
# Macaw Essentials

## Developing *Microservices* using Macaw Platform

**by Macaw Team**

# Chapter 1: Microservices and Macaw

## Microservices

Applications have emerged as a focal point of IT strategy as CXO leaders begin to realize that *"software is eating the world"*. Bold new entrants are strongly disrupting incumbent vendors with their innovative software solutions. With traditional IT and business applications designed monolithically and deployed on-premises, enterprises are struggling to keep pace with digital enterprise requisites. Enterprises are looking for ways to bring agility and support for rapid delivery cycles to their business applications without much disruption.

Modern cloud native applications and transformed legacy applications should ideally leverage Microservices architecture, which consists of a collection of the right-sized services that each run in highly portable environments called Containers. Enterprise and software vendors will benefit significantly from such modern architectures through accelerated development and delivery of innovative business models. The direct challenge here is in operating the environment and the infrastructure stack to adopt a Microservices-based DevOps model. This can be time consuming and often shifts focus away from delivering value through applications and services. Using a platform built upon a curated stack filled with the best of breed open source components and supporting popular runtimes will save time exponentially. For this main reason, Microservices are here to stay.

## Introducing Macaw Platform

Macaw is a *prescriptive Microservices platform* that helps enterprises accelerate their cloud native journey. Macaw is a natively container-ready platform that provides both the necessary environment plus tools to build, deploy, and manage modern applications based on Macaw Microservices.

Macaw provides various built-in and readily usable core application services like database, security, messaging, registry, API gateway, load-balancer/HA etc. that are essential for rapid development and turn-key operations. Macaw SDK supports building Microservices using popular programming languages such as Java, Python, JavaScript, NodeJS with a plan to add support for other programming platforms like .NET in the near future.

Macaw DevConsole is a browser-based graphical interface for DevOps professionals to administer the life cycle of Microservices. Macaw Platform enables new cloud native applications as well as modernizing existing core/legacy applications through value added integrations and exposing their true value for external end user/customer consumption through APIs without any major disruptions.

The key capabilities of Macaw Platform include:

**Multi-Cloud Support**: It can run on any cloud, public or private, and support most on-premises infrastructure platforms.

**Container ready**: It has built-in support for standard container formats like Docker, making it easy to build, compose, deploy and move your workloads.

**Built-in security and scalability**: Enterprises can define fine-grained policies for access management, scaling, and other life cycle management operations and let the platform natively support them. Its built-in workflow manager can be used to embed approval steps where required to provide an additional layer of administrative controls.
**Self-governance**: It is built on the core principle of providing pervasive governance to ensure it provides consistent operational behavior from an availability, performance, compliance, and security point of view.
**Blueprint support**: Developers and architects can group services into a logical entity called blueprint with common scaling policies and access controls. Once Microservices are modeled via these blueprints, blueprints can be used to deploy these group of services with a single click and manage the life cycle of the application as per the policies specified.
**DevOps Console**: Microservices-based applications are dynamic in nature and Macaw DevConsole can help developers and operations team to deploy and monitor their runtime behavior for easy troubleshooting.
**Application Modernization**: Traditional applications can be transformed without any major disruption to embrace the principle of cloud-native applications. The built-in discovery mechanism will provide the detailed application footprint necessary to develop a guide map to transform these monolithic applications.
**Curated stack of open source technologies**: The platform is built using the best of breed open source technologies including Kafka, Zookeeper, Spark, Cassandra, Docker, Swarm, Elastic Search etc., to provide the scalability, multi-tenancy, and robustness enterprises demand along with enhancement of automation and governance features to make it easy to use for the DevOps.
**Bundled Core & Essential Services**: Multi-Tenancy, Real time Monitoring, Built-in Security, Micro Governance, Logging as a Service, Elastic Search, Container Ready, DB as a Service, and more.

## Chapter 2: Advantages of Macaw Platform

The Macaw platform provides the following business and technical benefits.

**Business Benefits:**

**Accelerates Innovation**: Macaw platform allows the opportunity to build business services at a faster pace as it arrives bundled with core productivity services. The platform provides DevOps automation, containerization capability, and allows 3rd party integration.
**Empowers Enterprises:** Macaw platform empowers enterprises to adopt Big Data and IoT technologies. The platform allows enterprises to have high throughput based on Kafka, real-time message correlation, and provides visibility. The platform has event driven and analytics support.
**Allows Technology Independence and Choice:** The container support offered by Macaw allows technology isolation. The platform allows enterprises flexibility of choice in technologies by offering polyglot runtime, supporting multiple PaaS, and Container systems.
**Improves Customer Satisfaction**: The platform allows resilient and fault tolerant services, thereby improving customer experience. The platform comes with programmable load balancers and high availability (HA) proxy support. The platform allows real-time monitoring of services and their performance. It has built-in self-governance and allows auto scaling policy configurations. All of the above help improve customer satisfaction.
**Eliminates vendor lock-in**: The platform eliminates vendor lock-in with easy workload mobility across clouds. The built-in containerization support drives portability across environments. The high scale messaging infrastructure eliminates tight coupling. Extensible SDK allows to plugin into multiple environments.
**Provides infra cost saving**: The platform optimized application scaling leads to sizable infrastructure cost savings. The platform provides fine-grained auto scaling policy support. It enables event driven serverless computing capability. The platform comes with Micro Analytics that optimize computer infrastructure needs.

**Technology Benefits**:

**Micro Analytics**: The platform provides Micro Analytics and Big Data capabilities with granular and aggregated metrics that allow service level insight.
**Scalable Runtime**: The platform provides scalable runtime with support to multiple infrastructures.
**Real-time visibility**: The platform provides tools forend-to-end message tracing, notifications, and event triggers.
**Multi Tenancy**: The platform allows fine-grained policy controls with hierarchical views for Tenant, Project, User, etc. and also allows secure isolation per tenancy policies.
**Blueprint**: The blueprint support allows to model complex applications with a clear definition of cloud and/or infra agnostic definition.
**Micro Governance**: The platform allows micro governance through fine-grained policies to the service level, allows compliance audit and reports.

## Chapter 3: Macaw Platform Editions

The Macaw platform provides two versions - Community Edition and Enterprise Edition. The features from both editions are explained below.

### Community Edition

The community edition is free for use by developer and application teams. This edition allows the developer to develop and test most of Macaw's features and is commonly community supported.

The following deployment options are available.

- Vagrant/Virtual Box – suitable for use on personal laptop/developer workstation
- VMware environments using OVF – suitable for team/shared environments
- AWS using the published AMI's – suitable for developers/teams using AWS
- Existing Linux Hosts with CentOS – Works on any Cloud/VM/Container/Baremetal.

If you are interested in Macaw Software Community edition, you can submit your request at https://www.macaw.io/macaw-download/ and the Macaw support team will provide the link to binaries and necessary documentation.

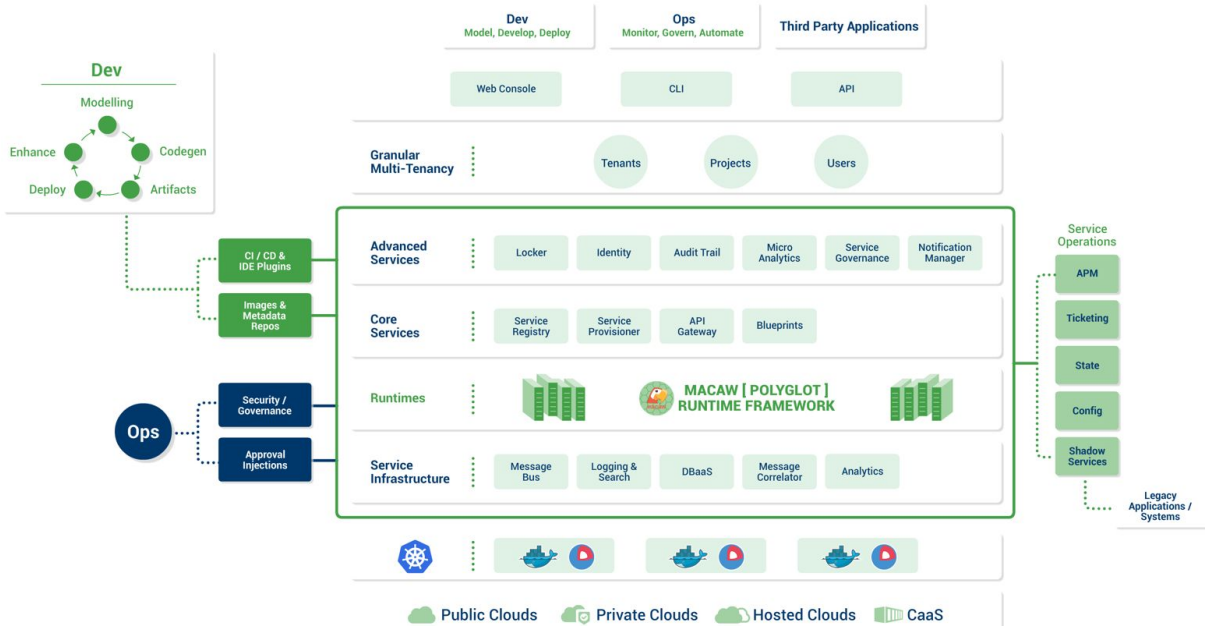
### Enterprise Edition

Macaw Enterprise Edition (EE) is designed to meet the needs of digital enterprises that require agile application development and effective governance. It is well suited for enterprises looking to build distributed cloud-native applications that are cloud agnostic, secure, multi-tenant, easy to manage, and must conform to complex corporate policies. The Bold content is available only on Enterprise Edition. While Community edition allows to try out various features on development platforms, for real time/production use, the customers need to use Enterprise Edition. The enterprise edition also offers additional advantages like identity management,and support to ticketing solutions apart from other features which are highlighted below.

The following diagram highlights the core differences between Community and Enterprise editions.

| Feature | Developer Version | Enterprise Version |
|---|---|---|
| **Install** | **Developer laptop**<br>•Vagrant + VirtualBox | **Hypervisor or Cloud**<br>•VMware Image<br>•AWS Image<br>•Any other cloud |
| **Scale** | **Single node only**<br>•Platform and services combined<br>•Suitable for Dev/Test | **Scale to multiple nodes**<br>•Price is per additional platform/service node<br>•Can install more platform nodes or horizontal nodes<br>•Suitable for staging/prod |
| **Application publish** | **To developer platform only**<br>•App publish/deploy only from local | **To any shared/public platform**<br>•Publish to team/shared repos |
| **Repositories** | **Local repositories only**<br>•Can only use native MDR/Docker registry | **Shared repositories**<br>•Can create/use native and shared MDR/Docker registries |
| **Tenants** | Default tenant only | Multiple tenants |
| **Container Infrastructure** | **Single node only**<br>•Single Docker node managed by Macaw | **Container Cluster**<br>-Managed by Kubernetes, Docker Swarm or Mesos |
| **Services** | **Core services**<br>•Database<br>•Logging<br>•IAM | **Core + Advanced services**<br>-All Core services<br>-Messaging over SSL<br>-Service Governance<br>-Locker<br>-Monitoring<br>-IAM with external providers<br>-Analytics |
| **Support** | **Community**<br>-Resources, FAQ, Forums etc. | **Enterprise support**<br>-PS, support contracts |

## Chapter 4: Macaw Architecture




Macaw is an enterprise grade prescriptive Microservices development and governance platform. The platform allows cloud native application development with an ability to govern Microservices. The platform supports server-less framework with full built-in security and scalability. The core components of Macaw Framework are:

**Runtime**: Central to Macaw framework is its polyglot runtime environment which provides support for multiple languages like Java, JavaScript, Python, .NET, etc.
**Service Infrastructure**: Macaw framework provides messaging, elastic search, DBaaS, message correlation, and Analytics services which are installed as part of Macaw Infrastructure installation. The core components as part of infrastructure installation are:

- Zookeeper
- Kafka
- Elastic Search
- MySQL
- Cassandra
- HAProxy
- Tomcat
- Redis

**Application Performance Monitoring (APM)**: APM service installation is optional and provides application performance monitoring agent and collector.
**Macaw Platform**: The platform provides services like Service Registry, Service Provisioner, Notification Manager, Identity Services, Console UI, and User preferences.
**Macaw Tools**: As part of this MDR (Meta Data Repository) and Docker Registry are installed. More on this is covered in a separate section.

**Macaw SDK**: Macaw Platform provides an SDK to enable developers to develop and publish Microservices. Macaw also provides an eclipse plugin for the developers who prefer Eclipse as IDE.

There are two types of Microservices that can be developed using Macaw Platform.
A) **Macaw Native Service** - It is like any other Microservice with full Macaw capabilities support and interacts with other Microservices via Macaw Platform.
B) **Macaw Shadow Service** - It is a shadow/proxy service which allows 3rd party services or applications exposed on Macaw Platform for service consumption by other services.

The development and governing of a Microservice using Macaw Platform can be divided into 5 broad categories:

**1. Develop**

Leverage an easy to use development pipeline to build simple as well as complex Microservices applications. Developers can use their choice of modeling frameworks and programming languages.

- Code generation tools
- Modeling: Yang & JSON
- Languages: Java, JavaScript, Python
- Stateless and Stateful Services
- Many built-in services: DBaaS, Logging as a Service.
- Build Tool integrations: Ant, Maven, and IVY.

**2. Publish**

Macaw supports both public and private repositories for developers to publish their images and metadata in a secure manner. Images can be tagged to keep track of the different versions. Multiple services can be bundled and published as a Blueprint for complex applications.

- Publish tools
- Private & Public Registry
- Docker Containers
- Blueprints: Metadata
- CI/CD integration

**3. Deploy**

Macaw provides both CLI and portal interface to manage deployment process. Enterprise can also attach runtime and access policies to ensure security, compliance, and consistency. Policies can be specified at the tenant level or project level or at service level.

- HA Proxy and scaling policies
- Orchestration tools - Kubernetes, Swarm
- Role based deployment policies: Dev, QA, Production
- Service Registry & API Gateway for routing
- Multi-Tenancy

**4. Operate / Govern**

Macaw provides several built-in capabilities to manage and govern distributed applications. Built-in analytics and diagnostics capabilities can guide the o,Ops team to simplify their operations through insights.

- DevOps Console
- Real-time Monitoring

- Service social graphs for message correlation
- Approval Injections
- Aggregated Logs and Search

**5. Update**

Macaw makes the update process easy and flexible through programmatic control of routing messages to multiple version instances. DevOps can manage different versions by tagging the instances.

- Programmatic HA proxies
- Canary release support
- Run multiple versions in parallel.
- API Gateway

## Chapter 5: Macaw Platform Installation

The Macaw Platform provides various installation options. For On-Premise options, Macaw platform can be installed on a local developer machine (Mac / Linux / PC) using VirtualBox and Vagrant or on VMware vSphere. For Cloud options, like Amazon AWS, Macaw provides Macaw AMI and as an appliance on Oracle Marketplace. Macaw also provides Cent-OS based installation scripts to be used for installation on various other cloud platform, like Google Cloud Platform, Microsoft Azure, etc.

The following terminology helps in understanding the important aspects of Macaw installation.

| **Type/VM** | **Purpose** | **Processes/Services** |
|---|---|---|
| Macaw Infrastructure Services<br><br>Platform VM | Macaw's infrastructure layer provides the DB access, indexing/search capabilities, and service communication infrastructure. These are mandatory core infra services. | Zookeeper<br>Kafka<br>MYSQL<br>Cassandra<br>Elasticsearch<br>Redis<br>Tomcat |
| Macaw Platform Services<br><br>Platform VM | Macaw platform layer includes the core essential foundation services. These platform services are shared across all tenants and provides critical services like Identity Management, Encryption/Decryption services for critical data, Provisioner for deploying microservices, Services Registry etc. These are mandatory core platform services. | Service Registry<br>Notification Manager<br>Identity Service<br>Service Provisioner |
| Macaw ADPM Services (Performance Monitoring)<br><br>Platform VM | Macaw platform provides performance monitoring capabilities for the deployed microservices. To enable the performance monitoring capability, optional performance monitoring services need to be installed. These services are optional and needs to be installed only if performance monitoring features are needed. | macaw-apm-agent<br>macaw-apm-collector |
| MACAW Services<br><br>Service VM | Macaw Services include various microservices and can be deployed on the Service Host. Services Hosts are logically grouped under an environment. You can have multiple environments with different services hosts and during provisioning user can select a specific environment for deployment. | Developed Microservices |

| | | |
|---|---|---|
| Macaw Tools<br><br>Platform VM | Macaw Tools are light weight containers providing the MDR (Meta Data Repository) and Docker Registry functionality. These are needed to publish and deploy user developed microservices.<br><br>Note: Docker Registry is installed from Central Docker and the version used is 2.3.1. For production deployments, it is highly recommended to deploy and configure tools on a separate host. | macaw-mdr<br>docker-registry |

## Chapter 6: Environment Preparation

There are two virtual machines in Macaw Platform - Macaw Platform VM and Macaw Services VM. While both the VMs can be installed on the same server for development or testing purposes, the general recommendation for production system is to install as two separate images.

### 1) VMware OVF

Two VMware OVFs - Macaw Platform VM and Macaw Service VM are provided. Up-to date information is available at VMWare OVF environment provision. The following are pre-requisites for installation.

| | **Macaw Platform VM** | **Macaw Services VM** |
|---|---|---|
| **Operating System** | CentOS 7.3.1611 (Core) | CentOS 7.3.1611 (Core) |
| **RAM** | 24 | 16 |
| **vCPU** | 8 | 8 |
| **NICs** | 1 | 1 |
| **IP Address** | IP, Subnet, Gateway,DNS | IP, Subnet, Gateway,DNS |
| **NTP** | NTP Server | NTP Server |
| **Hostname** | DNS Resolvable Hostname | DNS Resolvable Hostname |

| **OS** | **Tools Needed** |
|---|---|
| **Windows/Linux** | 7z Utility |
| **VMware vSphere** | 5.1 or above |
| **VMWare vSphere Client** | 5.1 or above |

Please contact support@macaw.io and get URLs for download of OVFs. Please refer to the documentation for up-to date information on VMware OVF environment preparation.

## 2) AWS

The subsequent instructions assume that AWS is setup with basic or necessary configurations like VPC, Networks, etc. Refer to the AWS link below on how to execute an AWS setup for a new account. You can get more information at **AWS Setup**. For provisioning of Macaw image and up-to date information please click **here.**

**Accessing Instance:** Macaw AMI instances are based on standard x86_64 CentOS7. The AMIs are programmed with CentOS user and the public key of the key pair you have used during the launch of the AMIs, is programmed for this user. You can use the private key of the pair to login.

ssh -i <path to the private key> macaw@<Platform Instance Public IP>
ssh -i <path to the private key> macaw@<Service Instance Public IP>
Refer to the below AWS links for accessing the instance using the key.
https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/putty.html
https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AccessingInstancesLinux.html

The Macaw AMIs are programmed with Macaw user. Password based SSH is enabled for this specific user. The default password is "macaw". It is strongly recommended that the password to be changed on all instances.

ssh macaw@<Platform Instance Public IP>
ssh macaw@<Service Instance Public IP>

You can use **putty** or any other standard windows based SSH client to access the Instance using SSH. From Linux, SSH client is standard utility and should be available on all flavors of Linux.

**macaw CLI Tool Installation:** Once you finished the launching of the required AMIs, you need to install Macaw CLI tool on the platform Instance. Execute the below command to install the Macaw CLI package.

sudo pip install <Location or HTTP link of macawcli tar Package>

When you registered for the Macaw Software download, the email confirmation will provide you a link to the Macaw CLI package. Download the package locally to platform instance or you can use the direct HTTP/HTTPS link in the command above.

## 3) Vagrant

Vagrant provides easy to configure, reproducible, and portable work environments built on top of industry-standard technology. Vagrant is controlled by a single consistent workflow to help maximize the productivity and flexibility of the user and their team. For more details on Vagrant Installation, refer to the Vagrant documentation.
Macaw Platform can be installed locally on a Windows desktop/MAC running Vagrant/Virtual Box. The documentation below assumes that a working

installation of Vagrant and Virtual box on the system are in place. The recommended version of Vagrant and Virtual Box is listed below.
Vagrant Download Link | Virtual Box Download Link
Note: To install Vagrant on Windows , openssh for Windows must also be installed as a prerequisite. For detailed and up-to date environment preparation and provision, please click **here.**

**4) Linux Host**

Macaw Platform is supported on generic 64 Bit Linux OS which supports Docker. The steps below provide an automated way of installing the necessary packages needed for Macaw Platform and also manual steps if required. This installed script is only supported on CentOS7 64-bit OS and version 3.10 or higher of the Linux kernel. For detailed up-to date information of environment provision on Linux Host can be found **here**.

## Chapter 7: Platform Installation

### 1. Platform Configuration

Macaw setup auto-generates the platform configuration. Once the platform configuration is auto-generated, if making any changes to the configuration, make sure to execute the command 'macaw sync' which keeps the platform configuration and provisioner configuration in sync. To understand details of the platform configuration, refer to the 'platform.README' file under the home directory.

### 2. Provisioner Configuration

Macaw-service-provisioner.properties is a JSON file which feeds Macaw service provisioner with environments where the user can provision their microservices. The JSON specification is an array of environment definitions. A default environment is mandated and created during the Macaw setup with inputs provided for service hosts.
For more up-to date information, please check **here**.

### 3. Environments

Macaw Platform segregates compute resources into environments. During the Service Blueprint deployment, you provision into an environment which is either backed by individual docker hosts or a Kubernetes cluster or swarm cluster. Macaw supports the following provisioning environments:

- Standalone Docker Hosts
- Swarm Cluster
- Kubernetes Cluster (Kubernetes v1.5.2)
-

### 4. Standalone Docker Hosts

A group of individual Docker hosts can be grouped together under an environment. The only criteria for grouping the nodes is: the storage definition defined under the environment should be accessible from all the nodes in the environment. For example, if you have defined a storage mount like below in the environment, then it is expected that all the nodes provide this storage.

```
"storage": [
    {
        "path": "/opt/java",
         "name": "JAVA_1.8",
         "read-write-mode": "ro"
    }
]
```

Though Macaw Platform doesn't enforce this, it is highly recommended to group hosts with the same Docker Version, Storage Mounts, CPU/Memory resources into an environment. As part of the Macaw setup, it is mandated to provide a single service host which is then used to create a default environment. Shown below is how compute resources are provided in the environment definition.

```
"machines": [
```

```
    {
        "pass-phrase": "",
        "ip": "macaw-s1.engr.cloudfabrix.com",
        "login": "macaw",
        "version": "7.1",
        "os": "centos"
         },
    {
        "pass-phrase": "",
        "ip": "macaw-s2.engr.cloudfabrix.com",
        "login": "macaw",
        "version": "7.1",
        "os": "centos"
    }
]
```

During the Macaw setup phase, password less SSH communication is established to these service endpoints from the platform host. Without this password-less SSH provisioning requests would fail.
For up-to date information, please refer **here**.

**5. Kubernetes**

Detailed explanation on Kubernetes and features supported by Macaw are covered under the advanced topics section. Macaw Platform provides the capability to provision services onto the Kubernetes cluster. Below are the requirements before a Kubernetes environment can be created.

1. Kubernetes Master
2. Credentials – User/password or Token based Authentication
3. Namespace – If no namespace is available, default can be provided. Any non-default namespace should be created upfront on the Kubernetes.
4. Mandatory PVCs and any optional PVCs required by services should be created up front on the K8 cluster

For details on how to enable Kubernetes environment and sample k8 environment JSON, please **here.**

## Chapter 8: Infrastructure Installation

Macaw Platform installation involves the installation of Macaw infrastructure and platform components. The sequence of commands below will install the Macaw Platform. The prerequisite for platform installation is the Macaw setup, completed in the previous step. Various Installation Commands are:

macaw infra install --tag <tag>
macaw apm install --tag <tag>
macaw platform dbinit --tag <tag>
macaw platform install --tag <tag>
macaw tools install --tag <tag> --service macaw-mdr
macaw tools install --tag 2.3.1 --service docker-registry

Stated **here** are additional details on the infrastructure, apm, platform, and tools installation sections of the Macaw Platform. While the link provides the full installation instructions, some of the highlights for quick reference are

1. As part of infrastructure installation, the following mandatory components will be installed:
Zookeeper
Elasticsearch
Mysql
Cassandra
HAProxy
Tomcat
Redis
2. APM install provides Performance Monitoring Agent and collector.
3. DBInit command initializes the database sections of MySQL and Cassandra.
4. Platform install provisions the following components:
Service Registry
Notification Manager
Service Provisioner
Identity Service
User Preferences
Console UI
5. As part of tools installation, MDR and docker-registry are provisioned. Docker registry is pulled from docker public repository and use 2.3.1 version.

Detailed installation instructions can be found **here**.

## Chapter 9: Developer SDK

Macaw Platform comes with an SDK (Software Development Kit) and eclipse plugin. Either the developer can use their favorite IDE or choose the Macaw provided eclipse plugin to develop Microservices. Macaw SDK is supported on the latest versions of CentOS, Ubuntu, Linux Mint, Mac OS, and Windows platforms. The Macaw team will provide the download link for the latest SDK. Once downloaded, it can be extracted to the developer's chosen folder. The class path to SDK location needs to be set up as environment variable (MACAW_SDK_HOME = <location of SDK>).

**SDK Organization**:

The SDK has the following directory structure:-

```
macaw-sdk
   |-- docs
   |-- quickstarts
   |-- runtime
   `-- Tools
```

From above:

- Docs – this directory houses all the bundled SDK documentation including this document.
- Quickstarts – this directory contains some example service projects which have been developed using this SDK.
- Runtime – this directory contains the Macaw platform runtime, which is required for compiling microservices.
- Tools – this directory has the tools, code-generator (in macaw-service-artifacts-generator subdirectory), and macawpublish (in macaw-publish-tools subdirectory). Code-generator is used to generate initial code for a microservice from a specified spec. macawpublish is used to publish the service to the docker registry.

For latest software requirements please click **here.**

### 1. Installation on Linux / Mac:

A Developer using the SDK must have the following software installed to facilitate developing microservices:-

- Oracle JDK, versions 8 and above
- JAVA_HOME env variable should be setup to point to this JDK installation.
- Apache Ant 1.9.x (and above) with Ant-contrib.
- ANT_HOME env variable should be setup to point to this Ant installation.

Publishing a microservice to a target Macaw Platform instance needs additional software installed on the machine:-

- Python 2.7 and above
- Docker version 1.11.x and above
- pip: package management system for Python
- requests package: See http://docs.python-requests.org/en/master/
- jsonschema package: See http://python-jsonschema.readthedocs.io/en/latest/
- paramiko==2.0.0 package: See http://www.paramiko.org/

- tabulate=0.7.7 package: See https://pypi.python.org/pypi/tabulate
- Note: The above Python packages can be installed via PIP, using the command

  ‘sudo pip install requests==2.11.1 jsonschema==2.5.1 paramiko==2.0.0 tabulate==0.7.7”

This SDK may be used to generate services and publish them to a Macaw platform installation (version 0.9.3). The Java-based microservices projects generated by the code-generator tool are Eclipse-based and can be imported into an Eclipse installation. There is no Eclipse version dependency in the project. It should work for all recent Eclipse versions; Eclipse Mars or a later version is recommended. However even if you are a IntelliJ or Netbeans user, the project can be imported into your favorite IDE with almost no extra effort.

**2. Installation on Windows PC**:

Follow the steps indexed here to install the necessary software dependencies for enabling Macaw SDK development on a Windows PC/Laptop. The steps also guide through the compiling/publishing/provisioning of an example service which comes as part of the SDK. The following are necessary software packages. Please ensure that these are installed on developer machine.

1. Python Installation – 2.7.x

Download Python from the below web site and follow through the regular installation steps. This would install Python 2.7.x on the local machine. The default path of the Python installation is C:\Python27.

Download Link: https://www.python.org/ftp/python/2.7.13/python-2.7.13.msi

2. Ant Installation – 1.9.X

Download the ant ZIP from the below location. Right click on the ZIP and extract to C:\.

This would create the folder C:\apache-ant-1.9.7

Download

Link: http://archive.apache.org/dist/ant/binaries/apache-ant-1.9.7-bin.zip

3. JDK Installation – 1.8.0.121

Download the JDK from the Oracle download site. Accept the license agreement and download the software. For Windows 64 bit, it is necessary to download, **Windows x64** version of the JDK.

Download

Link: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

For regular installation, the installer would install JDK in the below directory.

C:\Program Files\Java\jdk1.8.0_121

Once above installations are done, set up the environment as shown below.

**Setting up the environment**

1. Open a Windows Power Shell Window with Admin Permissions. By default, power shell opens in non-admin mode. Right click on the Power Shell icon and click on "Run As Administrator"
2. Execute the below 3 commands to set the environment. Once it is done, please close the Current Power Shell Window and Open a New Window.

```
[Environment]::SetEnvironmentVariable("Path",
"$env:Path;C:\Python27\;C:\Python27\Scripts\;C:\apache-ant-1.9.7\bin;
C:\Program Files\Java\jdk1.8.0_121\bin", "Machine")

[Environment]::SetEnvironmentVariable("ANT_HOME",
"C:\apache-ant-1.9.7", "Machine")

[Environment]::SetEnvironmentVariable("JAVA_HOME", "C:\Program
Files\Java\jdk1.8.0_121", "Machine")
```

**Note**: If any different versions of Python, JDK, ANT are installed, please change the paths accordingly.

**Verify the installation**:

Verify the installation with commands below. Open a new Power Shell Window with Admin Permissions like before, to do the verification below. Path/Environment changes will not be applicable to the current running the Power Shell window.

```
PS C:\WINDOWS\system32> ant -version
Apache Ant(TM) version 1.9.7 compiled on April 9 2016

PS C:\WINDOWS\system32>

PS C:\WINDOWS\system32> java -version
java version "1.8.0_121"
Java(TM) SE Runtime Environment (build 1.8.0_121-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)

PS C:\WINDOWS\system32>

PS C:\WINDOWS\system32> python --version
Python 2.7.13

PS C:\WINDOWS\system32>
```

- Install Python modules – Open Power Shell window with Admin Permissions and run the below command.

```
python -m pip install requests==2.11.1 paramiko==2.0.0
jsonschema==2.5.1 tabulate==0.7.7
```

**SDK Download and Setup**

**SDK Download** – Download the Macaw SDK (ZIP Version). The email invitation/guidelines would have the download link.
1. Unzip the SDK into the project directory. For example: C:\Users\foobar\Documents\foobar\. The SDK would be unzipped and show a folder structure C:\Users\foobar\Documents\foobar\macaw-sdk-<version>
2. Macaw SDK requires 2 environment variables to be able locate and identify the SDK run time libraries and locate the microservices. Execute the commands below in a Power Shell window opened with admin permissions. Adjust the paths accordingly to the directory paths.

```
[Environment]::SetEnvironmentVariable("MACAW_SDK_HOME",
"$ENV:HOMEDRIVE$ENV:HOMEPATH\Documents\foobar\macaw-sdk-<ver
sion>", "User")
[Environment]::SetEnvironmentVariable("MACAW_SERVICES_HOME",
"$ENV:HOMEDRIVE$ENV:HOMEPATH\Documents\foobar", "User")
```

From here onwards, $ENV:MACAW_SERVICE_HOME would be the location, under which macawtool would try to locate the services based on the name of the service folder. Note that this is not a strict limitation. If the structure below is followed, service artifacts can be easily published by the Macaw tool, by just specifying the service directory name. If this procedure is not followed, then the absolute path to the service must be specified. For details, read the Macaw Tool documentation provided at the end.

$ENV:MACAW_SERVICE_HOME/service1/api

$ENV:MACAW_SERVICE_HOME/service1/impl

$ENV:MACAW_SERVICE_HOME/service2/api

$ENV:MACAW_SERVICE_HOME/service2/impl

Note: Instead of HOMEDRIVE, HOMEPATH variables, you can provide the full path to your SDK and microservices directory.

**Macaw Tool Setup – For Publishing**

The below steps can only be performed, with a working platform instance or access to an existing platform installation.

- macaw publish tool is already bundled with the SDK. It is located at $ENV:MACAW_SDK_HOME\tools\macaw-publish-tools\bin. It is python script and can be executed using the python interpreter.
- macaw publish tool relies on a configuration file called "macawpublish.globals". The tool looks for this file under multiple locations with below preference.
- If an environment variable, MACAW_MDR_GLOBALS_FILE is set and pointing to a file, it is used.
- Else, it looks for "macawpublish.globals" in the user home directory which is typically C:\Users\<userid>.

- Else, it uses the default file which is shipped with the macaw SDK. Note that the default file doesn't provide any default configuration, other than the documentation.
- Assuming that the complete setup of MDR/Docker Pair as part of the installation, follow the below instructions.
- Download/Create the "macawpublish.globals" from the platform VM Instance to the HOME directory C:\Users\<userid>.
- On the platform VM, find the "macawpublish.globals" at the location**/opt/macaw-config/macaw-tools/macawpublish.globals**
- Once copied the "macawpublish.globals", verify the connectivity to MDR/Docker Pair, using the below command.

  ```
  cd $ENV:MACAW_SDK_HOME\tools\macaw-publish-tools\bin
  python macawpublish verify
  ```

With this the PC/Laptop setup is finished for development environment.

## Chapter 10: Developing Microservices

The following section will help you in developing, publishing, and deploying custom Microservices on the Macaw platform.

**Developing a Microservice**:
The Macaw Platform SDK bundles a code-generator tool for rapid development of custom Microservices on the Macaw Platform. The SDK support Java, JavaScript, NodeJS, and Python. More language features will be added at a later date.

The process for generating a Microservice is four Step process as defined below:
Step 1: Define a service API descriptor for the Microservice.
Step 2: Generate Microservice artifacts using code-generator tools.
Step 3: Provide Implementation using your favorite IDE. Macaw Platform provides an Eclipse plugin to help with development and publishing of Microservice.
Step 4: Compile Microservice API and Implementation.

Step 1: Define Service API
1. List out all the API methods that need to be exposed by the service. Detail the name of the service, its inputs, and outputs.
2. Identify the inputs and outputs entity types using either supported YANG notation or JSON notation.
For example, on sample API descriptor, please refer to the examples provided as part of the SDK.

Step 2: Generate Microservice Artifacts:
The Macaw Platform SDK bundles a code generator for rapid development of custom microservices. The project generation properties are defined in the file service-artefacts-gen.properties. These properties are used by the tool to generate the necessary project stub. Please refer to the Developer Guide for setting up of MACAW_SDK_HOME on how and where to execute 'run command' to generate project artifacts.

Step 3: Implement Business Logic:
The step above generated project stubs which can be imported as an eclipse project or any of your preferred IDE and provide implementation as per required business logic. The artifacts can be categorized into public API and service implementation. The public API(jar) contains the necessary artifacts that the service publisher/developer can handover/publish to consumers of that service.
The public APIs generated are under ${artefacts.output.dir}/<service name>/API directory. The service implementation artifacts are meant for the developer to implement the APIs exposed by the service. The stubs for the service implementation artifacts are generated under ${artefacts.output.dir}/<service-name>/impl directory.

Step 4: Compile Microservice API and Implementation:
The generated API jar of the service now needs to be copied over to the Microservice implementation as dependency. This can be done by running 'ant

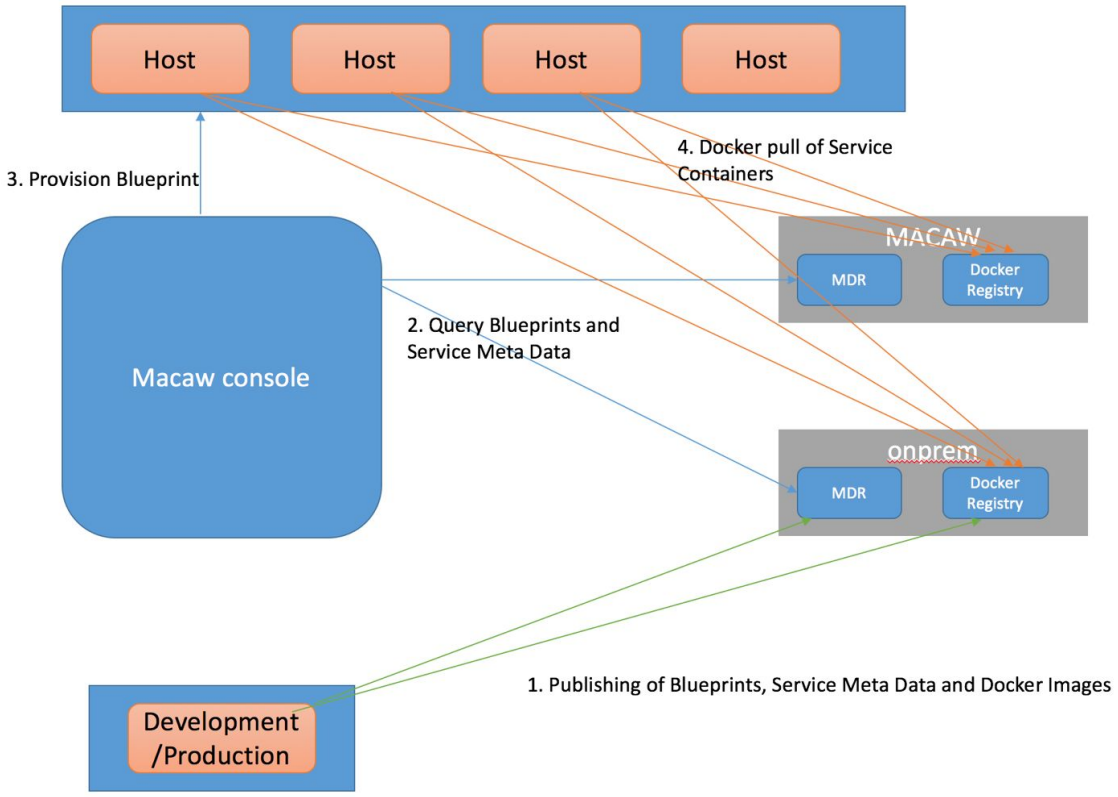
clean deploy’ in <service-name>/api directory. The Microservice implementation can be compiled by running ‘ant clean deploy’ in <service-name>impl directory. The above compiles both public API and service implementations. The service is now ready to be published to the target Macaw Platform and is explained in next section.

## Chapter 11: Publishing a Microservice

Macaw Platform runs the microservices as a Docker container. Each service that is generated by the code-generator tool, as above, will be provided with an auto-generated Dockerfile and it resides at <service-name>/impl/etc/docker/Dockerfile. Typically, the developer does not have to change any of the content of the file.

### MDR and Docker Registry

MDR (Meta Data Repository) and Docker Registry together play a key role in the Macaw Platform. MDR holds the Service Blueprints, Meta Data information, available Docker tags for a specific service. Docker Registry holds the service container images. In Macaw Platform, these two components go together and provide the end-to-end functionality of Service Provisioning.




MDR and Docker Registry information is provided as a configuration to Macaw Service Provisioner. Macaw Console interacts with the service provisioner and provides the ability to query a specific MDR/Docker Registry pair for available Blueprints and Services. For more information on MDR and Docker registry, click **here.**

### 1. Publishing metadata and docker image of the Microservice

The Macaw Platform SDK bundles a tool called macawpublish for publishing of custom microservices on Macaw Platform. Once the development of the Microservice is finished, the service can be published to Macaw Platform. However, all Microservices on Macaw Platform must be published as dockerized containers. The macaw publish tool is in the macaw-publish-tools/bin subdirectory.

The following command will publish the docker image for the Microservice,

./macawpublish service - -tag <<Specify your tag name here>>
<<Specify service>> for example for calculator Microservice, it will be
./macawpublish service - - tag <<specify your tag here>> calculator

Macawpublish supports the following command line options

--description, Used to specify tag description
--labels, Used to specify tag labels (comma separated if multiple labels)
--skip, Used to skip tag creation, if specified

For the label option, if any tag labels are not specified, then the mdr.tag.label.default from macaw-sdk/tools/macaw-publish-tools/macawpublish.globals file is used as a default label. If any tag labels (comma separated) are specified, then the tables are checked for validity against those specified in mdr.tag.lable.allowed from macaw-sdk/tools/macaw-publish-tools/macawpublish.globals.

**2. Publishing Microservice Blueprint**

A Microservice blueprint is an entity which can be used to define a provisioning specification for a set of Microservices. More often than that, customers have a requirement to provision a set of related Microservices collectively. A Microservice blueprint is the specification which fulfills that need. The Microservice blueprint must confirm the schema defined at macaw-publish-tools/bin/ServiceBlueprint.schema. Please go through this schema document to understand what each attribute in the blueprint stands for and where the possible values for any attribute are.

Now publish the blueprint after filling necessary values. The command is
./macawpulish blueprint <<Name of the blueprint in the service-blueprints directory>>

For publishing web application blue print, check details **here.**

**3. Publishing Web Applications**.

In Macaw platform, web applications (war files) can be packaged as standard Docker containers and deployed into existing Tomcat instances running as Macaw UI Pairs. Refer to Macaw UI pairs documentation on how to create and provision UI Pairs. For publishing web applications please click **here.**

**Note**: For up-to date and detailed instructions on publishing a microservice, please click **here.**

**4. Remote Publishing**

Macawpublish tool requires the presence of Docker on the local development machine to be able to push service container images to the Docker Registry. This would also require configuration of Docker Daemon on local host w.r.t certificates. This restriction can be avoided on the local development machine by enabling remote publishing in macawpublish tool. With remote publishing, none of the instructions and functionality of the macawpublish tool

would change, except that now the service artifacts necessary for the containerization of the service are uploaded to a remote build server and the docker build/tag/publish operations are executed on the remote server. This avoids having Docker install on your local machine. The remote build server should be configured with proper certificates and access to the on-prem registry installed as part of the macaw tools. Refer to MDR/Docker Section above for details on how to achieve this.
For details on remote publishing and how to enable it, please refer documentation **here**.

## Chapter 12: Deploy and Manage Microservice using Macaw Console

Macaw Platform provides Macaw Console (sometimes also referred to as Macaw DevOps Console) to deploy and invoke Microservices. Macaw console is a UI console that provides a diverse administrative, life-cycle, deployable, and manageable interface for microservices. Macaw DevOps Console provides the following functionalities:

1. Dashboard:

Once user logged in, the user is taken to default dashboard page which provides the following functionality and information:

- Total counts of service groups, platform essential service clusters, web applications, etc.
- Doughnut graphs depicting the status of all instances under platform service instances, web applications, etc.
- List of all service groups with environment name/type, available instance of cluster etc.

For more details with visual screens, please click **here.**

2. Service Manager:

The service manager option under Macaw console provides a comprehensive functionality pertaining to Macaw Microservices. For a more detailed explanation including visual screens, please click here.

3. Administration:

The Administration UI provides administrative functionalities to platform administrators like Tenant Management, Organization Management, Project Management, Managing Tenant Administrators, Platform Administrators, etc.
For more details with visual screens, please click **here.**

4. Service Events:

Events are captured whenever any RPC of a service is executed. ServiceEvents presents the metrics of these events, along with the details of failed/slow Invocations. There is separate page explaining Events under Monitoring section.
For more details and visual screens, please click here.

## Chapter 13: Analytics, Logs and Events

### 1. Service Analytics

Service Analytics is the discovery, interpretation, and communication of meaningful patterns in the data related to the Macaw-based microservices. The purpose of these analytics is to address the following:

- Detect abnormal conditions to generate alerts and/or to take corrective actions. These include conditions such as abnormal error rate, latency, resource usage, etc.
- Track usage metrics to optimize/limit resource utilization, SLA enforcement, and billing.
- Adaptive optimization/tuning of configurable parameters to maintain optimal service performance.
- Predict future growth in usage and enable resource planning.

The data used in the analytics include:

- Service configurations and blueprints
- Historical logs
- Deployment logs
- Operational logs
- Runtime logs (including application specific logs)
- Audit logs
- Real time / Near-realtime data
- Service invocation events
- Metrics specific to applications (service implementations)
- Deep metrics generated by language-specific APM instrumentation (such as JVM metrics in case of Java)

### 2. Service Logs

Macaw provides support for service implementers to log application specific runtime conditions while servicing requests. Macaw runtime automatically injects additional attributes along with each long entry such as the following to enable searching:

- class: name of the class
- method: method from which msg was logged
- thread: thread id
- level: logging level
- message: actual log message
- exception: exception content
- logger: name of the logger
- host: hostname
- clusterId : id of the service cluster
- instanceId : id of the service instance

These log entries are automatically indexed in Elasticsearch. These log entries can be accessed via Macaw Console by drilling-down from respective services. Alternately, these log entries can be searched and accessed via standard Elasticsearch front ends. In addition, Macaw provides support for generating open-tracing compliant log messages.

### 3. Service Events

Macaw runtime has an inbuilt mechanism for tracking every invocation (request/response) of RPCs supported by a service. These service events are used for computing various metrics such as response time, throughput, error rate, etc. These metrics are further aggregated to provide client-wise usage, service-instance-wise throughput, cluster-wise load, and so on. Additionally, Macaw runtime also tracks the asynchronous notification messages that are exchanged among services collaborating in application specific interactions.

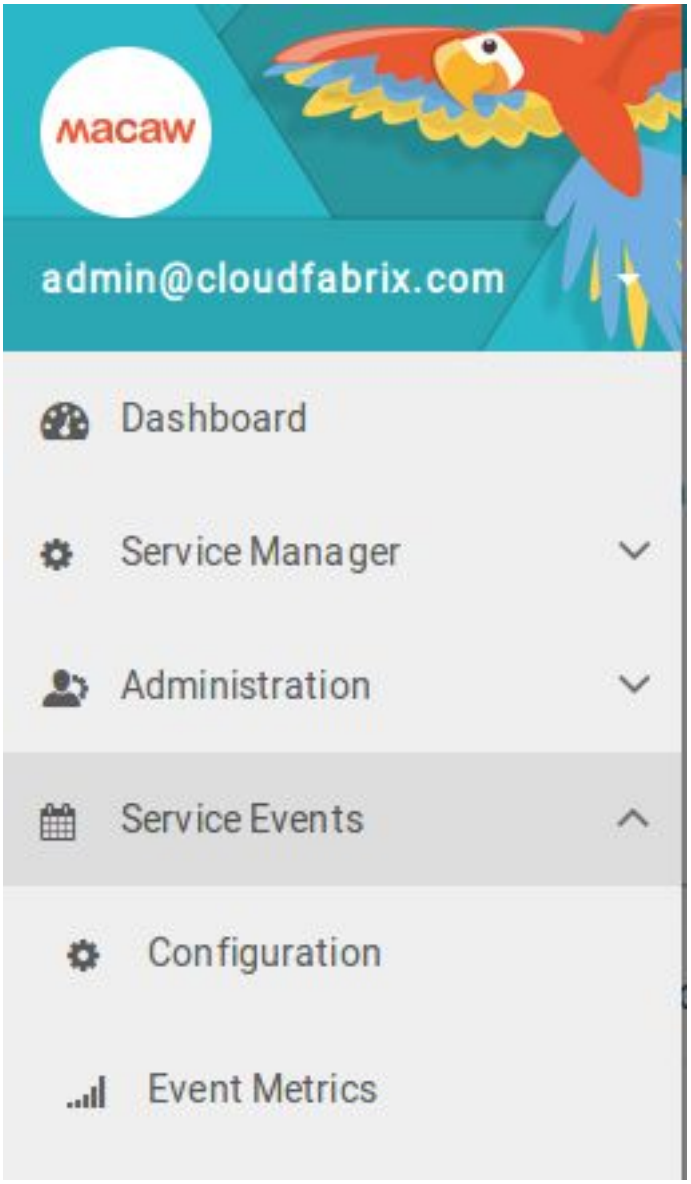
Business applications built using multiple microservices will typically involve transactions that span across multiple services. In each such business transaction, the collaborating services invoke one another's RPCs and send a sync notification messages to perform the underlying business activity. Macaw runtime assigns each such business transaction a correlation-id. All service events that are generating during the execution of the transaction are associated with the correlation-id. Taking advantage of this correlation-id, Macaw provides the business transaction level view of the service interactions along with the application specific logs associated with that transaction. For troubleshooting problems, this transaction level view provides a powerful means quickly identifying the RPC that is the source of the problem and determine the root cause.

**4. Application Custom Metrics**

Macaw provides APIs and runtime support for capturing custom metrics specific to an application. Custom measurements using counters, timers, gauges, meters, and histograms can be captured using the APIs provided. Macaw runtime automatically stores these metrics efficiently and provides APIs for metrics clients to fetch and display/process them.

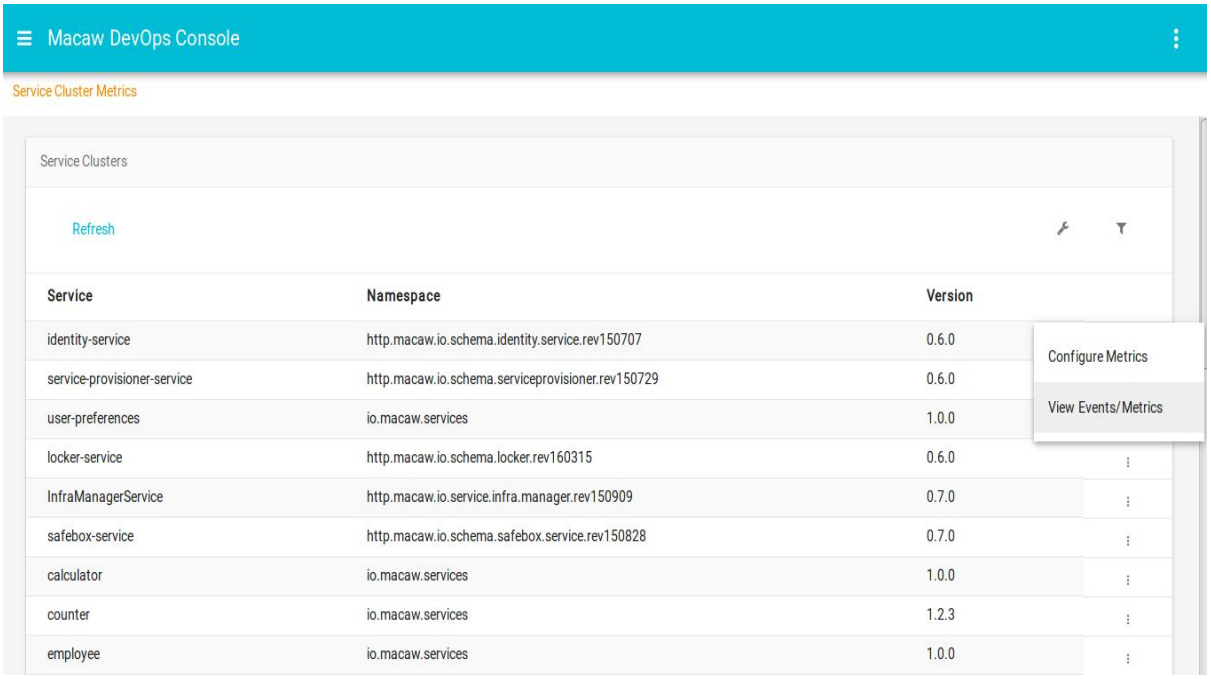
**5. Visualizing Metrics**

Macaw Console provides metrics dashboards with searching capability to fetch metrics for any RPC or service and visualize the data in time series charts and reports of the metrics. RPCs having high latency or containing errors are shown in the reports. There is a provision to drill-down from service-level metrics to RPC-level metrics and from there further down to invocations that are slow or failed. From a slow/failed invocation, the detailed logs specific to that invocation can be accessed. Additionally, from slow/failed invocation, one can access the collaboration view showing the interactions among all RPCs participating in that transaction.

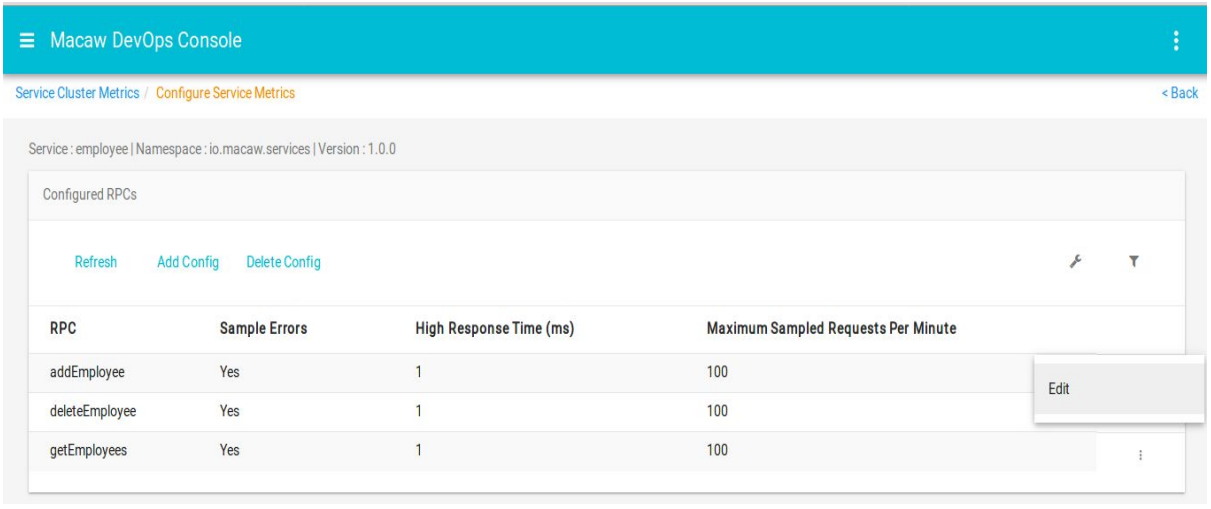



**Configuration**:
The Configuration section contains the list of all the services with name, namespace, version. Under each service,

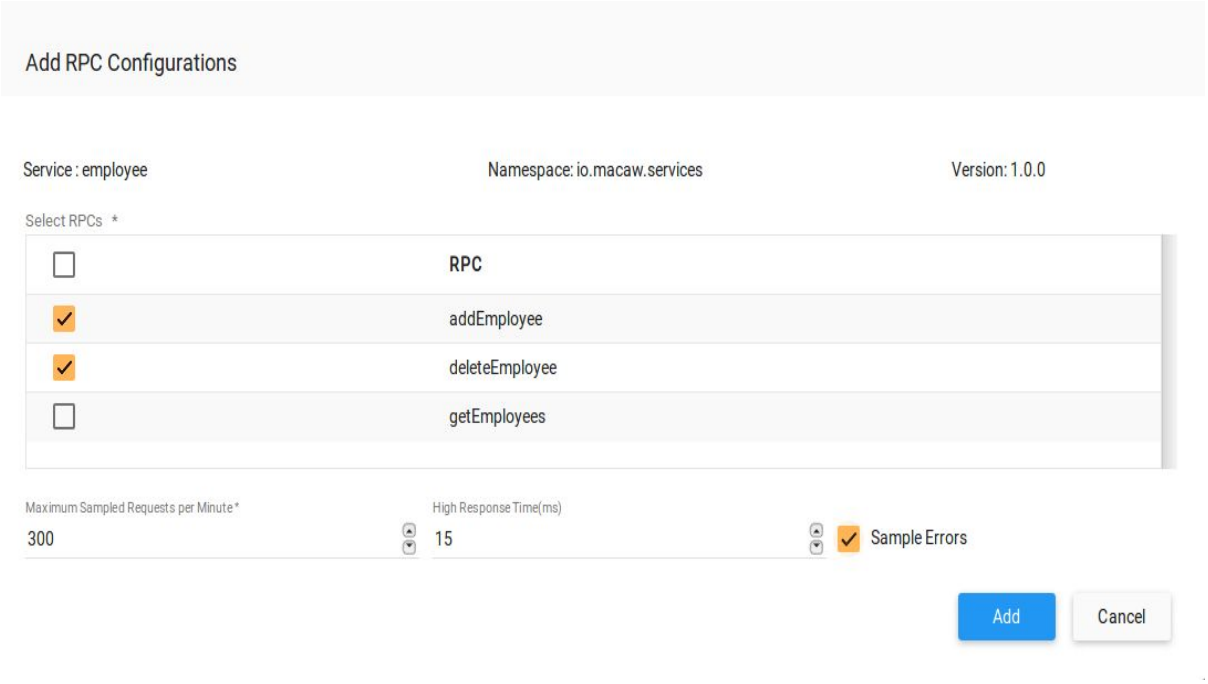
- There is the configuration to get the details of failed/slow Invocations of the service RPCs.
- View Events/Metrics of all the executed RPCs of service.




Configure Metrics:




'AddConfig' link can be used to add configuration for any RPC of the service

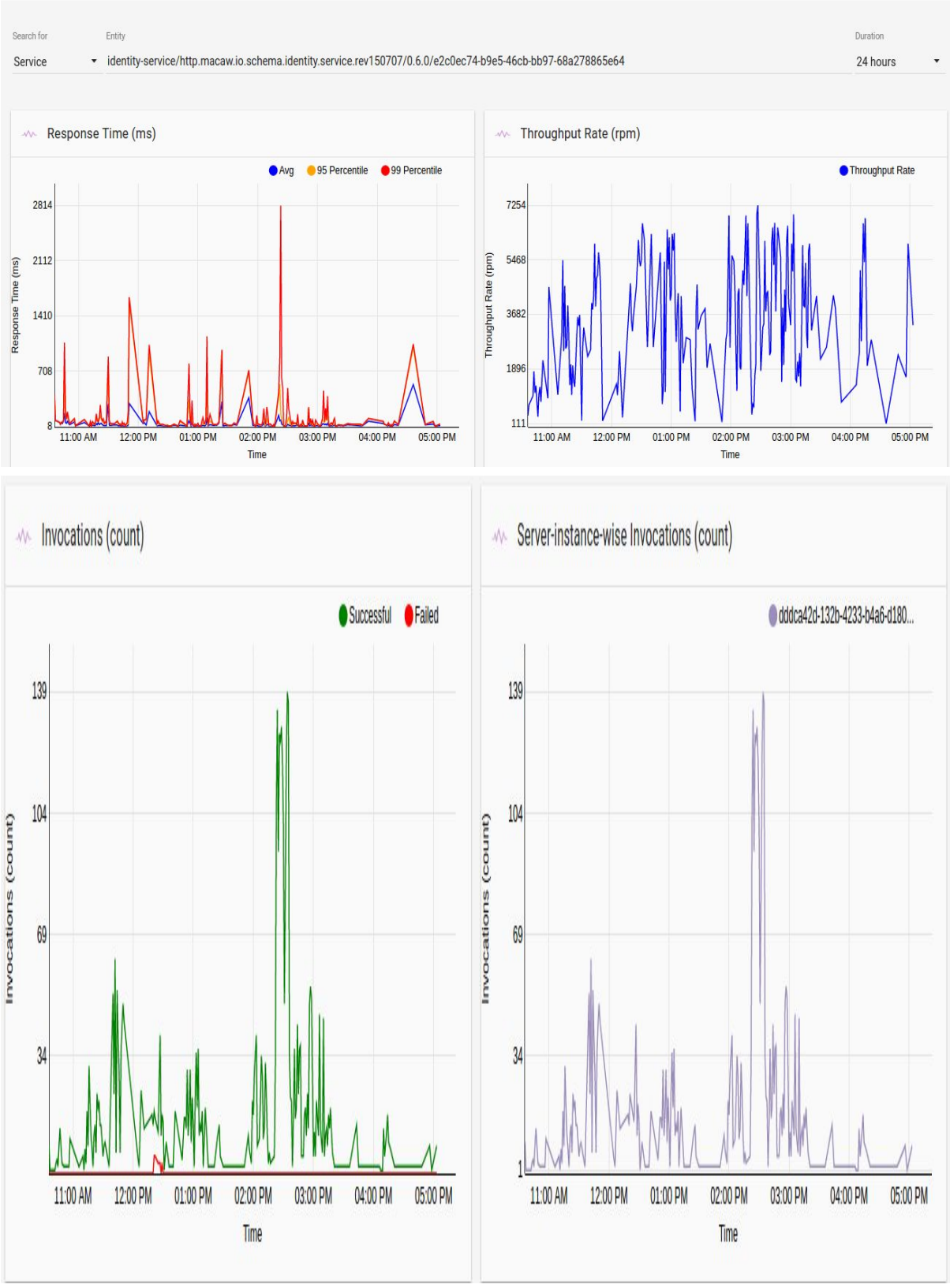



Add RPC configuration form inputs:

- Select RPCs from the list, for which the configuration has to be done.
- 'High Response Time' field is used to get the details of slow invocations. If the RPC takes more time to execute than the value given here, then it is considered as a Slow Invocation and its details are captured.
- 'Sample Errors' field is used to enable capturing the details of failed invocations. If the RPC execution fails, then the details of this invocation are captured.
- 'Maximum Sampled Requests per Minute' represents, maximum number of failed/slow invocations of the RPC(per minute), for which details have to be captured. If there are more invocations that are slow/failed, than the number specified here, then the details of remaining invocations are discarded.

Added configurations can be deleted/modified.

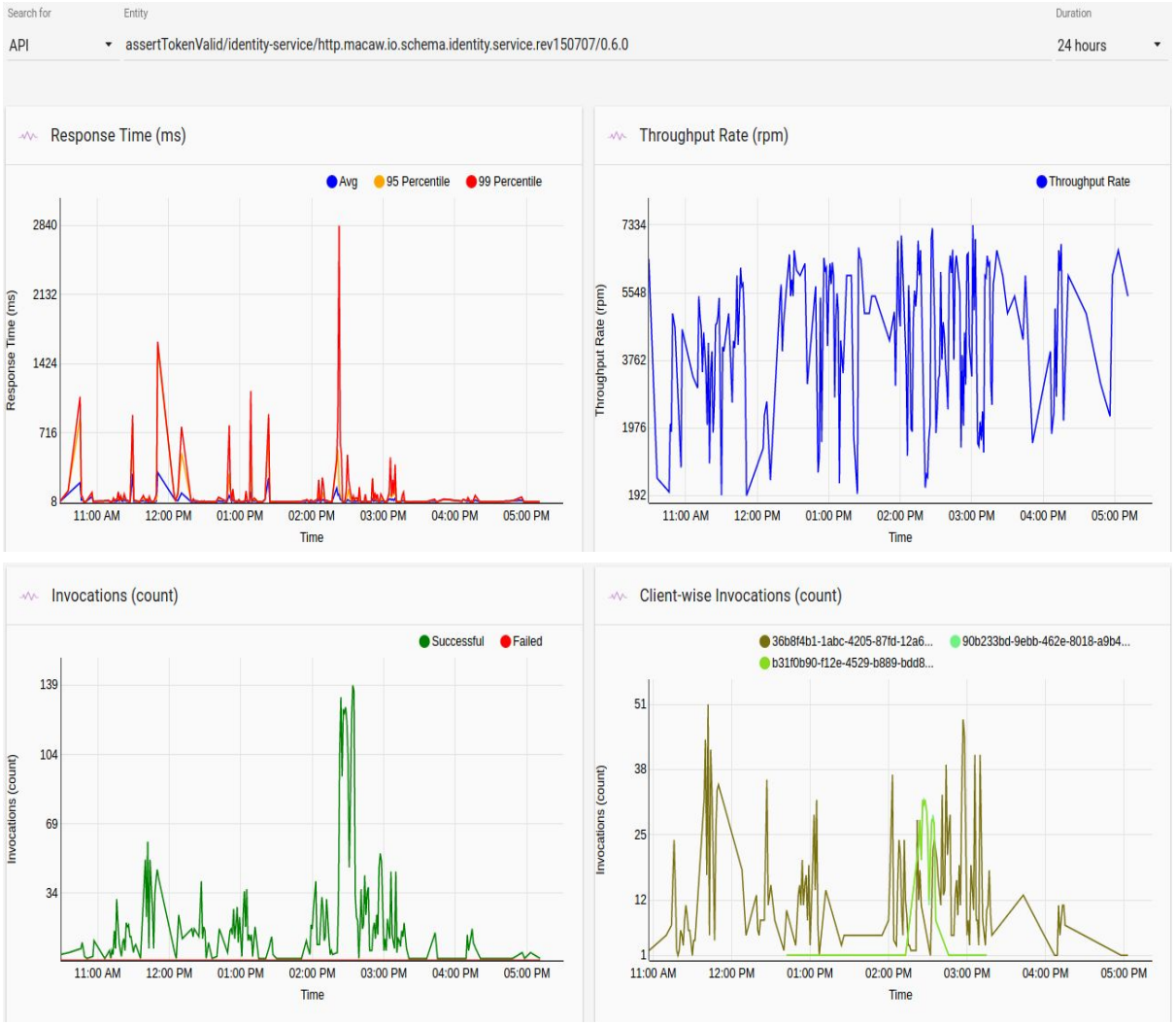
**View Events/Metrics**:
Using this dashboard, the metrics of specific services and RPCs can be fetched and visualized.
Here is the dashboard of a sample service:


Search for
Service
Entity
identity-service/http.macaw.io.schema.identity.service.rev150707/0.6.0/e2c0ec74-b9e5-46cb-bb97-68a278865e64
Duration
24 hours
Response Time (ms)
Avg
95 Percentile
99 Percentile
Response Time (ms)
2814
2112
1410
708
8
11:00 AM
12:00 PM
01:00 PM
02:00 PM
03:00 PM
04:00 PM
05:00 PM
Time
Throughput Rate (rpm)
Throughput Rate
Throughput Rate (rpm)
7254
5468
3682
1896
111
Time
Invocations (count)
Successful
Failed
Invocations (count)
139
104
69
34
Time
Server-instance-wise Invocations (count)
dddca42d-132b-4233-b4a6-d180...
Invocations (count)
139
104
69
34
1
Time

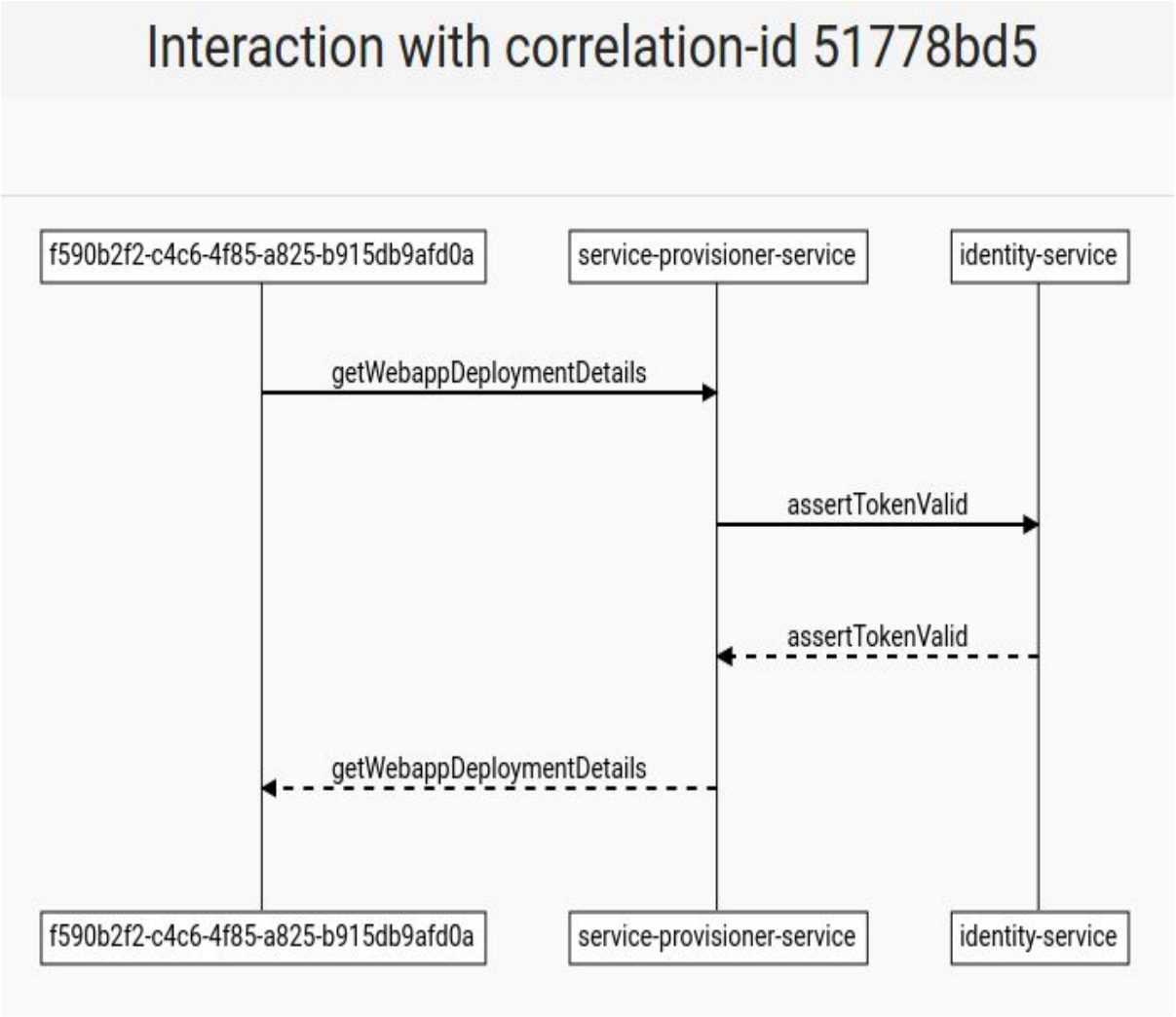
Invocation Metrics

| Service Name | Service Version | Method Name | Config Enabled | Invocation Count | Throughput (rpm) | Error Count | Error Rate % | Min (ms) | Avg (ms) | Max (ms) |
|---|---|---|---|---|---|---|---|---|---|---|
| identity-service | 0.6.0 | activateOrganization | Yes | 1 | 2068.97 | 1 | 100 | 29 | 29 | |
| identity-service | 0.6.0 | activateProject | Yes | 1 | 3157.89 | 1 | 100 | 19 | 19 | |
| identity-service | 0.6.0 | activateTenant | Yes | 2 | 2553.19 | 2 | 100 | 20 | 23.5 | |
| identity-service | 0.6.0 | activateUser | Yes | 1 | 2857.14 | 1 | 100 | 21 | 21 | |

Here is the dashboard of a sample API - accessed by drilling down from the service-level dashboard:




Drill-down to Interactions view of selected slow invocation




View application logs of selected invocation

## Chapter 14: Cloud native journey made simple with Macaw and Kubernetes

Cloud-native is an approach to building and running applications that fully exploit the advantages of the cloud computing delivery model, namely agility and innovation. A commonality among cloud native applications that most customers wants to adopt is that they are often built using microservices architecture, as a suite of independently deployable, small, and modular services that communicate through well-defined interfaces called APIs.

In the current Cloud Native vision, these independent Microservices are supplied, deployed, and run in something called a "container". Containers are lightweight, portable compute instances and offer better density per server than VMs. The original and most popular container technology was Docker but other technologies such as CoreOS are also widely used.

To help manage their container deployments across private and public clouds, Enterprises must rely on technologies like Kubernetes, the open-source container orchestration system popularized by Google.

In summary, enterprises seeking to commence a cloud native journey should be ideally equipped with the following technology stack:

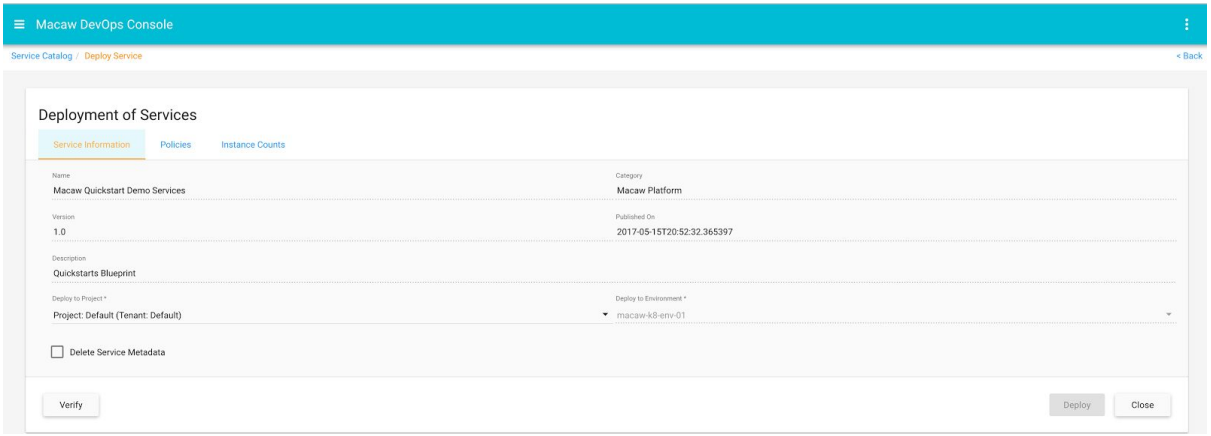
- Programming model to define and author Microservices applications
- Pipeline to automatically package and deploy Microservices as containers
- Runtime for Microservices applications
- Management and Governance of Microservices applications
- Container management and automation technology like Kubernetes
- On-demand provisioning of hosts in the cloud of their choice to run containers or use Containers as a Service (Infrastructure Service)

As shown above, building the technology stack involves substantial depth and complexity, hence one of the reasons we are not witnessing large scale Microservices applications in hybrid cloud environments. To address this problem and smoothen the transition process for enterprises, Macaw has integrated all the required capabilities and is offered as a turn-key platform. With Macaw, enterprises can deploy the complete stack at one go and start their cloud native journey. It is programming language agnostic as well as cloud agnostic, completely avoiding technology or cloud lock-in.

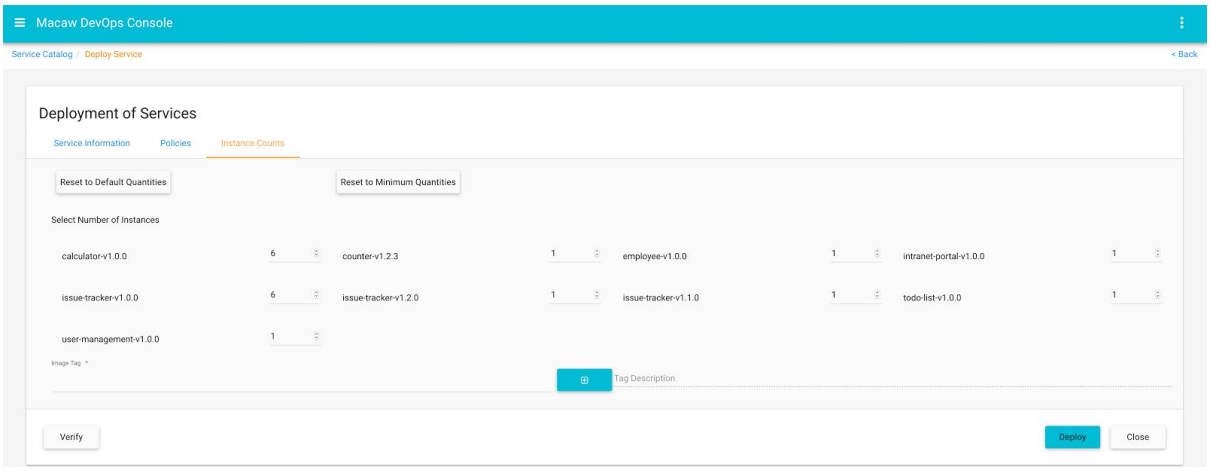
The Macaw Platform supports microservice provisioning to environments backed by Kubernetes cluster. Macaw Service provisioner leverages the Kubernetes deployment declaratives to orchestrate the seamless creation and management of service pods. Here are the high level features supported by Macaw platform release 0.9.4 w.r.t Kubernetes environment. The following are features currently supported by Macaw platform w.r.t. Kubernetes environment.

1. Ability to provision Service Blueprints into environments backed by the Kubernetes cluster. Macaw Service Provisioner interacts with the Kubernetes master either with basic-auth or token based authentication.

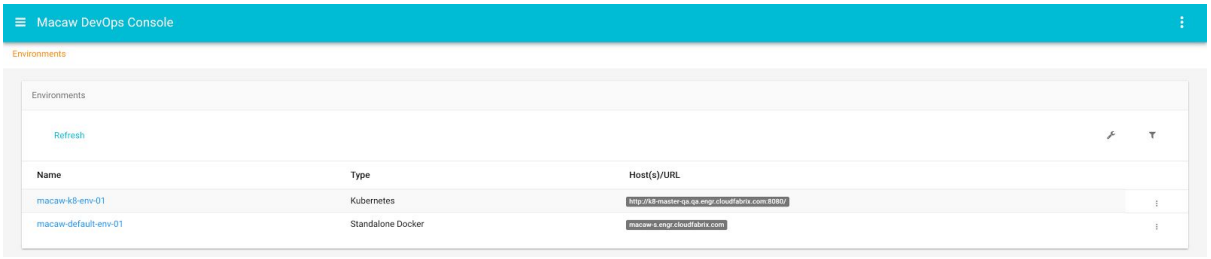
2. Support for Kubernetes virtual clusters (Namespaces). User can create multiple environments in Macaw pointing to the same Kubernetes cluster but different namespaces.

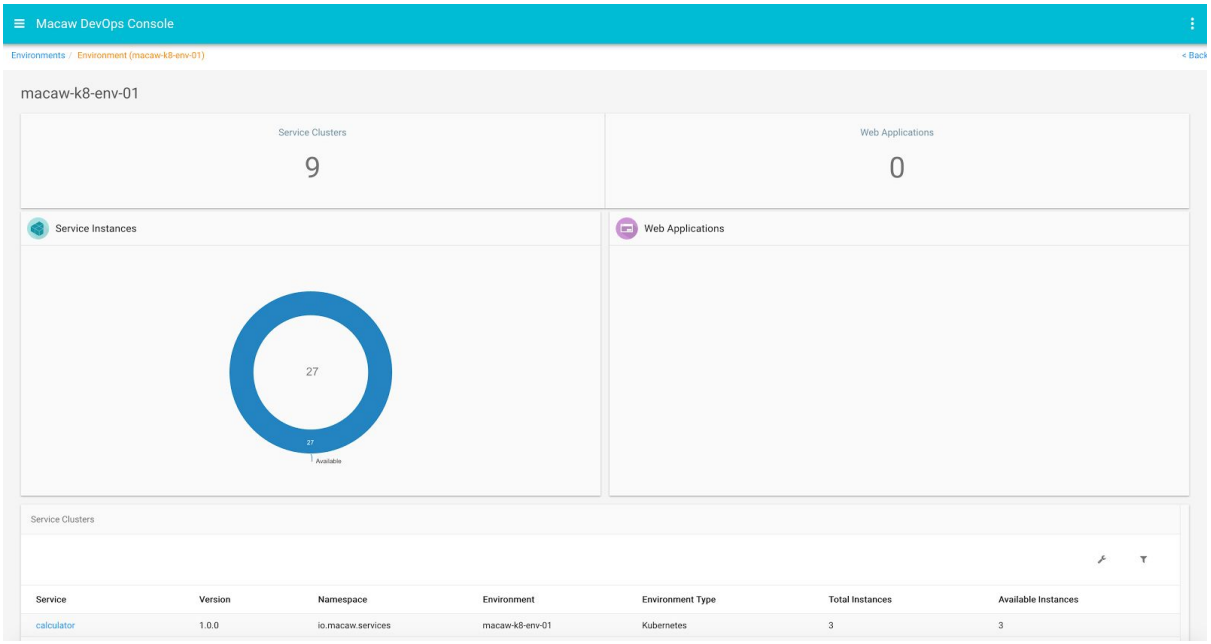



3. Support for multiple instances in a service cluster. This is achieved seamlessly through the Kubernetes deployment declarative.

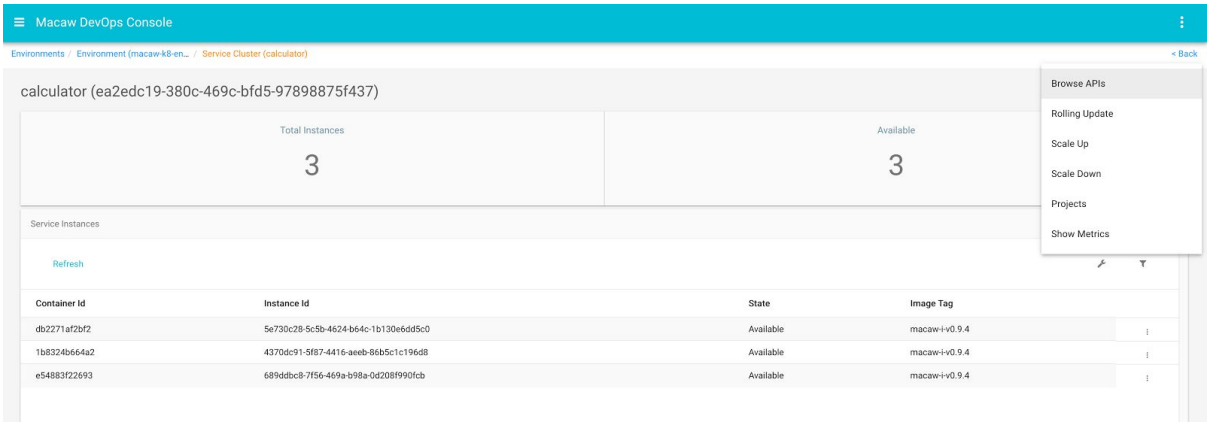


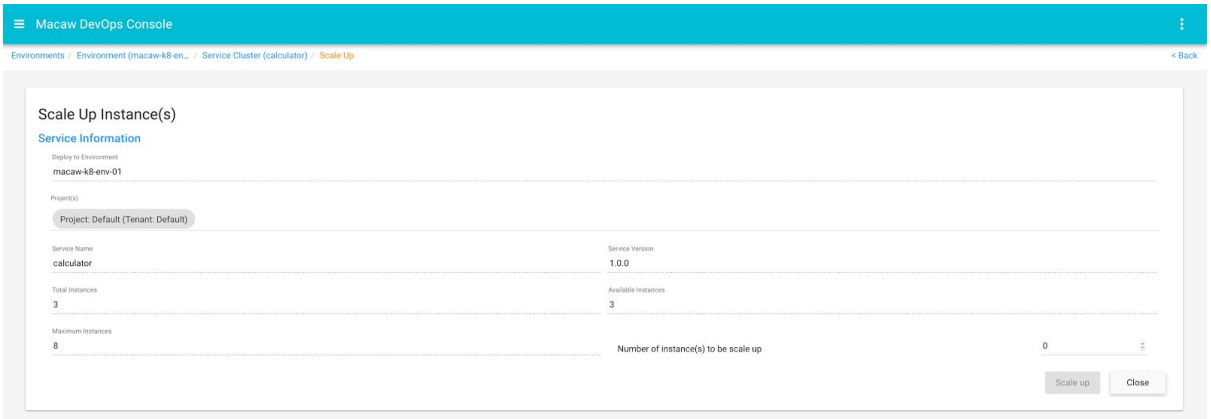

4. Ability to view Service clusters/pods in Macaw Console.

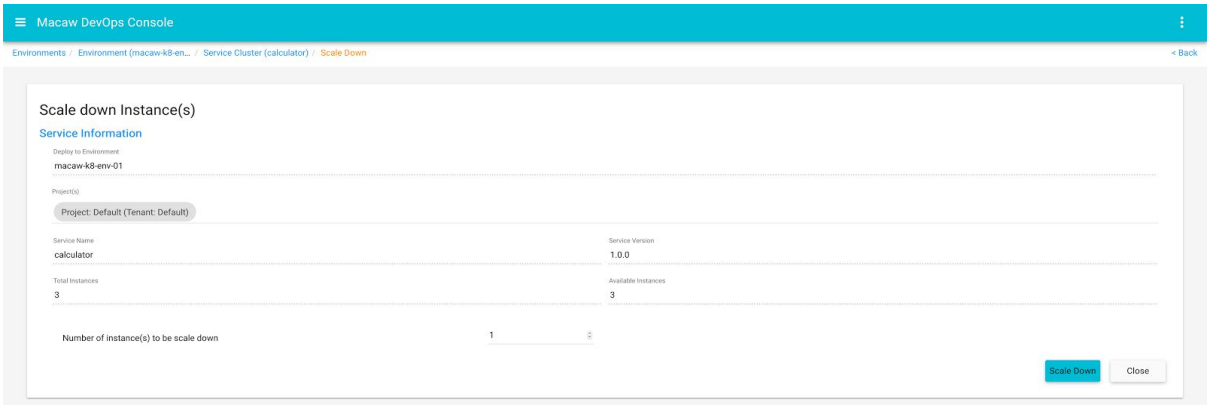




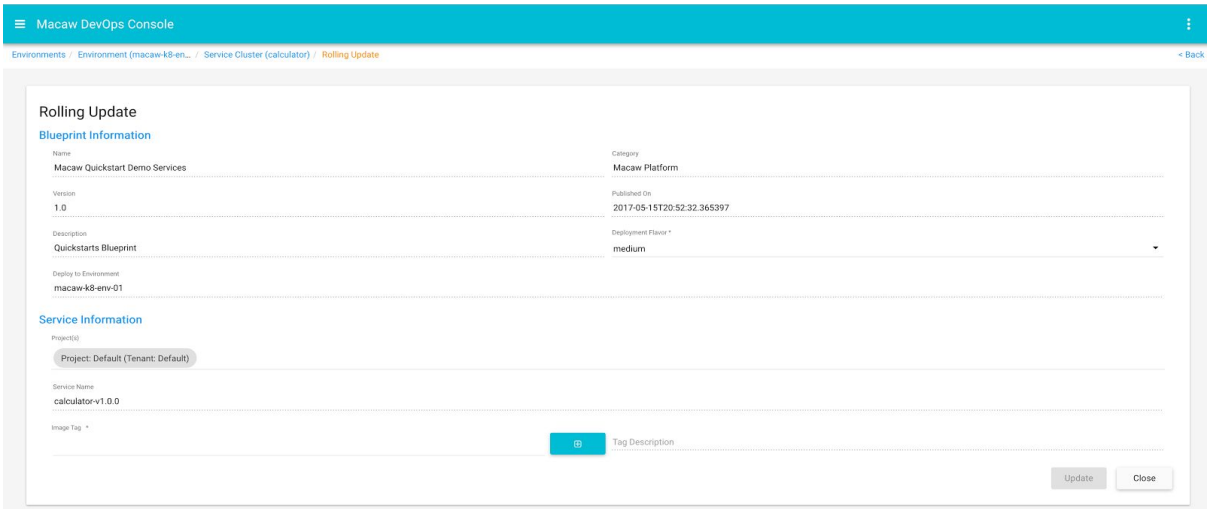

5. Ability to scale up and scale down a service cluster.






6. Rolling update of a service cluster with a new image.




7. Implicit support for Macaw features like Metrics, Events, Elastic Search Log integration.

## Chapter 15: Recipes

**This chapter contains recipes to help developers.**

### Recipe 1: Initialize Microservice Start/Stop

Typically, when your Microservice is started, you as a developer would want to execute code which for example creates database connection pools, initializes caches, etc. and then tear it down when the Microservice stops. In order to facilitate this, Macaw provides some service lifecycle hooks, where custom code can be plugged in. They are :-

- initialize() – executes when the service initializes.
- start() – executes when the service starts up.
- stop() – executes right before the service stops.

The exact signatures for these methods are as shown below.

```
public void initialize(final com.cfx.service.api.config.Configuration config)
        throws com.cfx.service.api.ServiceException;

public void start(com.cfx.service.api.StartContext startContext)
        throws com.cfx.service.api.ServiceException;

public void stop(com.cfx.service.api.StopContext stopContext)
        throws com.cfx.service.api.ServiceException;
```

| | |
|---|---|
| NOTE | For a concrete example of usage of these methods, please refer to <SDK InstalledDirectory>/quickstarts/todo-list/impl/src/main/java/com/macaw/quickstart/todo/impl/TodoList.java. |

**Recipe 2: Adding 3rd party libraries**

A non-trivial service will perform operations which involve usage of third party libraries. These libraries need to be bundled with the service. Macaw Platform supports this functionality out-of-the-box.

Any such third party jars must be kept within ,the *impl/src/main/lib* folder of your service implementation. The service impl build script is pre-configured to pickup any jars in this folder & make it available in the compile time classpath of the project as well as bundle it appropriately in the service archive.

For a concrete example of usage of third party libraries, please look at the *todo-list* sample service. In this service, we use a third party Cassandra database driver to execute queries against the service's database. This driver and all its dependencies (jars) are kept in the *quickstarts/todo-list/impl/src/main/lib* folder. The ant build script (*impl/build.xml*) adds any jars in lib to the classpath for compilation as well as bundles them in the service distribution that it creates.

| | |
|---|---|
| IMPORTANT | The macaw-service-api and macaw-service-client libraries, which provide access to Macaw specific interfaces, are automatically made available in the runtime of the services and MUST NOT be included in the *impl/src/main/lib* folder of your service. Including them here can cause classloading issues during runtime. |
| NOTE | Macaw service runtime currently exposes SLF4J API library/interfaces for use within the service implementations. As such the service developer is not expected to package this library/jar within the service and must not place this jar in the *impl/src/main/lib* folder. |

**Recipe 3: Instantiating Entities (Service API Descriptor)**

Any entities defined in the service API descriptor (in yang/json format) need special handling. On running the code-generator on the service API descriptor, it gives us an interface for the entity and an implementation (which is *internal* & not meant to be instantiated directly).

The reason for this is that these entities are sent across the wire. Not dealing with these entities directly gives us the flexibility to change the serialization/deserialization mechanism without affecting any existing code. For example, take a look at *quickstarts/todo-list/api/json/todo-list.json*. It defines a domain entity called *todo* which is used in the RPCs for this service.

```
"todo" : {
      "description" : "Represents a TODO item",
      "properties" : {
            "id" : {
                  "type" : "string",
                  "description" : "Id of the TODO"
            },
            "summary" : {
                  "type" : "string",
                  "description" : "Summary of the TODO"
            }
      }
}
```

Running the code-generator on this, generates 2 entities, one is an interface, the other is a class :

- com.macaw.quickstart.todo.Todo interface
- com.macaw.quickstart.todo.internal.impl.Todo class

Whenever we need to create an instance of Todo, we use the com.macaw.quickstart.todo.Todo interface. This is how we instantiate a Todo object in <SDK Directory>/quickstarts/todo-list/impl/src/main/java/com/macaw/quickstart/todo/impl/TodoList.java.

```
import com.macaw.quickstart.todo.Todo;
import com.macaw.quickstart.todo.DomainEntityInstantiator;

final Todo todo =
DomainEntityInstantiator.getInstance().newInstance(Todo.class);
todo.setId(todoId.toString()).setSummary(summary);
```

As you can see above, we use the *DomainEntityInstantiator.getInstance().newInstance()* mechanism to create an instance of the desired class. Note that the *DomainEntityInstantiator* is specific to this service. In fact, this is generated specifically for this service and is bundled in the API jar for the service. So, if any other service wants to create the Todo object (for example, if the Todo object is sent as the payload of a notification), they need to have the Todo API jar in their classpath.

**Recipe 4: Supporting Microservice Databases**

Macaw Platform doesn't mandate or restrict the use of databases within Microservices. Microservice implementations are free to use any database of their choice and interact with it within the implementation of their service.

Macaw Platform, however, does allow microservices to have their database schemas provisioned and managed by the platform. The developer can interact with this provisioned instance just like any other database instance. The database instance's lifecycle, however, will be managed by the Macaw Platform.

Currently, if a microservice wants the Macaw Platform to provision and manage a database instance, then the following database servers are supported by the platform (support for more database servers will be added in future):

- Cassandra 2.2.x

**Recipe 5: Database Schema Provisioning**

As a developer, you can enable provisioning and management of the database instance for your microservice. The *macaw publish* tool that we explained earlier looks for certain files within the implementation of the microservice to instruct the Macaw platform to provision the database instance.

As a developer, you are expected to have a file named ddl.ql and dml.ql under the following folder hierarchy:

```
<service-name>
    |
    |--- impl
    |     |
    |     |--- etc
    |     |    |
    |     |    |--- db
    |     |    |       |
    |     |    |   |--- ddl.ql
    |     |    |       |--- dml.ql
```

The ddl.ql is expected to contain the database creation queries, where as the dml.ql is expected to contain any initial/seed data that you need inserted into the database. If there is no initial data you want seeded, you can leave out the dml.ql file.

An example of ddl.ql for Cassandra database would look something like:

```
CREATE TABLE todo (
    id    UUID,
    summary text,
    PRIMARY KEY ( (id) )
);
```

| NOTE | At this time, Macaw supports Cassandra backends for microservices. In the near future, support for other backends will be added. |
|---|---|

**Recipe 6: Accessing Databases**

The Macaw platform sends the provisioned database instance details via the com.cfx.service.api.Configurationobject that gets passed to the initialize method of the service implementation. This configuration object has predefined configuration keys that can be used to get hold of the details of the provisioned database, so that the service implementation can then seamlessly interact with the database within the service implementation.

For Cassandra server, the passed configuration keys are as follows:

- db.cassandra.clusterNodes – The value of this configuration property will be a comma separated list of host:port combinations of the initial nodes belonging to the provisioned Cassandra database instance cluster.

Example: 10.10.20.20:9092

- db.cassandra.keyspace – The value of this configuration property will be the keyspace that has been provisioned, by the Macaw platform for the microservice.

Example: calculator-service-aea66ae2-b2f9-11e6-afb9-5bea1c3a4d5d

- db.cassandra.username – The value of this property will be the user name to connect to the provisioned database instance
- db.cassandra.password – The value of this property will be the password to use to connect to the provisioned database instance

The Microservice implementation can use these passed configurations to communicate with the database instance that has been provisioned for the service. We recommend the Datastax Cassandra driver, but you are free to use any other Cassandra driver for your service as long as it is compatible with the Cassandra version supported by Macaw. The Macaw platform provisions the database schema, one per service instance cluster. All service instances in the service instance cluster, share the same database schema. The generated database schema name is random.

| | |
|---|---|
| NOTE | You can refer to <SDK>/quickstarts/todo-list/impl/src/main/java/com/macaw/quickstart/todo/impl/TodoList.java for sample code which shows how to initialize connections to the provisioned schema, how to access data from it etc. |

**Recipe 7: Typed Microservice Invocation**

Assume that the Microservice you just developed and deployed needs some information from another microservice. How to invoke an RPC on it? There are two ways to do it :-
**Typed invocation** : This approach involves, including the target service's API library in the compile time and runtime classpath of the service which is invoking the target service.
**De-typed invocation** : In this approach, the service which wants to invoke on any target service, doesn't require access to the target service's API library either at compile time or at runtime. The type safety of the objects passed around during invocation of the APIs, is only verified at runtime, by the target service which handles the invocation. As a result, the caller service doesn't require any access to the target service's API library for static typing.

Rest of this section deals with typed invocations. The process for *typed invocation* is as follows :-

- Contact the target service developer and get the API jar for the target service. This is a necessary (manual) step for typed invocations. For example, let's assume you are developing a service which wants to invoke a method on the calculator service. In this case, you should contact the developer/owner of the calculator service to get the API jar.
- Place this API jar for the target service in the <service-project-root>/impl/src/main/lib folder of your service. This path contains the libraries that are necessary for the service being developed and these libraries will also get packaged into the generated service binary.
- Get hold of the ServiceClientContext. The Macaw service framework can inject the ServiceClientContext in the service impl class. You just need to declare a variable as shown below.

```
@Inject
private ServiceClientContext serviceClientContext;
```

- Get the ServiceLocator from the ServiceClientContext.

```
ServiceLocator serviceLocator = serviceClientContext.getServiceLocator();
```

- Any Service deployed on the Macaw platform registers with the Service Registry. So, we need to do a lookup for the target service on the registry. To accomplish that, call locateService() method on the locator with the unique identifiers of the target service (it's name, namespace & a optional version).

```
Calculator Servicecalculator =
serviceLocator.locateService(serviceClientContext.getInvocationContextSe
ssion(),
http.macaw.io.quickstart.service.calculator.rev160608.Calculator.class,
"io.macaw.services", "calculator");
```

Once we have the Service API post the lookup, we invoke the desired RPC on it similar to invoke methods on any other Java objects.

| | |
|---|---|
| NOTE | We are currently working on a mechanism using which service developers can publish the APIs of the services they develop, so that those can be consumed by interested parties. |

**Recipe 8: Detyped Microservice Invocation**

Assume that a microservice needs information from another microservice and needs to invoke RPC on it. This can be achieved in two ways - Typed and De-Typed. The rest of this section shows and explains an example of de-typed invocations. We will use the issue-tracker quickstart shipped with the Macaw SDK as a reference for de-typed invocations. Please refer to the issue-tracker quickstart code for a complete example.

The issue-tracker service exposes the create-account API which is expected to create a user account within the issue tracking system. Typically, in the microservices world, for a module like user management, you would typically have it as a separate service. We do the same, in our quickstarts too. We have a user management service which specifically deals with user creation and management. Of course, the purpose of that quickstart is meant to be a basic example of user management. Our issue-tracker service internally uses the user-management service for management of user accounts of the issue tracker system. So whenever, a create-account API is invoked, the implementation of the issue-tracker service looks up the user-management service and does a de-typed invocation on it to create a user. Following is the relevant snippet of (with inline comments on how it's done):

```
private static final String USER_MANAGEMENT_SERVICE_NAMESPACE =
"io.macaw.services";
private static final String USER_MANAGEMENT_SERVICE_NAME =
"user-management";
final Session session = this.serviceClientContext.getInvocationContextSession();
// lookup the user management service and invoke on it in a "detyped" way (i.e.
we *don't* require the interfaces
// of the user management service, statically in our classpath)
final ServiceInvoker serviceInvoker =
this.serviceClientContext.getServiceLocator().locateServiceInvoker(session,
USER_MANAGEMENT_SERVICE_NAMESPACE,
USER_MANAGEMENT_SERVICE_NAME);
// invoke the API to create the user, on the user management service
final String apiMethodName = "createUser";
final String[] apiMethodArgTypes = new String[]{String.class.getName(),
String.class.getName(), String.class.getName()};
final String[] apiMethodArgs = new String[]{userId, firstName, lastName};
try {
   serviceInvoker.invoke(apiMethodName, new
JSONMethodDescriptor(apiMethodArgTypes, apiMethodArgs));
} catch (Exception e) {
   throw new RuntimeException("Failed to assert validity of user account of user
" + userId, e);
}
```

Let's go over the snippet above to understand in more detail on how it's done. Let's start with this statement:
final Session session = this.serviceClientContext.getInvocationContextSession();

Here we get hold of the service invocation session, which we will then later use for lookup of services.

| NOTE | A service invocation always has a session associated with it. Furthermore, lookup and invocations on services aren't allowed without the usage of a valid session |
|---|---|

Once we have that session, the next thing we do in that code is to locate a ServiceInvoker for the user-management service. ServiceInvoker`s is an API exposed by the Macaw service framework to allow invoking on services in a de-typed manner. Once we get a `ServiceInvoker, the next step is to call the invoke API that it exposes. The invoke API expects the method name of the target service method, which we want to invoke and an instance of MethodDescriptor interface:

```
/**
 * Invokes on the service method named <code>methodName</code> and
which accepts method parameters of type specified
 * in <code>methodArgTypes</code>. The
<code>methodArgumentsProvider</code> will be used to get the method
arguments
 * that will be passed on to the invoked method.
 *
 * @param methodName
 *          The name of the method to invoke
 * @param methodDescriptor
 *          Provides method arguments that will be used for the method
invocation. The method arguments returned
 *          by the <code>methodArgumentsProvider</code> can either be
directly passed on to the invoked method or
 *          could potentially undergo some conversion to relevant type, before
being passed on to the invoked
 *          method of the service. Whether or not the conversion is needed,
depends on the
 *          {@link MethodDescriptor#getType() type of the MethodDescriptor}
 * @return
 */

Object invoke(String methodName, MethodDescriptor methodDescriptor) throws
Exception;
```

The MethodDescriptor interface itself looks as follows:

```
public interface MethodDescriptor {

   MethodParamType getParamType();

   String[] getMethodArgTypes();

   Object[] provideMethodArgs();
```

```
}
```

The code in our implementation of issue-tracker is creating a JSONMethodDescriptor, an implementation provided by the Macaw service framework library, that uses JSON format for handling the de-typed invocations. As noted in the code snippet previously, here's how the invocation looks like, in our issue-tracker service:

```
// invoke the API to create the user, on the user management service
final String apiMethodName = "createUser";
final String[] apiMethodArgTypes = new String[]{String.class.getName(),
String.class.getName(), String.class.getName()};
final String[] apiMethodArgs = new String[]{userId, firstName, lastName};
try {
   serviceInvoker.invoke(apiMethodName, new
JSONMethodDescriptor(apiMethodArgTypes, apiMethodArgs));
} catch (Exception e) {
   throw new RuntimeException("Failed to assert validity of user account of user
" + userId, e);
}
```

We pass in the method name of the target service method and the method argument types and the method arguments itself, to the ServiceInvoker's `invoke method to have the de-typed invocation done.

| | |
|---|---|
| NOTE | The ServiceInvoker.invoke returns an java.lang.Object type, which if invoked through the JSONMethodDescriptor is going to be a java.lang.String. The return value will be a valid JSON value type (JSON string, JSON object, JSON array, TRUE, FALSE, JSON number or JSON NULL). In the above example, we aren't concerned about the return type and hence we don't see it's usage there. Please refer to the issue-tracker quickstart to see how the return type gets used, in a different part of the code, which too does a de-typed invocation. |

**Recipe 9: Notification Publication**

Microservices are designed to run in bounded contexts. Each microservice is the master of some domain data. In a microservices installation, it's quite possible that other microservices are interested in consuming any notifications published by a microservice.

For example, a microservice that is responsible for user management can publish a notification on events like *user addition*, *user deletion,* etc. Other microservices interested in *user events* can subscribe to notifications from the user management microservice.

This is how a microservice can publish notifications in Macaw. In the service specification, the service must declare details of the notifications that it raises/publishes. For example, look at the service specification (in yang format) defined at <sdk folder>/quickstarts/employee/api/yang/employee.yang.

```
notification EMPLOYEE_RELIEVED {
   description "A notification which is published when an employee is relieved.";
   leaf email-id {
      type string;
      description "Email of the relieved employee.";
   }
}
```

The snippet above declares that the *employee* service raises a notification called *EMPLOYEE_RELIEVED*. The notification payload includes a string called *email-id*. This is a simple notification. Now assume that as a part of the notification payload, we want to include a custom object. How can this be done? Look at the same YANG file for another notification declaration as shown below :-

```
notification NEW_EMPLOYEE_ADDED {
   description "A notification which is published when a new employee is added
by the employee service.";
   uses grp-employee; // point to the employee object
}

grouping grp-employee{
   container employee {
      leaf id {
         type int32;
         mandatory false;
         description "Unique identifier for the employee";
      }
      leaf first-name {
         type string;
         mandatory true;
         description "First name of the employee";
      }
      leaf last-name {
         type string;
         mandatory false;
```

```
            description "Last name of the employee";
        }
        leaf email {
            type string;
            mandatory true;
            description "Email of the employee(Email id should be unique). Email id
will be used to login to the organization portal.";
        }
        ...
    }
}
```

The snippet above declares that the *employee* service raises a notification called *NEW_EMPLOYEE_ADDED*. The notification payload comprises of a *employee* object, which is a custom object defined by the service.

With these declarations we have expressed to the Macaw platform which type of notifications our employee sample service publishes. This is important so that the platform knows which notifications are available for consumption. Now, within the code of the service, these notifications must be raised. For an example of the same, please look at </impl/src/main/java/http/macaw/io/quickstart/service/employee/rev161201 /impl/EmployeeServiceImpl.java. Let's look at the addEmployee()method and how we raise the NEW_EMPLOYEE_ADDED notification when this RPC is invoked.

```
@Inject
private ServiceClientContext serviceClientContext;

@Override
public int
addEmployee(http.macaw.io.quickstart.service.employee.rev161201.Employee
employee) {
    if (employee == null) {
        throw new IllegalArgumentException("Null employee cannot be added");
    }
    if (employee.getEmail() == null) {
        throw new IllegalArgumentException("Employee with a null email address
cannot be added");
    }
    employee.setId(employeeIdCounter.getAndIncrement());
    employeeCache.put(employee.getId(), employee);
    safePublishNotification(EMPLOYEE_ADDED_NOTIFICATION_ID, employee);
    return employee.getId();
}

public void safePublishNotification(final String notificationId, Object payload) {
    try {
        this.serviceClientContext.getNotificationManager().publish(notificationId,
payload);
    } catch (Exception e) {
        e.printStackTrace();
    }
```

```
}
```

As shown in the snippet above, on execution of the *addEmployee* RPC, first the employee object is stored in an in-memory cache and then raise the notification by getting hold of the *NotificationManager* reference from the *ServiceClientContext* and then called *publish()* on it. That code takes care of publishing the notification & it can now be consumed by other services which subscribe to it.

| | |
|---|---|
| NOTE | If the user changes the service spec to add/modify/delete any notification publication declarations; they must regenerate the service via the code-generator. The notification subscription information is kept in the generated *impl/src/main/resources/conf/service-info.xml* artifact. It must reflect the user's notification related changes in the *notifications* block. |

**Recipe 10: Notification Subscription & Consumption**

This recipe explains how you can subscribe to the published notifications and also consume notifications in another microservice.

**Subscribing for a notification:**

In order to subscribe for notifications, a service needs to declare in its service spec which specific notifications it wants to subscribe to. For example, look at the service spec (in yang format) defined at quickstarts/intranet-portal/api/yang/intranet-portal.yang.

```
import macaw-notification {
  prefix n;
}
container notification-subscriptions {
   n:subscription employee-added {
      n:notification-id  "NEW_EMPLOYEE_ADDED";
      description "Notification subscription when new employee is added.";
      n:service-name "employee";
      n:service-version "1.0.0";
      n:service-namespace "io.macaw.services";
   }
   subscription employee-relieved {
      n:notification-id  "EMPLOYEE_RELIEVED";
      description "Notification subscription when employee is relieved.";
      n:service-name "employee";
      n:service-version "1.0.0";
      n:service-namespace "io.macaw.services";
   }
}
```

As you can see from the snippet above, we declare which notifications we want to subscribe to within a *notification-subscriptions* construct. For each notification that we want to subscribe to, we add a *subscription* declaration which specifies :-

- notification-id – id of the notification to subscribe to
- service-name – name of the service
- service-version – version of the service
- service-namespace – namespace of the service
-

| NOTE | that if you change the service spec to add/modify/delete any notification subscription declarations; you must regenerate the service via the code-generator. The notification subscription information is kept in the generated *impl/src/main/resources/conf/service-info.xml* artifact. It must reflect your notification related changes in the *notifications* block. |
|---|---|

**Consuming a notification:**

So now that we have declared in the previous section, which notifications we want to subscribe to; let's figure how we consume those notifications. Note that Macaw platform will take care of the subscribing (based on the declaration) and making the notification messages available for consumption. The messages are available for consumption in the *onNotification()* method in the service implementation class. Its signature is as shown below.

```
public void onNotification(com.cfx.service.api.notification.Notification notification);
```

Let's look at the quickstarts/intranet-portal/impl/src/main/java/http/macaw/io/quickstart/service/intranet/portal/rev161201/impl/IntranetPortalImpl.java class for an actual example of the same method.

```
public void onNotification(com.cfx.service.api.notification.Notification notification) {
   switch (notification.getIdentifier().getNotificationId()) {
   case EMPLOYEE_ADDED_NOTIFICATION_ID:
      System.out.println("proccessing notification for newly added employee to the organisation.");
      Employee addedEmployee = (Employee) notification.getContent();
      userDetails.put(addedEmployee.getEmail(), addedEmployee.getPassword());
      break;
   case EMPLOYEE_RELIEVED_NOTIFICATION_ID:
      System.out.println("proccessing notification for employee relieved.");
      String deletedUserEmailId = (String) notification.getContent();
      userDetails.remove(deletedUserEmailId);
      break;
   default:
         throw new IllegalArgumentException("Unknown notification received : " + notification.getIdentifier().getNotificationId());
   }
}
```

As you can see from the above code, based on the notification ID, we can figure out which notification we need to process. The payload of the notification is available via the *notification.getContent()* method. It can be cast to the correct type to be used further. Remember that this type (in this case, *Employee*) is defined in the publishing service. Hence you need to have the API jar of the publishing service in your classpath in order to perform the cast. This is why we have the *employee-api.jar* in the *impl/src/main/lib* folder of *intranet-portal* sample service.

**Recipe 11: Continued Notifications**

In order to subscribe for notifications, a service needs to declare in its service specification which specific notifications it wants to subscribe to. For example, look at the service spec (in YANG format) defined at quickstarts/intranet-portal/api/yang/intranet-portal.yang.

```
import macaw-notification {
  prefix n;
}
container notification-subscriptions {
   n:subscription employee-added {
      n:notification-id  "NEW_EMPLOYEE_ADDED";
      description "Notification subscription when new employee is added.";
      n:service-name "employee";
      n:service-version "1.0.0";
      n:service-namespace "io.macaw.services";
   }
   n:subscription employee-relieved {
      n:notification-id  "EMPLOYEE_RELIEVED";
      description "Notification subscription when employee is relieved.";
      n:service-name "employee";
      n:service-version "1.0.0";
      n:service-namespace "io.macaw.services";
   }
}
```

As can be seen from the snippet above, Macaw declares which notifications it wants to subscribe to within a *notification-subscriptions* construct. For each notification desired to subscribe to, Macaw adds a *subscription* declaration which specifies :-

- notification-id – id of the notification to subscribe to
- service-name – name of the service
- service-version – version of the service
- service-namespace – namespace of the service

| | |
|---|---|
| NOTE | that if user changes the service spec to add/modify/delete any notification subscription declarations; the user must regenerate the service via the code-generator. The notification subscription information is kept in the generated *impl/src/main/resources/conf/service-info.xml* artifact. It must reflect your notification related changes in the *notifications* block. |

**Consuming a notification:**

So now that the previous section has been declared, which notifications desired to subscribe to, let's figure how to consume those notifications. Note that Macaw platform will take care of the subscribing (based on the declaration) and making the notification messages available for consumption. The messages are

available for consumption in the *onNotification()* method in the service implementation class. Its signature is as shown below.

```
public void onNotification(com.cfx.service.api.notification.Notification notification);
```

Let's look at the quickstarts/intranet-portal/impl/src/main/java/http/macaw/io/quickstart/service/intranet/portal/rev161201/impl/IntranetPortalImpl.java class for an actual example of the same method.

```
public void onNotification(com.cfx.service.api.notification.Notification notification) {
   switch (notification.getIdentifier().getNotificationId()) {
   case EMPLOYEE_ADDED_NOTIFICATION_ID:
      System.out.println("proccessing notification for newly added employee to the organisation.");
      Employee addedEmployee = (Employee) notification.getContent();
      userDetails.put(addedEmployee.getEmail(), addedEmployee.getPassword());
      break;
   case EMPLOYEE_RELIEVED_NOTIFICATION_ID:
      System.out.println("proccessing notification for employee relieved.");
      String deletedUserEmailId = (String) notification.getContent();
      userDetails.remove(deletedUserEmailId);
      break;
   default:
      throw new IllegalArgumentException("Unknown notification received :" + notification.getIdentifier().getNotificationId());
   }
}
```

As seen from the above code, based on the notification ID, which notifications need to be processed can be figured out. The payload of the notification is available via the *notification.getContent()* method. It can be cast to the correct type to be used further. Remember that this type (in this case, *Employee*) is defined in the publishing service. Hence the user needs to have the API jar of the publishing service in your classpath in order to perform the cast. This is why we have the *employee-api.jar* in the *impl/src/main/lib* folder of *intranet-portal* sample service.

**Recipe 12: Support for Stateful Microservices**

Microservices are typically implemented as stateless processes running in clusters with all underlying state stored in a database. However, there are some use cases which require support for Stateful microservices.
Macaw platform supports implementation of stateful microservices by providing services with transparent access to a cluster-wide cache which can be used to cache & share information across services running in a cluster in a performant manner.
In order to use this functionality, services have to initialize the CacheContext in their start() method as shown below.

```
  @Inject
    private ServiceClientContext serviceClientContext;

    private CacheContext cacheContext;

    @Override
    public void start(com.cfx.service.api.StartContext startContext) throws
ServiceException {
        ...

        this.cacheContext =
serviceClientContext.getRuntimeFeature(CacheContext.class);
        try {
            this.cacheContext.initialize();
        } catch (CachingException e) {
            throw new ServiceException("Failed to initialize cache", e);
        }

        ...
    }
```

Once the service has access to a CacheContext instance, it can store/fetch/delete any objects in/from it. Here are the relevant methods exposed on the CacheContext interface :-

```
    /**
     * Associates the specified value with the specified key in this cache
     *
     * @param key with which the specified value is to be associated
     * @param value Serializable value to be associated with the specified key
     * @throws CachingException
     */
    public void set(String key, Object value) throws CachingException;

    /**
     * Returns the value to which the specified key is mapped,
     *
     * @param key the key whose associated value is to be returned
     * @return the value to which the specified key is mapped, or {@code null}
     *         if this map contains no mapping for the key.
```

```
     * @throws CachingException
     */
    public Object get(String key) throws CachingException;

    /**
     * Deletes the mapping for a key from this cache if it is present
     *
     * @param key key whose mapping is to be deleted from the cache
     * @throws CachingException
     */
    public void delete(String key) throws CachingException;
```

When the service shuts down, the CacheContext instance should be cleaned up. This can be done by invoking CacheContext.close() in the service stop() lifecycle method as shown below :-

```
    @Override
    public void stop(com.cfx.service.api.StopContext stopContext) throws
ServiceException {
        ...
        try {
            this.cacheContext.close();
        } catch (IOException e) {
            // Do nothing...
        }
        ...
    }
```

For example, usage of the cluster-wide caching functionality, please look at <SDK Folder> quickstarts/issue-tracker.

**Chapter 16: Where to go**

We tried to cover the core essential topics to enable you to understand and start developing Microservices using Macaw Framework. We are happy to receive your feedback, comments, or suggestions to improve this book. If you want more information on Macaw Framework, feel free to contact us. We are available at:

1) Email - info@macaw.io
2) Website - https://macaw.io/
3) Address - Macaw Software Inc.,
7901 Stoneridge Drive, #300,
Pleasanton, CA 94588

4) Follow us on Twitter at https://twitter.com/MacawBuzz
5) Get unto date information at http://www.macaw.io/macawblog/

**Disclaimer**: While we tried to provide up to date information on Macaw Framework, due to our rapid innovations and developments on Macaw Framework, this eBook could be out of date. We request you to always check latest documentation at https://macaw.io/documentation/default/.