Kaze CLI

Manual

Josep Maria Bernabé Gisbert <jbgisber@iti.es>

Version 1.0, 2017-07-11

Table of Contents

Get ready

Dependencies

`kaze` has been tested with NodeJS 7.9.0 and NPM 4.2.0. However, it should work with any ES6 compliant version of NodeJS.

Kaze also assumes that TypeScript is installed on your computer.

```
$> npm install -g typescript
```

Finally, docker should be also installed in the system.

Getting and installing Kaze

Kaze CLI is currently available in a SaaSDK project git repositories.

```
$> git clone git@gitlab.iti.upv.es:saasdk/devtools.git
```

After clonining Kaze CLI, run `npm install` to install all necessary dependencies. Then you must run one of below npm commands:

- `npm run build` will transpile all ts source codes into js codes and make it ready to link, publish, etc.
- `npm run local-install` to perform a local installation for testing purpose. This is achieved by creating a symlink to our binary file

Or, alternatively:

```
$> tsc -p .
$> sudo npm link // This will install locally kaze and create a symlink
```

To test the installation, the following commands are also available:

- `npm test` runs our complete test suite
- `npm run coverage` obtains current test coverage on transpiled js files

Getting and installing ECloud runtimes

To execute the local stamp and install your component dependencies, some images should be installed in your local Docker. To do that, you should use the `runtime-tool` you can found in ECloud's `sdk` repository:

```
$> git clone git@gitlab.com:ECloud/sdk.git
$> cd sdk/tools
$> ./install-dependencies.sh
$> ./runtime-tool.sh install -n slap://eslap.cloud/local-stamp/1_0_8
$> ./runtime-tool.sh install -n slap://eslap.cloud/elk/1_0_0
$> ./runtime-tool.sh install -n slap://eslap.cloud/runtime/native/1_0_1
```

Quick start guide

Creating a new workspace

A workspace is a folder containing everything related to your component and services, including the components code and the manifests.

To enable a folder as a workspace you should use the `init` command:

```
$> npm init
```

This command creates a set of folder and files in the current folder:

```
.
├──── builts
├──── components
├──── dependencies
├──── deployments
├──── kazeConfig.json
├──── resources
├──── runtimes
├──── services
└──── tests
```

- `builts`: in this folder kaze will store all generated bundles.
- `components`: contains all your components code and manifests.
- `dependencies`: contains all the external dependencies related to your services.
- `deployments`: contains all your deployment manifests.
- `kazeConfig.json`: kaze configuration file for this workspace. Contains information like the available execution environments. **This file is updated by kaze and you are not suposed to modify it manually**
- `resources`: contain all your resources manifests (for example, domains manifests).
- `runtimes`: contains all the manifests of the runtimes used in your comonents.
- `services`: contains all your service manifests.

- `tests`: contains all your test manifests.

Setting up your execution environments

An execution environment or *stamp* is a platform were your services can be hosted. Once a workspace is initialized you can register as stamps as you want, including a local stamp environment for testing purposes. New environments are registered by using the `kaze stamp` subcommand:

```
$> kaze stamp add argo http://admission.argo.kumori.cloud
$> kaze stamp add local http://localhost:8090
$> kaze stamp switch argo
```

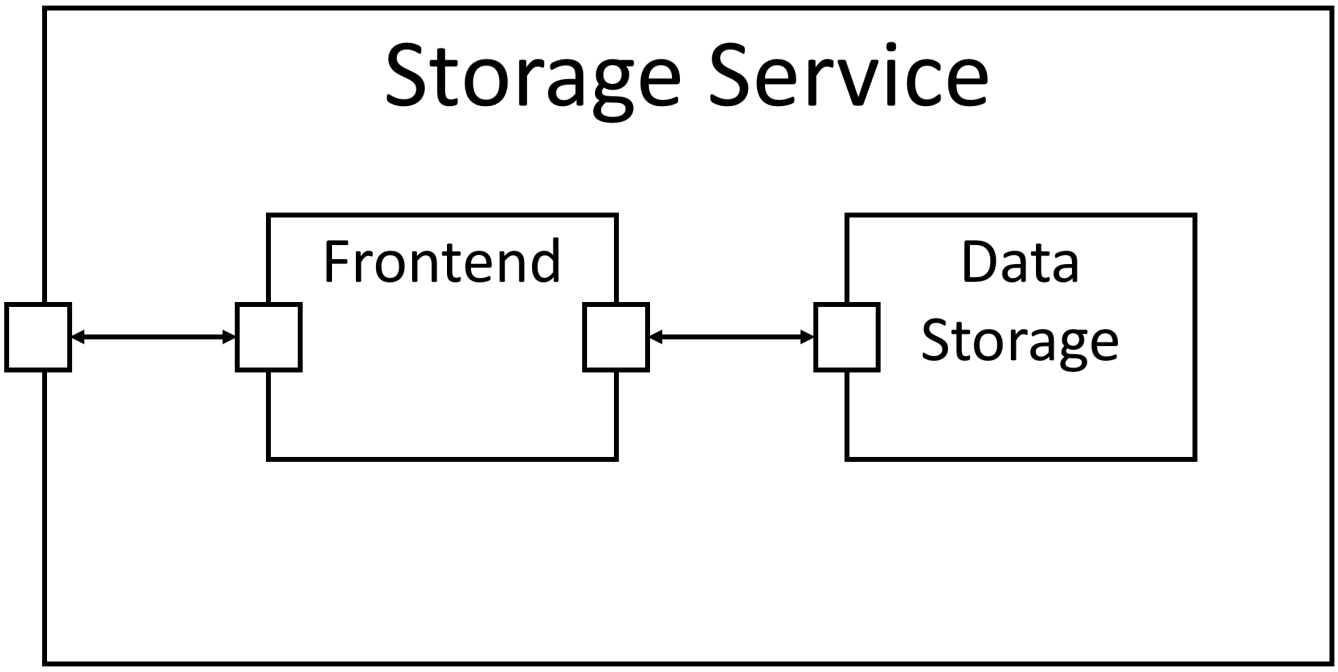
This registers two stamps. The first one is available in `http://admission.argo.kumori.cloud` and we will call it `argo`. The second one is a local stamp and we will call it `local`. Then, we set `argo` as the default stamp.

Populating the workspace with a sample service

For this example, we are going to use a sample storage serivce composed by two components:

- *Data storage*: it is in charge of storing and retrieving data.
- *Frontend*: it exposes the REST API.

The service exposes a single channel `service`. This channel is connected to the frontend, which connected at the same time to the data storage.



Once the component's code and all the manifests have been added to the workspace, it should look like this:

```
.
├── builts
├── components
│   └── quickstart
│       ├── datastorage
│       │   └── 0_0_1
│       │       ├── Manifest.json
│       │       └── code
│       │           ├── Manifest.json
│       │           └── src
│       │               ├── package.json
│       │               └── src
│       │                   └── index.coffee
│       └── fe
│           └── 0_0_1
│               ├── Manifest.json
│               └── code
│                   ├── Manifest.json
│                   └── src
│                       ├── package.json
│                       └── src
│                           ├── index.coffee
│                           └── restapi.coffee
├── dependencies
├── deployments
│   └── quickstart
│       └── sample
│           └── 0_0_1
│               └── Manifest.json
├── kazeConfig.json
├── resources
├── runtimes
├── services
│   └── quickstart
│       └── sample
│           └── 0_0_1
│               └── Manifest.json
└── tests
```

This is just an example. How to structure the code and the manifests in `components`, `services` and `deployments` folder is currently up to you.

Building and deploying elements

The *bundle registration* process os the way to provide and run your services in a stamp. A bundle is a zip file including components (manifest and code), service manifests, deployment manifests and/or any other ECloud element. Deployment manifests will be used to instantiate a new service with a specific configuration, as described in ECloud reference manual.

There are two ways to deploy a new service in a stamp by using kaze:

- Using the `kaze register` command. This command is used to register elements in the platform. If the path points to a zip file it is directly uploaded. If it points to a folder it is packed in a zip file and then uploaded.
- Using the `kaze deploy` command. This command can be used when only a deployment manifest is going to be uploaded. This time the path must point to a bundle containing the deployment manifest, to a folder containg the manifest or to a manifest itself.

ECloud requires that all components included in a bundle should have all dependencies already installed (for example, the `npm install` executed in case of node.js components or al the `jar` files in the `lib` folder for java ones) and, if necessary, all the code compiled (for example, all the `.class` files in the `class` folder for java components). Remember to install the dependencies using the component runtime image.

In the example, both components are based on Node.js.

```
$> docker run --rm -v
$PWD/components/quickstart/datastorage/0_0_1/code/src:/tmp/component --entrypoint=bash
-it eslap.cloud/runtime/native:1_0_1 -c 'cd /tmp/component && npm install'
$> docker run --rm -v $PWD/components/quickstart/fe/0_0_1/code/src:/tmp/component
--entrypoint=bash -it eslap.cloud/runtime/native:1_0_1 -c 'cd /tmp/component && npm
install'
```

For this example, we will use the first method:

```
$> kaze bundle components/quickstart/datastorage/0_0_1 components/quickstart/fe/0_0_1
services/quickstart/sample/0_0_1 deployments/quickstart/sample/0_0_1
Output zip = (1499939101655) test
Bundling
components/quickstart/datastorage/0_0_1,components/quickstart/fe/0_0_1,services/quicks
tart/sample/0_0_1,deployments/quickstart/sample/0_0_1 into builts/test.zip.
$> kaze register builts/test.zip
Target stamp: http://admission.argo.kumori.cloud
Paths [ 'builts/test.zip' ]
Registering builts/test.zip, it may take a while...
builts/test.zip registered
New deployment URN: slap://quickstart/deployments/20170713_095222/d7189989
New deployment entrypoint: sep_service: bank-frighten.argo.kumori.cloud
```

As a reply, `kaze` replies with the registration result, a list of error (if any) and the ID and entrypoints (URLs) of each deployment included in the bundle. In this case, a single auto-generated URL to access the sample service.

Getting information from a stamp

Kaze can retrieve information about the elements hosted in a registered stamp (currently, only

deployments):

```
$> kaze info -r deployments
```

The response is sent back in JSON format. For example, if we execute the previous command we get the following response:


```
{
  "success": true,
  "message": "SUCCESSFULLY PROCESSED DEPLOYMENT INFO.",
  "data": {
    "slap://quickstart/deployments/20170713_095222/d7189989": {
      "service": "eslap://quickstart/services/sample/0_0_1",
      "roles": {
        "fe": {
          "instances": {
            "quickstart_fe_262": {
              "id": "636c549c-048f-4af0-845a-001b19426903",
              "privateIp": "10.1.0.41",
              "publicIp": "147.135.132.239",
              "arrangement": {
                "__instances": 1,
                "__cpu": 1,
                "__memory": 1,
                "__ioperf": 1,
                "__iopsintensive": false,
                "__bandwidth": 1,
                "__resilience": 1,
                "mininstances": 1,
                "maxinstances": 1,
                "cpu": 1,
                "memory": 1,
                "bandwidth": 1,
                "failurezones": 1
              }
            }
          },
          "configuration": {
            "sampleParameter": "A value for sample parameter"
          }
        },
        "datastorage": {
          "instances": {
            "quickstart_datastorage_263": {
              "id": "636c549c-048f-4af0-845a-001b19426903",
              "privateIp": "10.1.0.41",
              "publicIp": "147.135.132.239",
              "arrangement": {
                "__instances": 4,
```

```
            "__cpu": 1,
            "__memory": 1,
            "__ioperf": 1,
            "__iopsintensive": false,
            "__bandwidth": 1,
            "__resilience": 1,
            "mininstances": 4,
            "maxinstances": 4,
            "cpu": 1,
            "memory": 1,
            "bandwidth": 1,
            "failurezones": 1
          }
        },
        "quickstart_datastorage_264": {
          "id": "92bb036d-db80-47e3-b909-3254e5fef767",
          "privateIp": "10.1.2.38",
          "publicIp": "217.182.140.238",
          "arrangement": {
            "__instances": 4,
            "__cpu": 1,
            "__memory": 1,
            "__ioperf": 1,
            "__iopsintensive": false,
            "__bandwidth": 1,
            "__resilience": 1,
            "mininstances": 4,
            "maxinstances": 4,
            "cpu": 1,
            "memory": 1,
            "bandwidth": 1,
            "failurezones": 1
          }
        },
        "quickstart_datastorage_265": {
          "id": "92bb036d-db80-47e3-b909-3254e5fef767",
          "privateIp": "10.1.2.38",
          "publicIp": "217.182.140.238",
          "arrangement": {
            "__instances": 4,
            "__cpu": 1,
            "__memory": 1,
            "__ioperf": 1,
            "__iopsintensive": false,
            "__bandwidth": 1,
            "__resilience": 1,
            "mininstances": 4,
            "maxinstances": 4,
            "cpu": 1,
            "memory": 1,
            "bandwidth": 1,
```

```
              "failurezones": 1
            }
          },
          "quickstart_datastorage_266": {
            "id": "92bb036d-db80-47e3-b909-3254e5fef767",
            "privateIp": "10.1.2.38",
            "publicIp": "217.182.140.238",
            "arrangement": {
              "__instances": 4,
              "__cpu": 1,
              "__memory": 1,
              "__ioperf": 1,
              "__iopsintensive": false,
              "__bandwidth": 1,
              "__resilience": 1,
              "mininstances": 4,
              "maxinstances": 4,
              "cpu": 1,
              "memory": 1,
              "bandwidth": 1,
              "failurezones": 1
            }
          }
        },
        "configuration": {}
      },
      "sep_service": {
        "instances": {
          "quickstart_sep_service_261": {
            "id": "92bb036d-db80-47e3-b909-3254e5fef767",
            "privateIp": "10.1.2.38",
            "publicIp": "217.182.140.238",
            "arrangement": {
              "cpu": 1,
              "memory": 1,
              "bandwidth": 1,
              "failurezones": 1,
              "mininstances": 1,
              "maxinstances": 1
            }
          }
        },
        "entrypoint": {
          "secrets": {},
          "sslonly": false,
          "domain": "bank-frighten.argo.kumori.cloud",
          "instancespath": false
        }
      }
    }
  }
```

```
  }
}
```

Undeploying an existing services

To undeploy a service we only need the service id we get when the service is deployed. For example, to undeploy the previously deployed service

```
kaze undeploy slap://quickstart/deployments/20170713_095222/d7189989
```

Unregistering elements in a stamp

Not implemented yet.

Reference manual

For more details, run `kaze -h`.

Note that some of below commands has optional `--stamp` which indicates the working stamp, however in this version of `kaze`, `--stamp` is mandatory, otherwise your command will fail.

- `init`
  - Initializes the current directory as a Kumori PaaS workspace.
- `bundle <path+>`
  - Creates a zip bundle including all elements pointed by path.
- `register <path+> [--stamp]`
  - Registers the elements pointed by the paths.
- `info -r <type>|--request=<type> [--stamp]`
  - Retrieves specific information from a stamp.
  - Currently `type` only can be `deployments`, i.e. retrieve deployment information.
- `deploy <path+> [--stamp]`
  - Deploys the deployment manifests pointed by the paths.
- `undeploy <uri+> [--stamp]`
  - Undeploys a deployed service.
- `stamp [options] [command]`
  - `stamp add <id> <url>`
    - Adds a new stamp to the configuration file.
  - `stamp rm <id>`
    - Removes a stamp from the configuration file.
  - `stamp switch <id>`

    - Sets the default stamp.

Pending:

- User authentication.
- Include a template mechanisms to help new components creation process using command. For example, a `component` command with a `init` subcommand and a `template` parameter might create a new component using the indicated template. By default, kaze will come with ECloud native templates. However, new templates might be installed and used by calling another command like `template add`.
- Templates will include a a flyjs/taskr file with some predefined targets for installing dependencies and creating a distributable bundle. Templates can be for *components* and for *libraries*. A component template should be able to create an ECloud bundle including all needed Manifests. Library templates bundles will create a ready-to-use bundle.
- Command `install`: compiles a component. By default, it will install the dependencies using npm install by using the base ecloud image. The image to use can be configured or indicated by parameters. The installation process can be customized by adding a *fly* file to the component root folder.
- Command `component` to manage components. It will initially have a single subcommand:
    - `init`: creates an initial structure of a component with a Manifest and the code section. It requests the name and version of the component. Optionally, other aspects can be requested like the runime and the type (`nodejs`,`java`, ...). It also might ask for resources and parameters.
    - `delete`: removes a component from the workspace.
- Command `service` to manage services.
    - `init`: in this case, the system will add for the name, the type and, then, the roles. It even might ask for how the channels are connected. It might also ask for resources and parameters.
- Command `deployment` to manage deployments.
    - `init`: it will ask for the service to be deployed. It might also ask for parameters, resources and roles initial configuration.
- Command `runtime` to manage runtimes:
    - `init`: it might ask for the parent runtime.
    - `install`: to install a runtime in your local Docker file.
- Command `test` to manage tests.
- Command `resource` to manage resources. This might be splitted in `volume`, `domain`, ...
- Local stamp management from kaze.
- Now, `register` command asks for a filename per path. It would be nice to not ask this to users since you are directly requesting to register your components.