C2Net Quick Start Guide

Table of Contents

1. Introduction 1
2. Environment Setup 2
    2.1. Software Prerequisites 2
    2.2. Install ***ECloud*** Dependencies 5
    2.3. Install C2Net Platform Image 6
3. Running the Example 7
    3.1. Overview 7
    3.2. Download the Example 7
    3.3. Compile the Test Bundle Components 8
    3.4. Build the Test Bundle 9
    3.5. Deploying Test Bundle 9
    3.6. The Test Results 10
4. Understanding the Example 13
    4.1. The Echo Service 13
    4.2. Echo Component 14
    4.3. Echo Service Application 18
    4.4. Echo Deployment 20
    4.5. Summary 21
5. C2Net Integration Testing Framework 22
    5.1. Echo Test Bundle 22
    5.2. The Tester Service 23
    5.3. Integration Testing Tool 25
6. Appendix 28
    6.1. Installation of a specific version of Docker 28
    6.2. C2Net FTP Server 28

1. Introduction

This Quick Start Guide will show you through an example how to:

- Prepare and pack a service, the Echo service, a simple service with just one component.
- Prepare another service, the Tester service, for testing the Echo service.
- Upload them in the Integration Testing Framework, in order to launch (eventually) the test in the ECloud stamp.
- Access the results.

In order to do so, this Quick Start Guide is structured in the following sections:

- Environment Setup : for setting up the environment for running the example.
- Running the Example : a step by step guide to run an example.
- Understanding the Example : Provides some hints and details about the example, giving instructions on how to develop a service and test it in C2NET project.
- C2Net Integration Testing Framework: Explains in detail the process of executing the test in C2Net.

Throughout this document, we refer to the C2Net PaaS for SLA as simply ***ECloud***

2. Environment Setup

The instructions of this Quick Start Guide are based on Ubuntu 16.04, which we highly recommend. It is also possible to install and run the ***ECloud*** Environment in other platforms such as Windows and Mac, but this is out of scope of this document.

2.1. Software Prerequisites

There are several software tools which you need to install before you can install the ***ECloud*** Environment in your computer. Additionally, there are some utilities, which are not mandatory, but that will help you develop CoffeeScript code, to zip your developments and to send/get your files to/from FTP server.

Mandatory software you need to install:

- Docker
- Nodejs

Additional tools:

- CoffeeScript
- Text Editor
- Git
- ZIP Utility
- FTP Client

2.1.1. Docker

To install Docker in Ubuntu 16.04, type following commands:

```
sudo apt-get update
sudo apt-get -y install docker.io
```

The recommended version is **1.10.3**. Verify the Docker version as follows:

```
docker --version
```

Result should be:

```
Docker version 1.10.3, build 20f81dd
```

If your Docker version is not 1.10, read the Appendix about how to install a specific version of Docker in Ubuntu.

2.1.2. Nodejs

In order to install Nodejs, you can whether visit the site https://nodejs.org/en/ and follow the instructions or you can install it from command line:

```
sudo apt-get install nodejs
```

To verify the installed version, just type:

```
node --version
```

Recommended Nodejs version is 4.3.2

2.1.3. CoffeeScript

Nodejs will also install in your computer npm, the Node Package Manager. Verify that is installed in your computer:

```
npm --version
```

If is not installed, you can install it by typing:

```
sudo apt-get install npm
```

and verify installation:

```
npm --version
```

Once npm is installed, type the following command to install CoffeeScript:

```
npm install -g coffee-script
```

and verify installation:

```
coffee --version
```

2.1.4. Text Editor Tool

It is recommended that you install a text editor with CoffeeScript HighLighting. There are many available, some of them are listed below:

- Sublime → https://www.sublimetext.com/
- Brackets → http://brackets.io/
- Webstorm → https://www.jetbrains.com/webstorm/
- Nodeclipse → http://www.nodeclipse.org/

2.1.5. GIT

The git version control system can be installed with the following command:

```
sudo apt install git
```

Verify git is installed:

```
git --version
```

It is recommended that you set up some global configuration variables

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

To see a list of global configuration variables:

```
git config --list
```

2.1.6. ZIP Tool

There are many tools available to zip and unzip files. You can install *zip* by typing the command:

```
sudo apt-get install zip
```

and to verify installation succeed:

```
zip --version
```

2.1.7. FTP Client

Again, there are many FTP Clients tools available, some of them listed here:

- Filezilla → http://filezilla.sourceforge.net/
- FireFTP → http://fireftp.mozdev.org/

- gFTP → http://gftp.seul.org/

For example, to install FileZilla you can type the following command:

```
sudo apt-get install filezilla
```

and verify if installation succeeded:

```
filezilla --version
```

2.2. Install *ECloud* Dependencies

The ***ECloud*** team has developed a tool which will install the required dependencies in your computer. Download from Open Source Projects the SDK project from the PaaS for SLA by cloning the project. You can do it by typing the following command:

```
git clone
https://USERNAME@opensourceprojects.eu/git/p/c2net/cpl/paas4sla/sdk/cod
e c2net-cpl-paas4sla-sdk-code
```

Change to your Open Source Project USERNAME to download the project from the repository

The project code will be placed into the destination folder *c2net-cpl-paas4sla-sdk-code*. You are free to place it in another folder.

The folder structure will be as follows:

```
c2net-cpl-paas4sla-sdk-code
├── testing
│   └── ...
└── tools
    ├── install-dependencies.sh
    ├── README.adoc
    ├── runtime-tool
    └── runtime-tool.sh
```

Then navigate to the tools folder and install dependencies:

```
cd c2net-cpl-paas4sla-sdk-code/tools
./install-dependencies.sh
```

2.3. Install C2Net Platform Image

Once all dependencies are installed, you can install the C2Net Runtime Image into your Docker Installation. As for now, it is the only image available for the C2Net Project. Navigate into the tool folder and type following command:

```
 ./runtime-tool.sh install -n
eslap://c2netproject.eu/runtime/java/dev/1_0_0
```

To verify the list of installed docker images, type the following command:

```
docker images
```

and you should see in the list the C2Net Runtime Image:

REPOSITORY	TAG	IMAGE_ID	CREATED	SIZE
c2netproject.eu/runtime/java/dev	1_0_0	b1d768d84297	3 weeks ago	906 MB

3. Running the Example

Once environment has been setup, it is time to explore the example.

3.1. Overview

The Echo Service consists of a single component; the Component Front End. The CFE Component has a *reply* channel which is connected to the provided service channel. This implies that all incoming request to the *service* will be forwarded to the *sepdest* endpoint of the CFE Component.

Figure below illustrates the example:

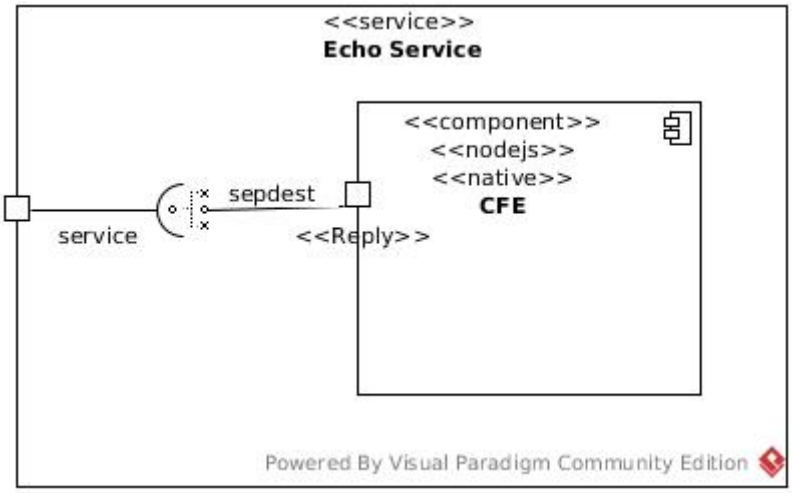

Figure 1. Echo Service

When a message arrives to the *service*, the CFE component will respond with the same message if the url is */echo* or return an error in other case.

3.2. Download the Example

Download the example from the Open Source Project Portal by typing the following command:

```
git clone
https://USERNAME@opensourceprojects.eu/git/p/c2net/cpl/paas4sla/training/code c2net-cpl-paas4sla-training-code
```

Change to your Open Source Projects User.

This will create the *c2net-cpl-paas4sla-training-code* folder. Navigate to the *Quickstart* folder, you should find the test example with the following folder structure:

```
.
echoTest
├── echoTestBundle                          (1)
│   ├── echoService                         (2)
│   ├── testerService                       (3)
│   └── Manifest.json                       (4)
└── 20160804T1200_trainingTeam.json         (5)
```

(1) Echo Test Bundle

(2) Echo Service

(3) Echo Tester

(4) Test Manifest

(5) Integration Test File

3.3. Compile the Test Bundle Components

In order to build the Test Bundle, you will need to generate your dependencies for each of your CoffeeScript components. We assume that you have previously installed ***ECloud*** Native Runtime as explained in section Install C2Net Platform Image.

To compile and generate the node_modules execute the following command inside the *src/* folder for each of component (same level as *package.json*):

3.3.1. Echo Component

```
cd echoTestBundle/echoService/components/cfe/code/src
docker run --rm -v $(pwd):/tmp/component
c2netproject.eu/runtime/java/dev:1_0_0 -c 'cd /tmp/component && npm
install'
```

If the code was compiled correctly then folder *node_modules* should have been created.

3.3.2. Tester Component

And then, navigate to the parent folder where you unzip the example and compile the Tester Component Code:

```
cd testerService/components/cfe/code/src
docker run --rm -v $(pwd):/tmp/component
c2netproject.eu/runtime/java/dev:1_0_0 -c 'cd /tmp/component && npm
install'
```

Again, verify that the *node_modules* folder was created.

3.4. Build the Test Bundle

Now it is time to put the Echo Service folder, the Tester Service folder, and the Test Manifest file inside a zip file. Navigate to the folder `echoTestBundle` *and then zip with the following command*

```
zip -r TEST_NAME echoService testerService Manifest.json
```

Where the TEST_NAME will be the name of the file which will be created.

3.4.1. Test Name

To avoid colliding with files from others partners, the file name should have the format **YYYYMMDDThhmm_username.json**.

3.5. Deploying Test Bundle

In order to exchange Test Bundle files and the results of the execution of test cases from the ***ECloud*** Platform, the C2Net Team has set up a FTP server to act as an intermediary service.

The Integration Testing Tool (ITT) will deploy a Test Bundle if a new Integration Test File is found in the FTP Server.

3.5.1. Integration Test File

The *20160804T1200_trainingTeam.json* is the Integration Test File for this example. Please modify the name in order not to collide with others as mentioned in the Test Name.

The source is as follows

```
{
  "name": "TEST_NAME",          ①
  "bundle":"TEST_NAME.zip",     ②
  "action": "start"             ③
}
```

① "name" : choose a the name which identifies uniquely your test

② "bundle" : filename of your TEST_NAME .zip file located in /bundles

③ "action" : "start", this parameter can be modified in the future to accept more parameters

Change the string TEST_NAME with **YYYYMMDDThhmm_username.json**

3.5.2. Upload files to FTP Server

Use your preferred FTP Client to upload the Test Bundle and the Integration Test File or follow

the instructions:

Connect to the server as C2Net :

```
sftp c2net@130.206.116.69
```

When prompted, type the login as password. You will be logged into the /srv/ftp folder.

Upload the Test Bundle into the /bundles folder

```
sftp> put TestEchoExample.zip /srv/ftp/integration-test/bundles
```

When the upload is completed, upload the Integration Test File into /manifests folder

```
sftp> put 20160804T1200_trainingTeam.json /srv/ftp/integration-
test/manifests
```

Wait until uploading the Test Bundle is completed before uploading the Integration Test File because the ITT will attempt to download the bundle test file before is ready on the server.

Now the ITT will detect the request of a new test execution and will run according to the steps described in section Integration Testing Tool

3.6. The Test Results

Once the Tester Service has finished the execution of test cases, and if everything executed correctly, the ITT will upload into the *'logs'* FTP server folder, two files:

- TEST_NAME-TEST.json with th result of the test cases
- TEST_NAME-LOG.json.gz with the logs

Absolute path in the FTP Server is */srv/ftp/integration-test/logs*

3.6.1. TEST_NAME-TEST.json

Contains the result of the execution of the test cases (Described later on echoTester .coffee Section). The file is as follows:

```
{
  "name": "TEST_NAME",
  "finished": true,
  "message": "Test finished",
  "result": [
    {
      "test": "before",
      "success": true,
      "message": "before() is empty"
    },
    {
      "test": "pathInvalidTest",
      "success": true,
      "message": "Response error code 404 as expected"
    },
    {
      "test": "postEchoTest",
      "success": true,
      "message": "Response was echoed: Will you echo me? == Will you
echo me?"
    },
    {
      "test": "after",
      "success": true,
      "message": "after() is empty"
    }
  ]
}
```


If the execution was NOT successful, then the file *TEST_NAME-TEST.json* will contain the error message.

3.6.2. TEST_NAME-LOG.json.gz

A compressed file containing the logs from the ***ECloud*** Stamp in Json format. An example of a log entry is as follows:

```
{
  "_index": "log-2016.07.27",
  "_type": "log",
  "_id": "AVYszgySeaJTJT1fNQcC",
  "_score": 12.70393,
  "_source": {
    "message": "Winston Logger reconfigured (id=c4692904-edbc-4d06-
9e08-fbc8a1edc1ff, level=4)",
    "@version": "1",
    "@timestamp": "2016-07-27T14:41:56.263Z",
    "host": "10.0.0.41",
    "port": 35264,
    "type": "log",
    "vm": "de256eb5-bcf0-4e4d-926b-39d33cf87214",
    "context":
"slap://echo_basic.examples.ecloud/deployments/20160727_144126/807fefdf
",
    "owner": "echo_basic.examples.ecloud_cfe_tester_82",
    "slap_timestamp": "2016-07-27T14:41:56.133Z",
    "level": "info",
    "label": "node"
  }
}
```


The idea is that you use the file as it is to view it in Kibana; an open source data visualization plugin for Elasticsearch.

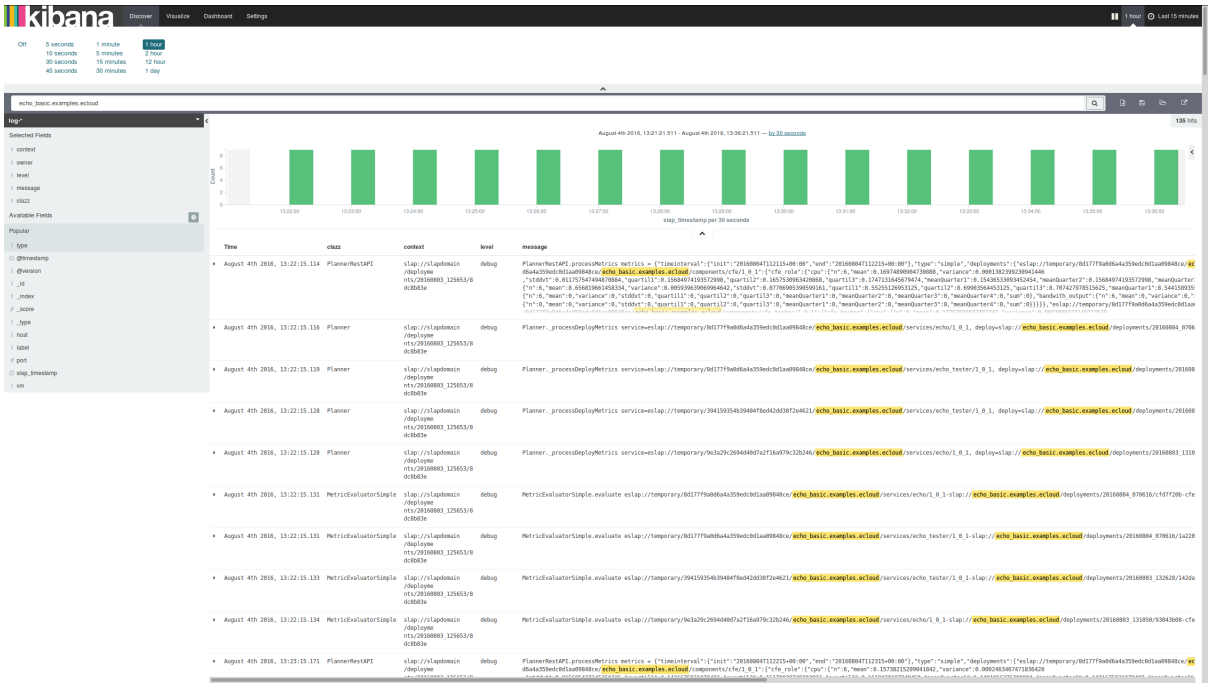

Figure 2. Kibana Screen Shot

For more details, check the guide describing how to view the logs in C2Net Open Source Project, located in the folder c2net/cpl/paas4sla/sdk/code/testing/log-viewer.

4. Understanding the Example

Now, let's look into the example step by step.

The example consists of an echoTest folder containing the Echo Test Bundle. A Test Bundle is a folder which contains our Echo Service, the Tester Service and a file which describes the content.

4.1. The Echo Service

In this section, we are going to focus on how to develop the Echo Service.

Basically, an ***ECloud*** Service defines the topology of the components that compose the service as well as the channels that the components use to communicate between them and with other services. When components become runnable parts of a service, they are referred as Service Roles a the service context. Service Roles are included in the *depended* or *provided* Channel Connectors.

This guide introduces the basics of the ***ECloud*** Platform. Please refer to the ***ECloud*** Manual for a complete specification.

The Echo Service consists of a single component; the Component Front End. The CFE Component has a *reply* channel which is connected to the provided service channel. This implies that all incoming request to the *service* will be forwarded to the *sepdest* endpoint of the CFE Component.

Figure below illustrates the example:

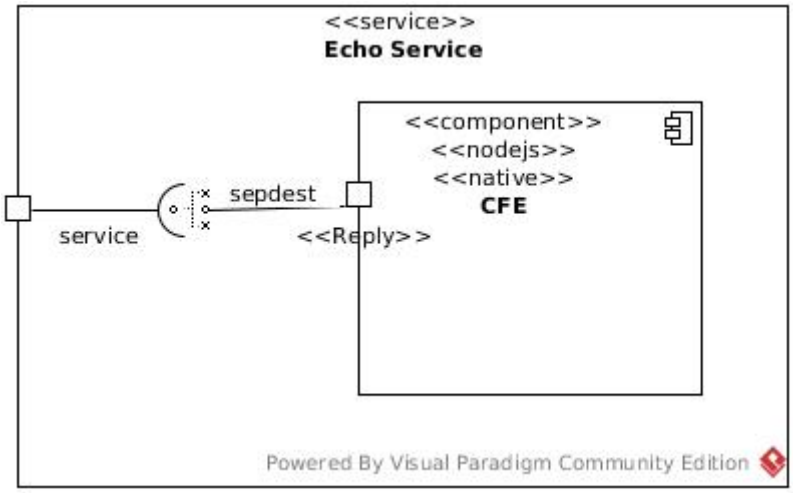

Figure 3. Echo Service

When a message arrives to the *service*, the CFE component will respond with the same message if the url is */echo* or return an error in other case.

Let's look into the Echo Service example source. Navigate to the echoService folder inside the echoTestBundle. The tree structure is as follows:

```
echoService/
├── components
│   └── cfe                                  ①
│       ├── code
│       │   ├── src
│       │   │   ├── src
│       │   │   │   └── index.coffee         ②
│       │   │   └── package.json             ③
│       │   └── Manifest.json                ④
│       └── Manifest.json                    ⑤
├── service
│   └── Manifest.json                        ⑥
└── Manifest.json ⑦
```

The `echoService` folder structure is a proposal of a ***ECloud*** Bundle containing a Service. It is a good example to start with the development of your components. We will explore each of them in following sections:

① Echo Component: The runnable nodejs code to be deployed and execute in the ***ECloud*** Stamp. It is configured in the component manifest.

② index .coffee: Component's Source code

③ package.json: Library dependencies.

④ Code Blob Manifest: Designed to carry code.

⑤ Component Manifest: Describing the component.

⑥ Echo Service Application: The Echo Component will be a part of the Echo Service Application, declaring a role and a channel to be able to offer the echo functionality. The service manifest file describes the Service Application.

⑦ Echo Deployment: The Service Application can be deployed into ***ECloud*** Stamp to become a service. The initial deployment is specified via the deployment manifest.

4.2. Echo Component

A component should be seen as an autonomous executable. The Echo Component *provides* the echo functionality. There is only one folder inside the folder components which corresponds to the Echo Component. It is a simple example component and as such, it is only composed by a single file which contains the cfe.class.

4.2.1. index .coffee

The component's source code file. There are many comments throughout the file which will help you understand the code.

Class Structure

The Echo Component is a class that inherits from the ***ECloud*** Component Class and have the

following methods:

```
class Cfe
    constructor: (                          (1)
        @slap,                              (2)
        @role,                              (3)
        @iid,                               (4)
        @incnum,                            (5)
        @localData,                         (6)
        @resources,                         (7)
        @parameters,                        (8)
        @requires,                          (9)
        @provides                           (10)
    )     ->
    run: ->                                         (11)
        # ...
    shutdown: ->                                    (12)
        # ...
    reconfig: (resources, parameters) ->            (13)
        # ...
    startHttpServer: () =>                          (14)
        # ...
    onHttpRequest: (request, response) =>           (15)
        # ...
    onHttpError: (error) =>                         (16)
        # ...
    _echoFunction: (postData, response) =>          (17)
        # ...
module.exports = Cfe (18)
```

(1) The constructor returned is used by the ***ECloud*** Stamp to create an instance of the component

(2) The `slap` object, through which the component can interact with the platform

(3) The `role` this instance belongs to.

(4) The `instance id` that ***ECloud*** assigns to this instance.

(5) The `incarnation number` of the instance. It indicates how many times the instance has been started.

(6) The path where to store local data.

(7) The set of resource objects assigned to this instance.

(8) The set of configuration parameters, organized as a dictionary with their names as keys.

(9) The set of `requires` channels organized as a dictionary.

(10) The set of `provides` channels organized as a dictionary.

(11) Method invoked to start execution of the instance.

(12) Method invoked to warn that the instance is going to be stopped.

⑬ Change of configuration parameters/and or resources.

⑭ Method that creates a HTTP server listening on channel `sepdest`. It also associates *request* event to *onHttpRequest()* method and *error* event to *onHttpError()* method

⑮ Method event handler invoked when new http request comes to the server. If path is *'/echo'* then it delegates the response to *_echoFunction()*. If path is not *'/echo'* then returns an error message with code *'404'*

⑯ Method handler for error event. Creates some error messages to the logger and it throws the error.

⑰ Internal method which writes the incoming message back to the response. This is where the logic of the component is.

⑱ The last line where the methods of the class are exposed to make them accessible to the ***ECloud*** Stamp.

You need to have a *super* call with correct parameters at the beginning of each method, as shown in the example.

Behaviour

The CFE *run()* method is invoked by the runtime when the component is instantiated. This method calls the *super ()* method of the component and then creates a HTTP server by calling the *@startHttpServer()*. The HTTP server will be listening on channel `sepdest`. Also notice that the *@startHttpServer()* will link the *request* event to *onHttpRequest()* method and *error* event to *onHttpError()* method.

Now, when a new request event occurs, the *@onHttpRequest()* will handle the request and analyze the path. If the path is */echo* then it will forward the request to the *@_echoFunction()* or return error with code *'404'*

Finally, the internal method *echoFunction()* will return the incoming json object in the response.

Logging facilities

In order to use logging facilities you must use the library *Slaputils* as follows:

```
Slaputils.setLogger [Cfe]
```

After that, you can use the logger facility to the level of your choice:

- `@logger.debug`: Designates fine-grained informational events that are most useful to debug an application.
- `@logger.info`: Designates informational messages that highlight the progress of the application at coarse-grained level.
- `@logger.warn`: Designates potentially harmful situations.
- `@logger.error`: Designates error events that might still allow the application to continue running.

4.2.2. package.json

Relative to CoffeeScript developing. Contains documentation, library dependencies and versions.

The main file points to index.coffee, which is the entry point of your component *index.coffee*.

```
{"main": "src/index"}
```

4.2.3. Code Blob Manifest

The Code Blob Manifest is only designed to carry code.

```
{
  "spec" : "http://eslap.cloud/manifest/blob/1_0_0",     (1)
  "name" : "cfe-code-blob"   (2)
}
```

(1) Manifest specification version type: Blob Code Manifest

(2) Identifying the code inside the bundle, must be different for each bundle

4.2.4. Component Manifest

Describes the component, which is an autonomously runnable executable among with the channels the component requires or provides. The Echo Example doesn't use any additional configuration properties.

```
{
  "spec": "http://eslap.cloud/manifest/component/1_0_0",     (1)
  "name": "eslap://echo_basic.examples.ecloud/components/cfe/1_0_1", (2)
  "runtime": "eslap://c2netproject.eu/runtime/nodejs/1_0_0",     (3)
  "code": "cfe-code-blob",          (4)
  "channels": {                     (5)
    "provides": [{                  (6)
      "name": "sepdest",            (7)
      "type": "eslap://eslap.cloud/channel/reply/1_0_0",     (8)
      "protocol": "TBD"
    }],
    "requires": []                  (9)
  },
  "configuration":{                 (10)
    "resources": [],
    "parameters": []
  },
  "profile": {
    "threadability": "*"
  }
}
```

(1) Manifest specification version type: ***ECloud*** Component Manifest

(2) Name of the component, will be used in the service manifest to register it with a *role*. The domain in this example `echo_basic.examples.ecloud`

(3) Runtime to execute the component. When writing this document, the C2Net runtime is `eslap://c2netproject .eu/runtime/nodejs/1_0_0`

(4) Referencing to the code blob name from Code Blob Manifest

(5) List of channels in the component.

(6) List of provided channels in the component.

(7) Name of the channel

(8) Type of the channel

(9) List of required channels.

(10) Configuration resources and parameters

Modify the domain of your Component in the name property field to avoid overwriting others components.

4.3. Echo Service Application

Now, it is time to define the Service Application. This is done through the service manifest. In this manifest you identify the components of the service, the role of the components in the service and how they interact through channels.

In this example, we only have one component which has a `cfe_role` and one channel called `service` which links the incoming service calls to the cfe component through this channel.

Let's look into the details of the Echo Service Manifest in the next section.

4.3.1. Echo Service Manifest

Specifying a service application requires specifying its topology: linking the service channels to the component roles via the channel connectors:

```
{
  "spec": "http://eslap.cloud/manifest/service/1_0_0",                    ①
  "name": "eslap://echo_basic.examples.ecloud/services/echo/1_0_1",       ②
  "configuration": {                                                      ③
    "resources": [],
    "parameters": []
  },
  "roles": [{                                                             ④
    "name": "cfe_role",                                                   ⑤
    "component":
"eslap://echo_basic.examples.ecloud/components/cfe/1_0_1",                ⑥
    "resources": {},
    "parameters": {}
  }],
  "channels": {                                                           ⑦
    "provides": [{                                                        ⑧
      "name": "service",                                                  ⑨
      "type": "eslap://eslap.cloud/channel/reply/1_0_0",                  ⑩
      "protocol": "eslap://eslap.cloud/protocol/message/http/1_0_0"
    }],
    "requires": []                                                        ⑪
  },
  "connectors": [{                                                        ⑫
    "type": "eslap://eslap.cloud/connector/loadbalancer/1_0_0",           ⑬
    "depended": [{
      "endpoint": "service"                                               ⑨
    }],
    "provided": [{
      "role": "cfe_role",                                                 ⑤
      "endpoint": "sepdest"                                               ⑭
    }]
  }]
}
```

① Manifest specification version type: ***ECloud*** Service Application Manifest

② Name of the service. The domain is `echo_basic.examples.ecloud`

③ Configuration resources and parameters. Not used in this example

④ Roles of the service components

⑤ `cfe_role`: Role name for the CFE component

⑥ Registered Component name as specified in the Component Manifest

⑦ List of channels of the service. Channels are used to communicate between other services and with the Service Entrypoint.

⑧ List of channels provided by the service.

⑨ Channel provided by the service named *service*

⑩ Channel of type *reply*

⑪ List of required channels. This service is not calling any other service.

⑫ Describes how components are wired together and also to the service channels

⑬ Defines one connector wiring the *service* channel defined `service` with the `cfe_role` using the provided channel `sepdest`

⑭ Name of the channel end point to be used in the cfe_role (CFE Component). It will be available in the Component *'@provided'* parameter

In the Echo Example, only one channel is provided and wired to the cfe_role. By doing so, all incoming messages to the Echo Service will be forwarded to the CFE Component.

Modify the domain of your Service in the name property field to avoid overwriting others services.

4.4. Echo Deployment

The Service Application becomes a service when is deployed into a ***ECloud*** Stamp. Service and roles deployment configuration are specified via the deployment manifest.

4.4.1. Echo Deployment Manifest

Let's look into the Echo Deployment Manifest example:

```
{
  "spec": "http://eslap.cloud/manifest/deployment/1_0_0",    ①
  "servicename":
"eslap://echo_basic.examples.ecloud/services/echo/1_0_1",   ②
  "name": "deployment_echo_1",       ③
  "configuration": {                 ④
    "resources": {},
    "parameters": {}
  },
  "roles": {                         ⑤
    "cfe_role": {                    ⑥
      "resources": {
        "__instances": 1,
        "__cpu": 1,
        "__memory": 1,
        "__ioperf": 1,
        "__iopsintensive": false,
        "__bandwidth": 1,
        "__resilience": 1
      }
    }
  }
}
```

① Manifest specification version type: ***ECloud*** Deployment Manifest

② Name of the service application which will be deployed

③ Name of the deployment, to be used later in the Test Deployment Manifest

④ Configuration parameters, not used for this example

⑤ Defines the properties for each role. Each deployed component must have a role and thus an entry in the roles list.

⑥ Name of the role and the initial deployment resources

Now that we have created an example of a service, we are ready to deploy it into ***ECloud*** Stamp and execute some tests.

4.5. Summary

We have presented an overview of a basic ***ECloud*** Service, containing an Echo Component.

In the next section we will present in detail how a C2Net user can test the component in an ***ECloud*** Stamp using the C2Net Integration Testing Framework.

5. C2Net Integration Testing Framework

In this section, we will explain how to execute tests using the C2Net Testing Framework.

By the time of writing this guide, it is not possible to grant access to the ***ECloud*** Platform for C2Net Users. This is the reason why we have created the C2Net Integration Testing Framework (ITF), to enable the C2Net partners to test their components.

In order to exchange Test Bundle files and the results of the execution of test cases from the ***ECloud*** Platform, the C2Net Team has set up a FTP server to act as an intermediary service. You can find more details about FTP Server in the Appendix Section C2Net FTP Server

Let's look into the example of a Test Bundle.

5.1. Echo Test Bundle

In order to test a component, the user needs to create the following:

- Application Service: which is a service containing the components to be tested. We can refer it to as the Service Under Test.
- Tester Service: contains the test class, which the user will need to develop. The Test Class belongs to a Test Service, which is a service already created and the user only needs to code the test cases.
- A Test Manifest: which configures the test deployments of the Service Under Test and the Test Service.

Once these elements are created, the user will pack everything in a Test Bundle and upload it into a FTP server, where a daemon process will deploy it, execute the tests and return the results.

A Test Bundle is a directory structure where the top directory contains a file with name Manifest.json describing what is inside of it. In our example, the Test Manifest describes the Echo Service (which contain our Echo Component) and the Tester Service.

The example has the following folder structure:

```
.
├── echoService          (1)
├── testerService        (2)
└── Manifest.json (3)
```

(1) echoService: The Echo Service which we want to test.

(2) testerService: The Tester Service which have the test cases.

(3) Test Manifest: File describing the Test Bundle.

We have already presented the echoService in section The Echo Service. Let's look into the Test Manifest and the Tester Service in the following sections.

5.2. The Tester Service

The Tester Service has one CFE component which delegates the logic to the Tester Class. The folder structure is an example of a Service Bundle:

```
testerService/
├── components
│   └── cfe                                  ①
│       ├── code
│       │   ├── src
│       │   │   ├── src
│       │   │   │   ├── index.coffee         ②
│       │   │   │   ├── cfe.coffee           ③
│       │   │   │   ├── tester.coffee        ④
│       │   │   │   └── echoTester.coffee    ⑤
│       │   │   └── package.json             ⑥
│       │   └── Manifest.json                ⑦
│       └── Manifest.json                    ⑧
├── service
│   └── Manifest.json                        ⑨
└── Manifest.json ⑩
```

① The Test Service has only one component

② index .coffee: main code as specified in the package.json.

③ cfe .coffee: Component Front End.

④ tester .coffee: Tester Class from which your tests will inherit.

⑤ echoTester .coffee: The Class containing the test cases.

⑥ package.json: Library dependencies.

⑦ Code Blob Manifest.

⑧ Component Manifest.

⑨ Service Manifest.

⑩ Deployment Manifest.

We will focus on the Test Component, and more specifically on the EchoTester class defined inside the echoTester.coffee, which contains an example of test cases.

There are more manifest in the Tester Service, but you don't need to know more details about the Tester Service in order to create your test cases.

We introduce briefly the files of the Test Component:

5.2.1. index .coffee

It is the entry point of the Component and it only exposes the API of Cfe Class.

5.2.2. cfe .coffee

It is the Front End Component (CFE) which extends the Component Class. This class is where the *@tester* variable is instantiated inside the *run()* method.

```
@tester = new EchoTester()
```

When you create a new test class, you will need to instantiate your class and assign it to the `@test` variable of the CFE Class.

5.2.3. echoTester .coffee

This file contains the tests cases. If you need to create a new test class of your own, you can copy and rename this file to fit your needs and rewrite the tests.

Your test class needs to inherit from the Tester Class. The test cases are methods in your class whose name ends with **Test** and must return a promise with the result.

A promise represents the eventual result of an asynchronous operation.

We use library *Q* because it makes the use of promises easy to use.

Let's explore a snippet of the code:

```
class EchoTest extends Tester
  constructor: () ->                          (1)
  before: () =>                               (2)
  after: () =>                                (3)
  case1Test: () =>                            (4)
    return q.promise (resolve, reject) =>     (5)
        @request.post ... =>                  (6)
            resolve "test passed"             (7)
            reject "test failed"              (8)
  case2Test: () =>
  ...
  xxxTest: () =>
```

(1) Constructor method without parameters

(2) before() method. It will be executed before all test cases. It is an optional method that you could use, for example, to setup some initial configuration.

(3) after() method. It will be executed after all test cases have been run. This method is optional.

(4) Test case method 1: You can declare as many test cases as you want but the name of the functions MUST end with *Test*.

(5) Test cases MUST return a *promise*.

⑥ It is a wrapper of the *npm request package* with some default values which you can use to access the Service Under Test. This is the way you can use to make HTTP request to the Service Under Test.

⑦ When the test case passes, then the *promise* should return a *resolve* with an optional string message which will be shown in the result.

⑧ If your test case didn't pass, then your method should *reject* the promise along with a message.

Assign the instance of the tester class to the @tester variable in the CFE class cfe .coffee

Request is a simplified HTTP request client. For additional information visit https://www.npmjs.com/package/request

5.2.4. tester .coffee

The Tester Class is responsible of running the test cases. It obtains deployment setup parameters of the Tested Service and executes the test cases declared in the EchoTester Class. You don't need to know any further details of this class to create your tests.

5.2.5. Test Manifest

The Test Manifest is a json file which defines the test series to be executed. In order to simplify the example, only one series is present in this test bundle. Both the *echoService* and the *testerService* are folders containing those services. The content of the file is as follows:

```
{
  "spec": "http://eslap.cloud/manifest/test/1_0_0", ①
  "series" : [
    {
      "tester" : "deployment_tester_1",      ②
      "tested" : "deployment_tested_1"       ③
    }
  ]
}
```

① Manifest specification version type: Test Manifest

② Name of the deployment Tester Service, which is is defined in the Deployment Manifest of the tester service

③ Name of the deployment of the Service Under Test, as defined in the Deployment Manifest of the service.

5.3. Integration Testing Tool

The Integration Testing Tool (ITT) is a daemon process which will handle autonomously the deployment and the execution of test cases in ***ECloud*** Stamp. Let's see an overall picture:

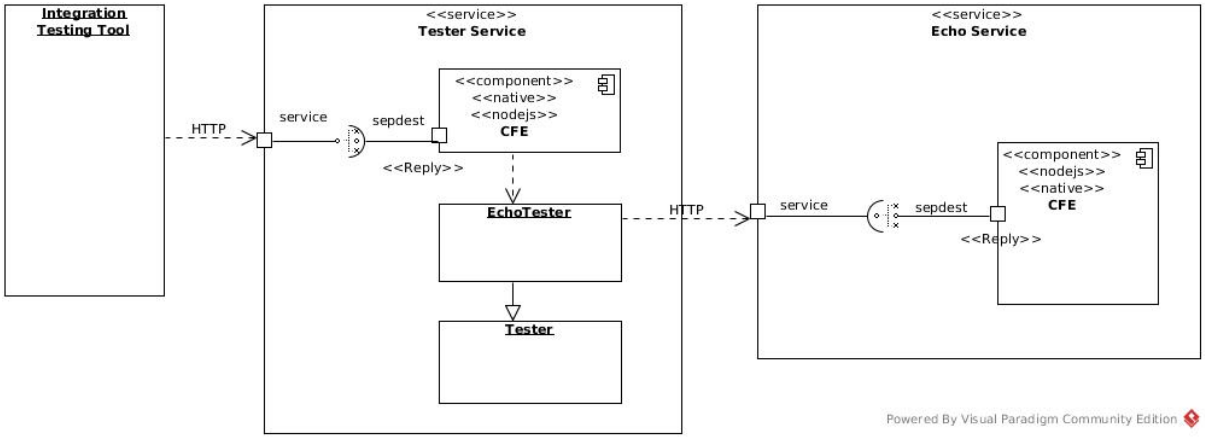

Figure 4. C2Net Testing Framework

The ITT expects two files in order to start the execution of a test:

- Test Bundle: explained with an example in section Echo Test Bundle
- Integration Test File: file describing the test to be run, explained later in section Deploying Test Bundle.

When ITT detects that a new Test Manifest has been uploaded to the server, it will perform the following actions:

1. Download the Test Manifest
2. Download the Test Bundle
3. Register the Test Bundle into ***ECloud*** Stamp
4. Configure the Tester Service with deployment details of the Service Under Test
5. Run the test cases
6. Create a file with the results of the test cases and upload it to the FTP Server
7. Collect the logs from ***ECloud*** Stamp, create a gzip file and upload it to the FTP Server
8. Undeploy and unregister the services

5.3.1. Deploying Test Bundle

In order to exchange Test Bundle files and the results of the execution of test cases from the ***ECloud*** Platform, the C2Net Team has set up a FTP server to act as an intermediary service.

The Integration Testing Tool (ITT) will deploy a Test Bundle if a new Integration Test File is found in the FTP Server.

Integration Test File

So let's create a Json file with the following structure:

```
{
  "name": "TEST_NAME",           (1)
  "bundle":"TEST_NAME.zip",      (2)
  "action": "start"              (3)
}
```

(1) "name" : choose a name which identifies uniquely your test

(2) "bundle" : filename of your TEST_NAME .zip file located in /bundles

(3) "action" : "start", this parameter can be modified in the future to accept more parameters

Once the Integration Test File has been created, connect to the FTP Server and upload it along with the Test Bundle

5.3.2. The Test Results

Once the Tester Service has finished the execution of test cases, the ITT will upload both the log file and the result of the test execution into the *'logs'* FTP server folder, in the path /srv/ftp/integration-test/logs. The created files are as follows:

- TEST_NAME-TEST.json: Text file containing the tests execution results in JSON format.
- TEST_NAME-LOG.json.gz: A file containing the service log entries. Such file is made of JSON documents in *Elasticsearch* format, and compressed using *gzip*.

The name of the file **Test** is the name of your test as appears in the Integration Test File, described in section Deploying Test Bundle

There is a guide describing how to view the logs in C2Net Open Source Project, located in the folder c2net/cpl/paas4sla/sdk/code/testing/log-viewer.

6. Appendix

6.1. Installation of a specific version of Docker

To install the specific version, open a terminal and run the following commands:

```
sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80
--recv-keys 58118E89F3A912897C070ADBF76221572C52609D
mkdir -p /etc/apt/sources.list.d
sudo echo deb https://apt.dockerproject.org/repo ubuntu-trusty main >
/etc/apt/sources.list.d/docker.list
sudo apt-get update
sudo apt-cache show docker-engine | grep Version
sudo apt-get install docker-engine=1.10.3-0~trusty
```

and verify installation:

```
docker --version
```

Result should be:

```
Docker version 1.10.3, build 20f81dd
```

6.2. C2Net FTP Server

When the ITT detects a new Integration Test file, it will execute the deployment and test execution as described in section Integration Testing Tool

The folders in the FTP Server have the following structure:

```
/srv/ftp/integration-test/
├── bundles            (1)
│   └── readme.txt
├── logs               (2)
│   └── readme.txt
└── manifests          (3)
    └── readme.txt
```

(1) Folder used to put your test bundles

(2) Folder where the Testing Tool will put the log files

(3) Folder to put the test manifest

There are readme.txt files inside each folder for further explanations.