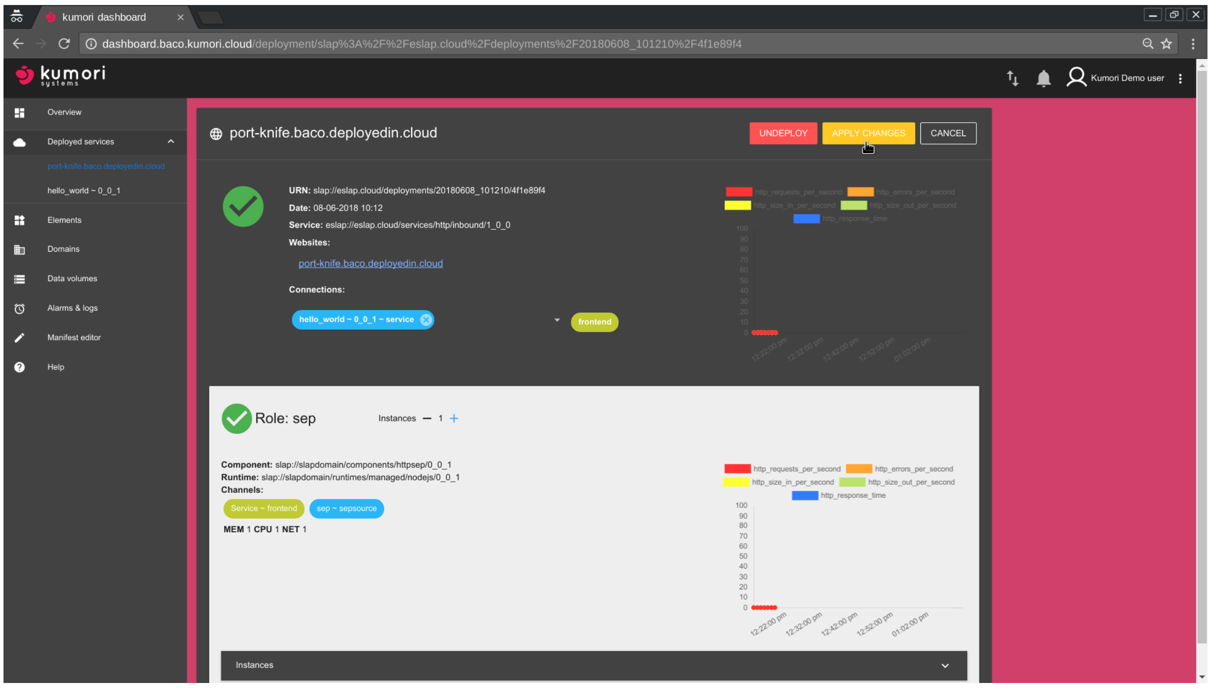
The main view will change to show you service deployment details, such as the random domain that has been assigned to it. Under Connections, on the dropdown menu next to `frontend` (that's a service channel), select `hello_world ~ 0_0_1 ~ service`.



URN: slap://eslap.cloud/deployments/20180608_101210/4f1e89f4
Date: 08-06-2018 10:12
Service: eslap://eslap.cloud/services/http/inbound/1_0_0
Websites:
port-knife.baco.deployedin.cloud
Connections:

Component: slap://slapdomain/components/httpsep/0_0_1
Runtime: slap://slapdomain/runtimes/managed/nodejs/0_0_1
Channels:
MEM 1 **CPU** 1 **NET** 1

Press Apply Changes button.



URN: slap://eslap.cloud/deployments/20180608_101210/4f1e89f4
Date: 08-06-2018 10:12
Service: eslap://eslap.cloud/services/http/inbound/1_0_0
Websites:
port-knife.baco.deployedin.cloud
Connections:

Component: slap://slapdomain/components/httpsep/0_0_1
Runtime: slap://slapdomain/runtimes/managed/nodejs/0_0_1
Channels:
MEM 1 **CPU** 1 **NET** 1

This links the HTTP inbound service with the Hello World service through their service channels.

Finally, go the provided website URL and have fun.

If you have been extremely quick doing the above steps and see an error message telling you that the requested service is not deployed, just wait a bit and press Try Again button.

Apart from the static page available in `/`, check `/api/sayhello` and `/api/echo/:msg` routes from your browser or command-line `curl`.

Hooray! You have deployed your first service in Kumori PaaS. Now, let’s see what we have done and give some insights. A second example will come afterwards.

4. The Kumori PaaS Service Model in detail

As we have briefly seen, manifests are used for the definition of elements and its registration in Kumori PaaS. A manifest declares the type of element (component, service application, service deployment) and its characteristics (name, configuration parameters, channels, connectors, etc.).

A more detailed description of the elements of the Kumori PaaS Service Model and their associated manifests is provided hereunder.

4.1. Component

A component is a standalone piece of software that can be scaled independently. Although a component can be arbitrarily complex, it is recommended that it encapsulates simple and specific functionality, following the criteria of microservice-based architectures.

Components communicate with each other via **channels** and must implement the `Component` interface (described in examples code and later in this guide) in order to be managed by Kumori PaaS. There are implementations of this interface for Node.js and Java.

Registering a component in Kumori PaaS requires of:

- The component binaries (or the code in case of interpreted languages) and their dependencies (`node_modules` for Node.js components).
- A component manifest declaring its characteristics, including:
  - Name of the component. The name is a URN that must follow the format `eslap://YOUR_CHOSEN_DOMAIN/component/COMPONENT_NAME/VERSION`. E.g. `eslap://kumori.systems/component/hello_world/1_0_0.`
  - Name of the component runtime. It must be one of the runtimes registered in the platform. E.g. `eslap://eslap.cloud/runtime/native/2_0_0`.
  - Channels needed to provide its functionality to other components.
  - Channels that requires to make use of other components functionality.
  - Name and type of the configuration parameters.
  - Name and type of resources it needs (e.g. volatile or persistent volumes).

Channels are objects provided by Kumori PaaS to component instances based on what is stated in their manifest. The semantics of the channels depend on their type:

- Send: allows sending messages and assigning them a topic (`sendChannel.send(message)`).
- Receive: allows receiving messages and subscribe to topics (`receiveChannel.on('message', (message) ▯ {…})`).
- Request: allows issuing a request that expects an asynchronous response using promises (`requestChannel.sendRequest(message).then((response) ▯ {…})`).
- Reply: allows handling requests and replying them asynchronously

(`replyChannel.handleRequest = (request) ▯ {…; return promise;}`).

- Duplex: allows to send messages specifying the recipient and receive messages sent by others. It is like a combination of *Send* and *Receive* channels except that the sender chooses the recipient and it is not possible to use topics.

4.1.1. Component interface

As we said, components must implement the `Component` interface, which is provided as a class that component code should extend. The `Component` interface is decribed below.

```
class Component
    constructor: (
        @runtime,        ①
        @role,           ②
        @iid,            ③
        @incnum,         ④
        @localData,      ⑤
        @resources,      ⑥
        @parameters,     ⑦
        @dependencies,   ⑧
        @offerings       ⑨
    ) ->

    run: () ->           ⑩

    shutdown: () ->      ⑪

    reconfig: (parameters) -> ⑫

module.exports = Component
```

① Object providing toolkit API of the runtime agent that runs the component

② Name of the role carried out by the component.

③ Identifier (string) assigned to the instance of the component. Typically there will be several instances of the component at a given time (variable over time depending on the load and performance).

④ Incarnation number. If an instance "dies" unexpectedly (e.g. due to a bug), the platform will restart it and increase this value.

⑤ Path where the instance can store data. It is volatile, and data persistence is not ensured on instance restart or relocation. All instances have this resource by default.

⑥ Dictionary of assigned Kumori resources to the instance. For example, a persistent volume.

⑦ Dictionary of instance configuration parameters. When deploying a service, value is given to the different parameters of each component, that particular instances can retrieve through this argument.

⑧ Dictionary of the channels required by the component, through which it can issue requests to other roles or services. Keys are channel names, values are channel objects.

(9) Dictionary of the channels offered by the component, through which it can answer requests from other roles or services.

(10) Method invoked by Kumori PaaS to start instance execution.

(11) Method invoked by Kumori PaaS to warn instance about its inminent shutdown. Instance should take necessary actions in this situation, persisting its state if needed. If the instance doesn't gracefully shutdown, it will be killed.

(12) Method invoked by Kumori PaaS to modify instance configuration.

4.2. Service application

A service application is a set of components interconnected to provide a certain functionality. Each component carries out a **role** and its channels are paired using **connectors**.

A service application also declares its own channels that can be used to link it to other services. A service channel must necessarily be paired with a channel of one of the roles that compose the service application.

A service application is defined in a manifest, the **service manifest**, in which it is declared:

- Name of the service application. The name is a URN that must follow the format `eslap://YOUR_CHOSEN_DOMAIN/service/COMPONENT_NAME/VERSION`. E.g. `eslap://kumori.systems/service/hello_world/1_0_0`.
- The roles that compose the service and the components that will carry them out.
- The service parameters, their type and how their values will be propagated to role parameters.
- The resources required for the service, their type and how they are distributed among the roles.
- The connectors that will pair the role channels. There are three possible types of connectors:
  - Publish/Subscribe: allows to pair *Send* and *Receive* channels.
  - Load Balancer: allows to pair *Request* and *Reply* channels.
  - Full: allows to pair *Duplex* channels.

To register a service application in Kumori PaaS, the components it uses must be previously registered.

4.3. Service

A service is the result of deploying a service application with a certain configuration. The deployment process involves creating instances of the component of each role and configuring them appropriately. Kumori PaaS can host multiple services of a single service application at the same time, each with its own configuration and component instances. The execution of deployment process starts with a **deployment manifest** that must contain, among other things:

- The name of the service application to be deployed.

- Values for the configuration parameters declared in the service application manifest.
- The resource elements (e.g. volumes) to be assigned to the service.
- The initial system resource allocation (e.g. CPU and RAM units) per role.

Once deployed, the number of instances assigned to each role in a service will vary over time depending on the fluctuation of the load that each of them handles. When an instance sends a message through one of its channels, it will reach one or more instances of the paired channels, depending on the type of connector used:

- Publish/Subscribe: reaches all target instances. If the message sent has a topic, only instances that subscribe to that topic in the paired channel will receive the message.
- Load Balancer: the request will only be handled by one of the instances.
- Full: the recipient is determined by the instance sending the message.

Since the components are standalone pieces of software, they can be scaled independently. Because Kumori PaaS knows the topology of services and manages communication channels, it is able to understand the interdependencies between roles in a service and take them into account to anticipate possible variations in the environment (e.g. load).

4.4. Manifest versioning

Once an element has been registered in Kumori PaaS (e.g. the component `eslap://kumori.systems/component/hello_world/0_0_1`), it cannot be modified.

Any modification requires registering again the element with an incremented version in its manifest (e.g. `eslap://kumori.systems/component/hello_world/0_0_2`).

If the element is referenced in another manifest, you will need to update the later (and increment its version) as well.

5. Hello World example manifests in detail

There are three manifests in Hello World example: one for the FE component, another for the service application, and a final one for the specific deployment of the service application that we have get into Kumori PaaS.

5.1. FE component

```
{
  "spec": "http://eslap.cloud/manifest/component/1_0_0",
  "name": "eslap://kumori.systems/components/hello_world/0_0_1", ①
  "runtime": "eslap://eslap.cloud/runtime/native/2_0_0", ②
  "code": "hello_world-code-blob", ③
  "configuration": {
    "resources": [ ], ④
    "parameters": [{ ⑤
      "name": "logzioToken",
      "type": "eslap://eslap.cloud/parameter/string/1_0_0"
    }]
  },
  "channels": { ⑥
    "requires": [],
    "provides": [{
      "name": "entrypoint",
      "type": "eslap://eslap.cloud/channel/reply/1_0_0",
      "protocol": "eslap://eslap.cloud/protocol/message/http/1_0_0"
    }]
  },
  "profile": {
    "threadability": "*"
  }
}
```

① Name of the component.

② Name of the component runtime. In this case, the corresponding to Node.js (native runtime of Kumori PaaS).

③ Name of the *bundle* that contains all the stuff belonging to this component.

④ Resources it needs (e.g. volatile or persistent volumes). Not used in this example.

⑤ Component parameters. In the component instantiation process (Component class constructor), it will be given a value for the `logzioToken` parameter, of *string* type. These values are set in the deployment manifest, and are propagated to all instances of this component.

⑥ Component channels. In this case, it provides a single *Reply* channel named `entrypoint`. This channel is not used for communicating to other components (since there are none), and is linked (service manifest) to the service channel.

Currently the *protocol* field is not checked, so you could just put the value `TBD`.

5.2. Service application

```
{
  "spec": "http://eslap.cloud/manifest/service/1_0_0",
  "name": "eslap://kumori.systems/services/hello_world/0_0_1", ①
  "configuration": {
    "resources": [],
    "parameters": [{ ②
      "name": "hello_world-fe",
      "type": "eslap://eslap.cloud/parameter/json/1_0_0"
    }]
  },
  "roles": [{ ③
    "name": "hello_world-fe",
    "component": "eslap://kumori.systems/components/hello_world/0_0_1"
  }],
  "channels": { ④
    "provides": [{
      "name": "service",
      "type": "eslap://eslap.cloud/channel/reply/1_0_0",
      "protocol": "eslap://eslap.cloud/protocol/message/http/1_0_0"
    }],
    "requires": []
  },
  "connectors": [{ ⑤
    "type": "eslap://eslap.cloud/connector/loadbalancer/1_0_0",
    "depended": [{
      "endpoint": "service"
    }],
    "provided": [{
      "role": "hello_world-fe",
      "endpoint": "entrypoint"
    }]
  }]
}
```

① Name of the service application.

② Spreading of the parameter values from the ones set in the deployment manifest to the parameters declared in the component.

③ List of *roles* that compose the service application. A component can carry out one or more roles in the service, and instances will be created for each of them. In this example, the component `eslap://kumori.systems/components/hello_world/0_0_1` is assigned a single *role* named "hello_world-fe".

④ Service channels. In this case it provides a single *Reply* channel named `service`, which allows access from the outside to the REST API that provides the service.

⑤ Connectors. We use a *LB* connector to link the `service` service channel to the `entrypoint` channel of the FE component. *LB* connectors pair *Request* channels with *Reply* channels, with one exception: when they connect a service *Reply* channel to a component *Reply* channel (in this case, *LB* performs *forward* functions).

5.3. Service application deployment

```
{
  "spec": "http://eslap.cloud/manifest/deployment/1_0_0",
  "servicename": "eslap://kumori.systems/services/hello_world/0_0_1",
(1)
  "name": "hello-world-service", (2)
  "configuration": {
    "resources": {},
    "parameters": { (3)
      "hello_world-fe": {
        "logzioToken":"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
      }
    }
  },
  "roles": {
    "hello_world-fe": {
      "resources": { (4)
        "__instances": 1,
        "__maxinstances": 3,
        "__cpu": 1,
        "__memory": 1,
        "__ioperf": 1,
        "__iopsintensive": false,
        "__bandwidth": 10,
        "__resilience": 1
      }
    }
  }
}
```

(1) URN of the service application to be deployed.

(2) Short name for the service. You will see this name in Kumori Dashboard, for example.

(3) Initial values for the component parameters. The various possibilities of the spreading mechanism from the values set in the deployment to the parameters declared in the components are beyond the scope of this guide.
In this case, we use a very simple mechanism: in the deployment manifest we have a *json* for each of the components, the values of this *json* are mapped one-to-one with the role parameters.

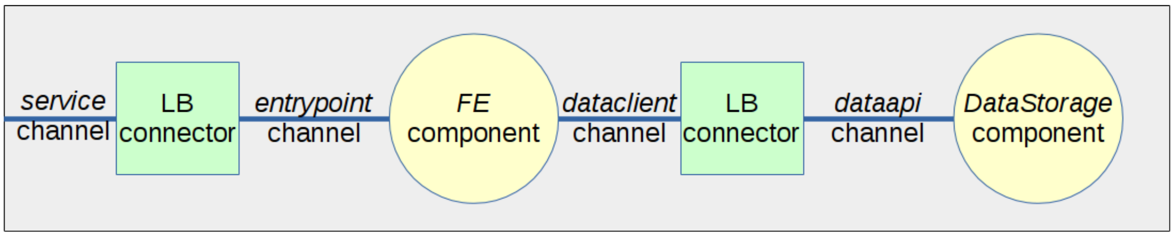
(4) For each role, we set the initial number of instances, the maximum number of instances (Kumori PaaS won't scale the role beyond this amount), resource allocation units, and resilience.

6. Hello World v2: FE + DataStorage (with Load Balancing)

The second example is a service application that offers a REST API that allows saving and retrieving key-value pairs.

As in the previous example, we have a FE component that implements the REST API using Express. The storage and retrieval of information falls upon a second component: DataStorage (a simple dictionary in memory, without any kind of persistence).

The communication between both components is of Request-Reply type, so both are connected through an *LB* connector (FE using *Request* channel, DataStorage using *Reply* channel).



This is not a valid solution as DataStorage is a component with a state that is not being shared across all its instances. If there is more than one DataStorage instance during the life cycle of the service, then it may happen that a request to recover a key-value pair is handled by an instance that does not have that key stored.

6.1. Add FE component to workspace

For this second example, we are going to use the `hello-world-v2-fe` template, that provides the fully implemented FE component, and we are going to name the component as `hello_world_v2_fe` (due to limitations in template system, the name must end with `_fe`):

```
kumori component add -t hello-world-v2-fe hello_world_v2_fe
```

This will create the component under `components/YOUR_CHOSEN_DOMAIN/hello_world_v2_fe/` workspace path.

6.1.1. Implementation

The source code of this example is extensively commented and we encourage you to review it for implementation details.

The implementation of the FE component is very similar to that of the first example. The difference being that RestAPI delegates the processing of the request to the DataStorage component, to which it sends the request (and from which it expects the reply) via the `dataclient` *Request* channel.

The method `RestAPI.send() shows how to send a message through a _Request` channel and receive the reply asynchronously through a promise.

6.2. Add DataStorage component to workspace

This time, we are going to use the `hello-world-v2-datastorage` template, that provides the fully implemented DataStorage component, and we are going to name the component as `hello_world_v2_datastorage` (due to limitations in template system, the name must end with `_datastorage`):

```
kumori component add -t hello-world-v2-datastorage
hello_world_v2_datastorage
```

This will create the component under `components/YOUR_CHOSEN_DOMAIN/hello_world_v2_datastorage/` workspace path.

6.2.1. Implementation

The source code of this example is extensively commented and we encourage you to review it for implementation details.

DataStorage contains the dictionary with the key-value pairs, on which it performs read and write operations.
Through the `dataapi` *Reply* channel, it receives requests from the FE component.
The method `DataStorage.handleRequest() shows how to process messages received by a _Reply` channel.

6.3. Build components

Once developed the components, we must install their dependencies, before registering them on the platform.

To build the components, run the following Kumori CLI commands:

```
kumori component build hello_world_v2_fe
kumori component build hello_world_v2_datastorage
```

6.4. Add Hello World v2 service aplication to workspace

For this second example, we are going to use the `hello-world-v2` template, that provides the fully defined service application, and (due to limitations in template system) we need to name it with the same prefix as the components: `hello_world_v2`:

```
kumori service add -t hello-world-v2 hello_world_v2
```

This will create the service application under `services/YOUR_CHOSEN_DOMAIN/hello_world_v2/` workspace path.

6.5. Add Hello World v2 service application deployment to workspace

We are one step away from deploying the service. What lasts is to add a new deployment of the previous service application to our workspace using Kumori CLI. We are going to name it (e.g.) as `hello-world-v2-service`:

```
kumori deployment add hello-world-v2-service hello_world_v2
```

This will create the service application deployment under `deployments/YOUR_CHOSEN_DOMAIN/hello-world-v2-service/` workspace path.

Again, the default values for resource allocation are good enough for this example.

6.5.1. Configure logging system (set `logzioToken` parameter)

The following is not mandatory (our examples work), but we encourage you to do it, especially if you introduce modifications in the code.

As in the first example, set the `logzioToken` value in `deployments/YOUR_CHOSEN_DOMAIN/hello-world-v2-service/Manifest.json`.

You will be able to check logs in https://app.logz.io/#/dashboard/kibana once you have deployed the service.

6.6. Manifests

We have four manifests: one for each of the components (FE and DataStorage), another for the service application, and a final one for the service application deployment.

6.6.1. FE component

```
{
  "spec": "http://eslap.cloud/manifest/component/1_0_0",
  "name": "eslap://kumori.systems/components/hello_world_v2_fe/0_0_1",
  "runtime": "eslap://eslap.cloud/runtime/native/2_0_0",
  "code": "hello_world_v2_fe-code-blob",
  "configuration": {
    "resources": [ ],
    "parameters": [{
      "name": "logzioToken",
      "type": "eslap://eslap.cloud/parameter/string/1_0_0"
    }]
  },
  "channels": {
    "requires": [{ (1)
      "name": "dataclient",
      "type": "eslap://eslap.cloud/channel/request/1_0_0",
      "protocol": "TBD"
    }],
    "provides": [{
      "name": "entrypoint",
      "type": "eslap://eslap.cloud/channel/reply/1_0_0",
      "protocol": "eslap://eslap.cloud/protocol/message/http/1_0_0"
    }]
  },
  "profile": {
    "threadability": "*"
  }
}
```

(1) The FE component manifest is basically the same as in the first example, to which we have added a *Request* channel. It appears in the `channels/requires` section as it is a dependency of the FE component on another component.

6.6.2. DataStorage component

```
{
  "spec": "http://eslap.cloud/manifest/component/1_0_0",
  "name":
"eslap://kumori.systems/components/hello_world_v2_datastorage/0_0_1",
  "runtime": "eslap://eslap.cloud/runtime/native/2_0_0",
  "code": "hello_world_v2_datastorage-code-blob",
  "configuration": {
    "resources": [ ],
    "parameters": [{
      "name": "logzioToken",
      "type": "eslap://eslap.cloud/parameter/string/1_0_0"
    }]
  },
  "channels": {
    "provides": [{ (1)
      "name": "dataapi",
      "type": "eslap://eslap.cloud/channel/reply/1_0_0",
      "protocol": "TBD"
    }],
    "requires": []
  },
  "profile": {
    "threadability": "*"
  }
}
```

(1) The DataStorage component provides (`channels/provides`) a *Reply* channel.

6.6.3. Service application

```
{
  "spec": "http://eslap.cloud/manifest/service/1_0_0",
  "name": "eslap://kumori.systems/services/hello_world_v2/0_0_1",
  "configuration": {
    "resources": [],
    "parameters": [{
      "name": "hello_world_v2-fe",
      "type": "eslap://eslap.cloud/parameter/json/1_0_0"
    },
    {
      "name": "hello_world_v2-datastorage", (1)
      "type": "eslap://eslap.cloud/parameter/json/1_0_0"
    }]
  },
  "roles": [{ (2)
    "name": "hello_world_v2-fe",
    "component":
"eslap://kumori.systems/components/hello_world_v2_fe/0_0_1"
  },{
    "name": "hello_world_v2-datastorage",
    "component":
"eslap://kumori.systems/components/hello_world_v2_datastorage/0_0_1"
  }],
  "channels": { (3)
    "provides": [{
      "name": "service",
      "type": "eslap://eslap.cloud/channel/reply/1_0_0",
      "protocol": "eslap://eslap.cloud/protocol/message/http/1_0_0"
    }],
    "requires": []
  },
  "connectors": [{
    "type": "eslap://eslap.cloud/connector/loadbalancer/1_0_0",
    "depended": [{
      "endpoint": "service"
    }],
    "provided": [{
      "role": "hello_world_v2-fe",
      "endpoint": "entrypoint"
    }]
  },{
    "type": "eslap://eslap.cloud/connector/loadbalancer/1_0_0", (4)
    "depended": [{
      "role": "hello_world_v2-fe",
      "endpoint": "dataclient"
    }],
    "provided": [{
      "role": "hello_world_v2-datastorage",
      "endpoint": "dataapi"
    }]
  }]
}
```

(1) DataStorage role also needs to receive the `logzioToken` parameter.

(2) We have added a new role, associated with the DataStorage component.

(3) The service channels section has not changed: the service still has a single entrypoint.

(4) Includes a new *LB* connector to link the FE *Request* channel to the DataStorage *Reply* channel.

6.6.4. Service application deployment

```
{
  "spec": "http://eslap.cloud/manifest/deployment/1_0_0",
  "servicename":
"eslap://kumori.systems/services/hello_world_v2/0_0_1",
  "name": "hello-world-v2-service",
  "configuration": {
    "resources": {},
    "parameters": {
      "hello_world_v2-fe": {
        "logzioToken":"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
      },
      "hello_world_v2-datastorage": { ①
        "logzioToken":"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
      }
    }
  },
  "roles": {
    "hello_world_v2-fe": {
      "resources": {
        "__instances": 1,
        "__maxinstances": 3,
        "__cpu": 1,
        "__memory": 1,
        "__ioperf": 1,
        "__iopsintensive": false,
        "__bandwidth": 10,
        "__resilience": 1
      },
    "hello_world_v2-datastorage": { ②
      "resources": {
        "__instances": 1,
        "__maxinstances": 3,
        "__cpu": 1,
        "__memory": 1,
        "__ioperf": 1,
        "__iopsintensive": false,
        "__bandwidth": 10,
        "__resilience": 1
      }
    }
  }
}
```

① DataStorage role also needs to receive the `logzioToken` parameter.

② We have added a new section of initial resource allocation for the new DataStorage role.

6.7. Deploy Hello World v2 service

As you may have noticed, we have skipped the registration of the different elements. Those are not mandatory steps, as `kumori deployment deploy` registers the necessary elements (provided they are available in the workspaces) to deploy a service application.

It's time to deploy! Once again, you can do it using Kumori CLI:

```
kumori deployment deploy hello-world-v2-service
```

Take note of deployment URN.

At this point, you have deployed your service, but it is not accesible from outside. Go to Kumori Dashboard - Overview, and press Add Entrypoint button. Do not select any domain, so a random domain is generated, and press Deploy button. This deploys an HTTP inbound service.

Then, click on the blue Info button of HTTP inbound service. The main view will change to show you service deployment details, such as the random domain that has been assigned to it. Under Connections, on the dropdown menu next to `frontend` (that's a service channel), select `hello_world_v2 ~ 0_0_1 ~ service` and press Apply Changes button. This links the HTTP inbound service with the Hello World v2 service through their service channels.

6.8. Testing the service

We can test the deployed service application by adding a new key-value pair and then retrieving it:

```
curl -s http://random-
domain.baco.deployedin.cloud/write?key=mykey1&value=myvalue1

{
  "instance": "hello_world_v2-datastorage_2" // DataStorage instance
processing the request
}

curl -s http://random-domain.baco.deployedin.cloud/read?key=mykey1

{
  "value": "myvalue1",
  "instance": "hello_world_v2-datastorage_2" // DataStorage instance
processing the request
}
```

6.9. Components with state (scaling a role in a service)

DataStorage is, as we have seen, a component with a state.
In this example, we have not been careful to manage the case where there is more than one instance of this component. As previously stated, it may happen that a request to recover a key-

value pair is handled by an instance that does not have that key stored.

We can manually scale the DataStorage role in the service using Kumori CLI to prove it:

```
kumori deployment scale <DEPLOYMENT_URN> hello_world_v2-datastorage 2
```

We can see that the DataStorage component is too basic, as we are not managing a shared state among all instances:

```
curl -s http://random-
domain.baco.deployedin.cloud/write?key=mykey1&value=myvalue1

{
  "instance": "hello_world_v2-datastorage_2" // DataStorage instance
processing the request
}

curl -s http://random-domain.baco.deployedin.cloud/read?key=mykey1

{
  "instance": "hello_world_v2-datastorage_3",
  "value": "not found"
}

curl -s http://random-domain.baco.deployedin.cloud/read?key=mykey1

{
  "instance": "hello_world_v2-datastorage_2",
  "value": "myvalue1"
}
```