



# OMP MENTORSHIP PROGRAM 2020: PROJECT ZEBRA

Project Team  
**Yongkook(Alex) Kim**  
**Salisu Ali**

## Contents:

### **User Documentation**

• How to Run App .....	1
• How to Configure App .....	2
• How to Connect Prometheus .....	3
• How to Connect Grafana .....	5
• How to Connect MongoDB .....	7
• App Queries .....	8

### **Contributor Documentation**

• Project Description .....	15
• User Requirement .....	15
• System Environment .....	16
• Design Approach .....	16
• Design Patterns .....	16
• Design Consideration .....	17
• Architecture Design .....	18
• Class Diagram .....	19
• Activity Diagram .....	20
• Sequence Diagram .....	23
• Entity Diagram .....	27

## **Setting up your Environment**

In order to run zebra, you will need to set up your environment by installing Nodejs version 8.11.2. Installing Nodejs comes along with npm. Git is also needed for cloning the app from a github repository. Prometheus, MongoDB and Grafana are other programs needed to run Zebra.

## **Clone Zebra**

To clone zebra from OMP github repository, use the command:

- git clone <https://github.com/openmainframeproject-internship/Zowe-Parsing-Engine-for-SMF-or-RMF-PP-Reports.git>

## **Installing App Packages**

Follow these steps to install the app packages using a terminal/command prompt:

- cd into the cloned app root folder
- run npm install.

The app will install packages as contained in the package.json folder.

## **How to Run App**

To run Zebra App on a server or Local Machine, a user can choose to run the app using npm, nodemon or pm2.

### **i. Using NPM (For Testing/Contributing)**

Follow these steps to run Zebra App using npm from a terminal/command prompt:

- cd into the cloned app root folder (if you are not there already).
- run npm start

**Note:** using this method, a user has to stop and restart zebra whenever he/she made a change to the apps configuration for the changes to take effect.

### **ii. Using Nodemon (For Testing/Contributing)**

After installing nodemon, follow these steps to run Zebra App using nodemon from a terminal/command prompt:

- cd into the cloned app root folder (if you are not there already).
- run nodemon

**Note:** using this method, Zebra Automatically restart whenever a user made a change to the apps configuration.

### iii. Using PM2

Follow these steps to run Zebra App using pm2 from a terminal/command prompt:

- cd into the cloned app root folder (if you are not there already).
- run pm2 start ./bin/www

**Note:** use this method for production, it helps manage and keep your application running 24/7

### How To Configure App

Zebra's configuration gives users the flexibility to run the app according to their needs. Zebra provides “/addsettings” endpoint for editing config values. This endpoint can take one or more parameters recognized by zebra at once. E.g.

<http://localhost:3090/addsettings?appurl=127.0.0.1>

### Recognized Zebra Config Parameters

- **ddsbseurl: (Distributed Data Server base URL)**

This is the IP address of the DDS from which the App can retrieve RMF Data. E.g.

<http://localhost:3090/addsettings?ddsurl=127.0.0.1>

- **ddsbseport: (Distributed Data Server base Port)**

This is the Port number of the DDS from which from which the App can retrieve RMF Data. E.g

<http://localhost:3090/addsettings?ddsport=8888>

- **rmf3filename: (DDS RMF Monitor III Filename)**

This is the filename from which The App can retrieve RMF Monitor III data. E.g.

<http://localhost:3090/addsettings?rmf3filename=rmfm3.xml>

- **rmfppfilename: (DDS RMF Monitor I Filename)**

This is the filename from which The App can retrieve RMF Monitor I data. E.g.

<http://localhost:3090/addsettings?rmfppfilename=rmfpp.xml>

- **mvsResource: (DDS RMF Monitor III resource identifier)**

This parameter represent Monitor III resource identifier. E.g.

<http://localhost:3090/addsettings?mvsResource=,SYS,RESOURCE>

- **mongourl: (Mongo DB URL)**

This is the IP address of MongoDB to which RMF III data will be saved. E.g.

<http://localhost:3090/addsettings?mongourl=127.0.0.1>

- **dbinterval: (Database Interval)**

This is the Interval for which data will be saved to MongoDB. Its unit is in seconds. E.g. Every 100 seconds

<http://localhost:3090/addsettings?dbinterval=100>

- **dbname: (Database Name)**

This is the name of the database for which RMF III data will be saved. E.g.

<http://localhost:3090/addsettings?dbname=ompzebra>

- **appurl: (Running Zebra App URL)**

This is the IP address of running Zebra instance (after hosting). This value is needed for MongoDB, Prometheus and Grafana to work with Zebra. E.g. 127.0.0.1.

<http://localhost:3090/addsettings?appurl=127.0.0.1>

- **appport: (Running Zebra App Port)**

This is the Port of a running Zebra Instance (after hosting). This value is needed for MongoDB, Prometheus and Grafana to work with Zebra. E.g. 3000

<http://localhost:3090/addsettings?appport=3000>

- **mongoport: (Mongo DB Port)**

This is the port number of MongoDB to which RMF III data will be saved. E.g.

<http://localhost:3090/addsettings?mongoport=27017>

- **ppminutesInterval: (RMF Monitor I PP Report Interval)**

This is the Interval for which DDS Produce RMF I report. Its unit is in minutes E.g. Every 30 Minutes

<http://localhost:3090/addsettings?ppminutesInterval=30>

- **rmf3interval: (RMF Monitor III Report Interval)**

This is the Interval for which DDS Produce RMF III report. E.g. Every 100 seconds

<http://localhost:3090/addsettings?rmf3interval=100>

- **httptype: (http type of running Zebra App)**

This is the hypertext transfer protocol type of the running zebra app (after hosting). Its value is either http or https. E.g.

<http://localhost:3090/addsettings?httptype=http>

- **useDbAuth: ( Use Database Authentication)**

This config parameter determines the type of connection to mongodb, either with authentication or without authentication. Its value is either **true** or **false**. If value is set to true, zebra will require username and password with which to connect to mongodb. E.g.

<http://localhost:3090/addsettings?useDbAuth=true>

- **dbUser: (Database Username)**

This is mongodb username with which zebra will connect to database if value of useDbAuth is set to true. E.g.

<http://localhost:3090/addsettings?dbUser=salisuali>

- **dbPassword: (Database Password)**

This is mongodb password with which zebra will connect to database if value of useDbAuth is set to true. E.g.

<http://localhost:3090/addsettings?dbPassword=salisu>

- **authSource: (Authentication Source)**

This is mongodb database which contains the username and password for authentication. The default is mongodb's "**admin**" database. E.g.

<http://localhost:3090/addsettings?authSource=admin>

- **useMongo: (Use Mongo DB)**

This parameter needs to be set to true before a user can connect MongoDB to Zebra. Its default value is false. E.g.

<http://localhost:3090/addsettings?useMongo=true>

- **usePrometheus: (Use Prometheus server)**

This parameter needs to be set to true before a user can connect Prometheus to Zebra. Its default value is false. E.g.

<http://localhost:3090/addsettings?usePrometheus=true>

- **https: (Use TLS)**

This parameter needs to be set to true before a user can use TLS for Zebra. Its default value is false. E.g.

<http://localhost:3090/addsettings?https=true>

- **pfx: (password protected certificate)**

This parameter needs to be set to true before a user can use password protected certificate. Its default value is undefined. E.g.

<http://localhost:3090/addsettings?pfx=true>

- **passphrase: (password protected certificate passphrase)**

This parameter contains the passphrase of the pfx. Its default value is None. E.g.

<http://localhost:3090/addsettings?passphrase=1234>

The user can specify multiple configuration parameters at once:

<http://localhost:3090/addsettings?dbinterval=3600&appport=3000>

The above URL contains two parameters dbinterval and appport. The user can specify as many parameters as possible with their value each. These new values specified by the user will overwrite the value stored in zebra's config file.

### **How to view Zebra Config**

Zebra provides default values for config parameters. It provides “/settings” endpoint to view app config settings and choose the parameters to modify:

<http://localhost:3090/settings>

### **How to Connect MongoDB to Zebra**

The first step in connecting MongoDB to zebra is setting the value of **useMongo** config parameter to “true”.

- Without Authentication**

If a user chooses to connect Zebra to MongoDB without Authentication, he/she will need to set **useDbAuth** parameter to **false** and provide the correct values for other config parameters.

### ii. With Authentication

If a user choose to connect to MongoDB using username and password. The user will need to follow these procedures:

#### a. Enable MongoDB Access Control

To Enable MongoDB Access Control, the user will first need to create a Username and password with a read Write Permission/Role. To do this, the user will need to:

##### ○ Connect to the MongoDB instance

For example, open a new terminal and connect a mongo shell to the instance:

```
mongo --port 27017
```

Specify additional command line options as appropriate to connect the mongo shell to your deployment, such as **-host**

##### ○ Create the user administrator

From the mongo shell, add a user with the **userAdminAnyDatabase** role in the admin database. Include additional roles as needed for this user. For example, the following creates the user **myUserAdmin** in the admin database with the **userAdminAnyDatabase** role and the **readWriteAnyDatabase** role.

```
use admin
db.createUser(
  {
    user: "myUserAdmin",
    pwd: passwordPrompt(), // or cleartext password
    roles: [ { role: "userAdminAnyDatabase", db: "admin" }, "readWriteAnyDatabase" ]
  }
)
```

#### b. Modify MongoDB Configuration File

The next step is to add the **security.authorization** configuration file setting:

```
security:
  authorization: enabled
```

**Note:** after making the above changes, **Re-start the MongoDB instance**

#### c. Change Zebra Config Parameters



To use MongoDB with Authentication from Zebra, a user need to set **useDbAuth** parameter to **true**. The user should also provide values for **dbUser**, **dbPassword**, **authSource** and the correct values for remaining config parameters.

### How to Connect Prometheus to Zebra

The first step in connecting MongoDB to zebra is setting the value of **usePrometheus** config parameter to **"true"**.

Zebra provides **"/prommetric"** Endpoint that expose Custom Prometheus metrics. Users can scrape these metrics by using Prometheus. Simply modify the config file of Prometheus to point to Zebra's **"/prommetric"** Endpoint:

```
31
32   - job_name: 'zebra'
33     metrics_path: '/prommetric'
34     scrape_interval: 60s
35     static_configs:
36       - targets: ['localhost:3090']
37
```

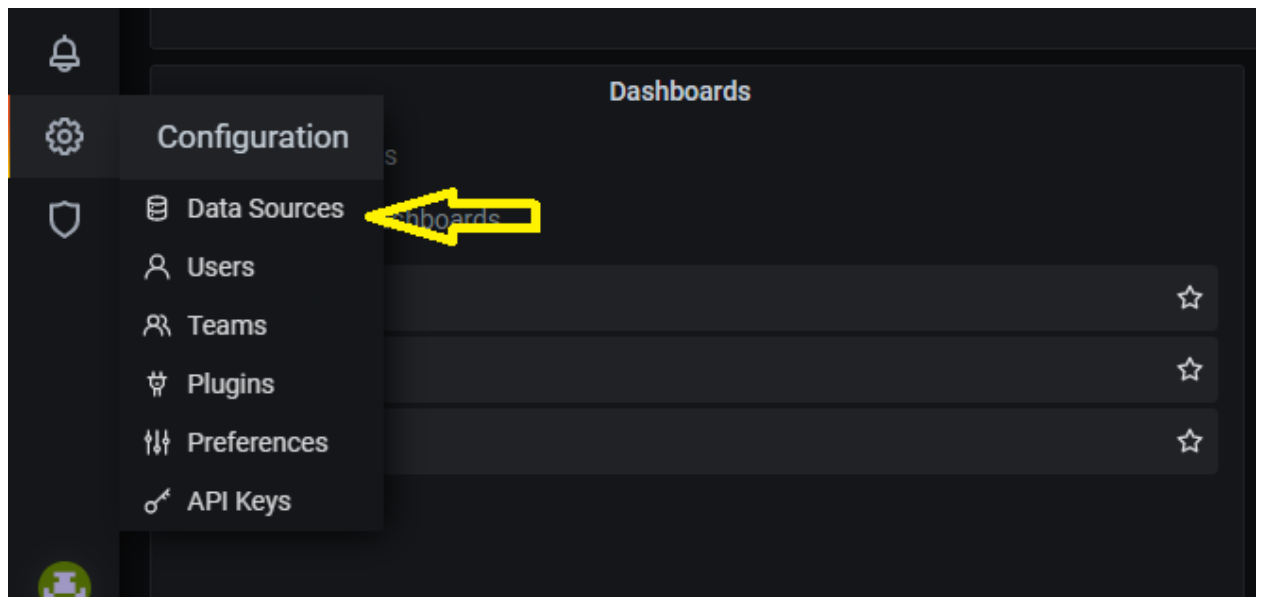
**Note:** Modify the value for target to point to IP address and port of your running Zebra Instance.

### How to Connect Grafana to Prometheus

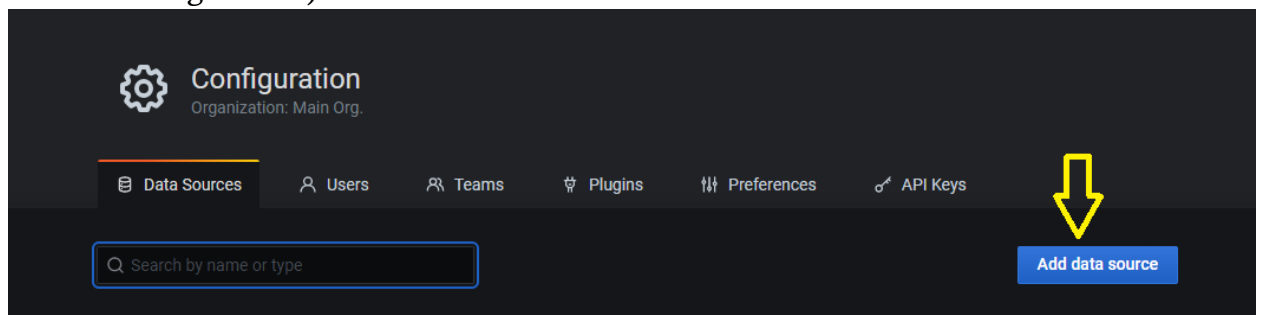
When Prometheus has been connected to Zebra, the values scraped by Prometheus can be used to create Realtime dashboards using Grafana.

#### - Data Source

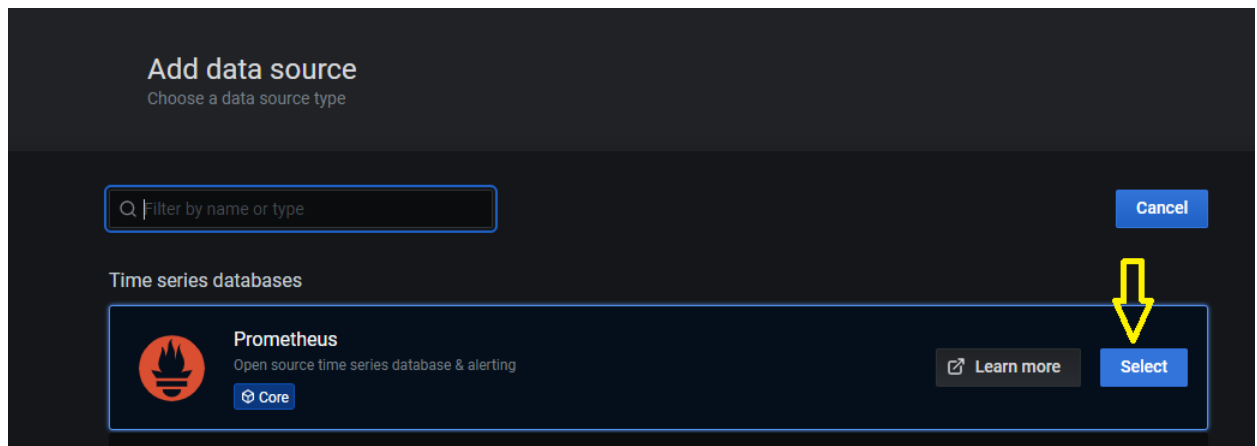
Prometheus will be our data source and the data it scrapes will be used to plot our real time chart. Use the following steps to add Prometheus as a data source to Grafana.



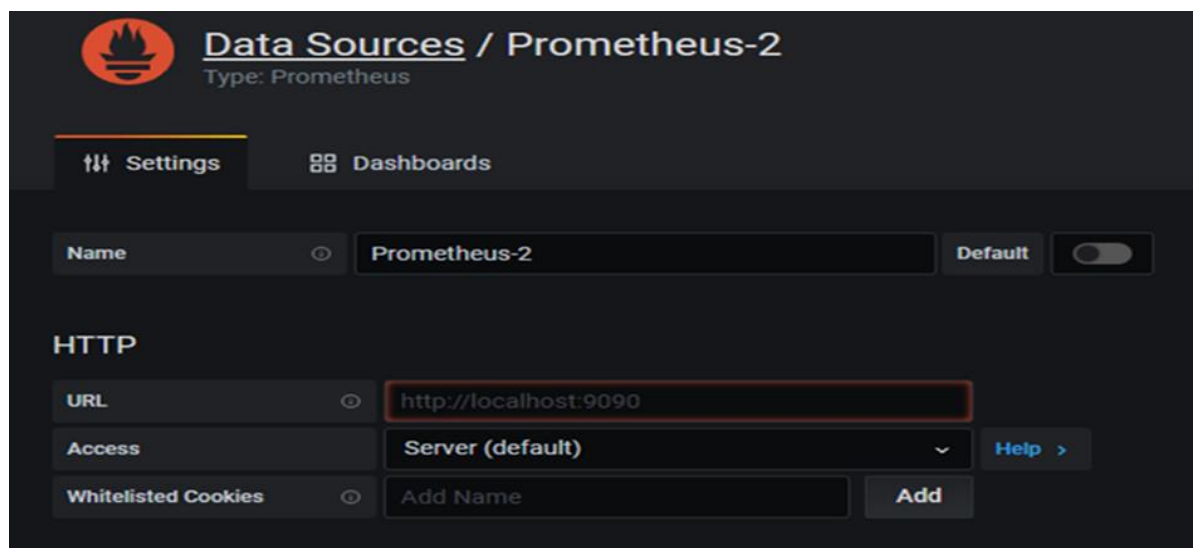
- i. After logging into Grafana, click on the **configuration button** and select **Data source** (as shown by the yellow arrow in the image above).
- ii. Click on **Add data source** button (as shown by the yellow arrow in the image below)



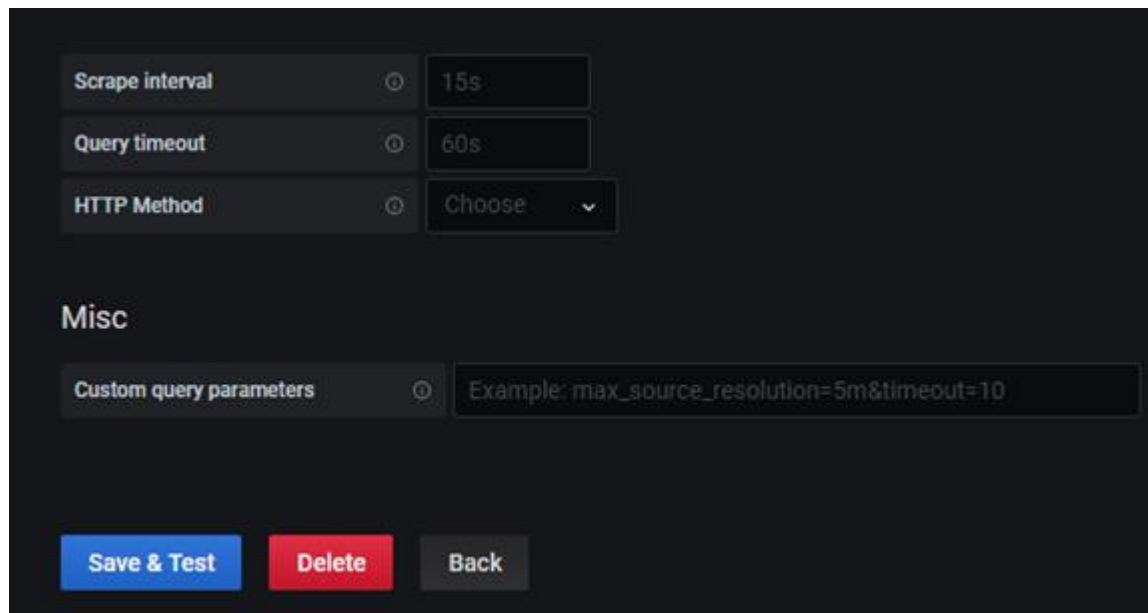
- iii. Hover over Prometheus option and click on **select** (as shown by the yellow arrow in the image below)



- iv. Choose a **name** for the data source and enter the **URL** of your running **Prometheus** instance.



- v. You can choose to change the values for **scrape interval**, **Query timeout** and **HTTP method**. You can as well choose to stick with the default values.



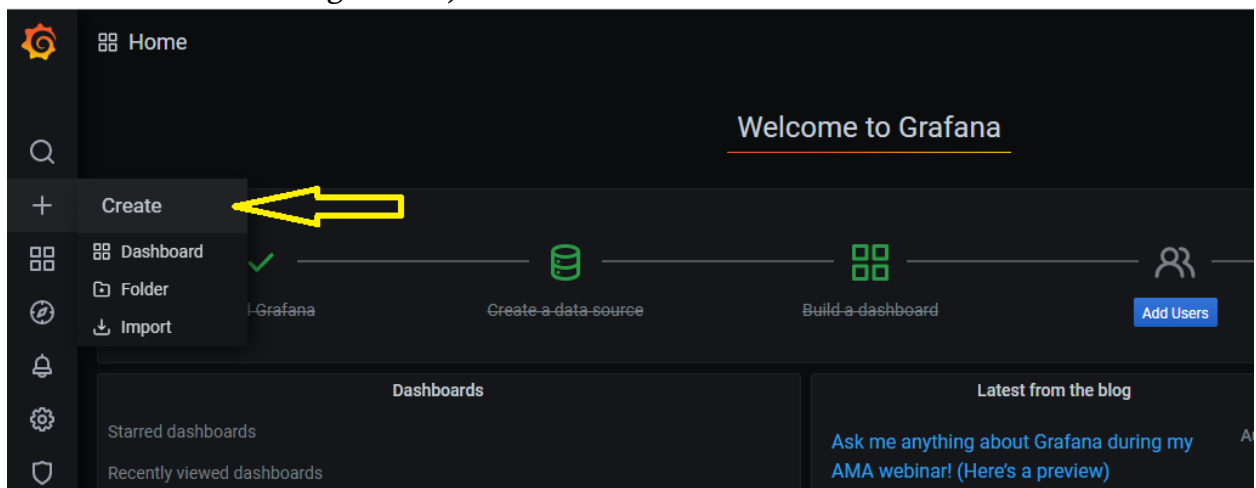
The image shows a configuration interface for a data source in Grafana. It has a dark theme. At the top, there are three settings: 'Scrape interval' set to '15s', 'Query timeout' set to '60s', and 'HTTP Method' set to 'Choose' with a dropdown arrow. Below these is a section titled 'Misc' containing a 'Custom query parameters' field with the example text 'Example: max\_source\_resolution=5m&timeout=10'. At the bottom, there are three buttons: 'Save & Test' (blue), 'Delete' (red), and 'Back' (grey).

Finally, Click on **Save & Test** button as seen in the image above.

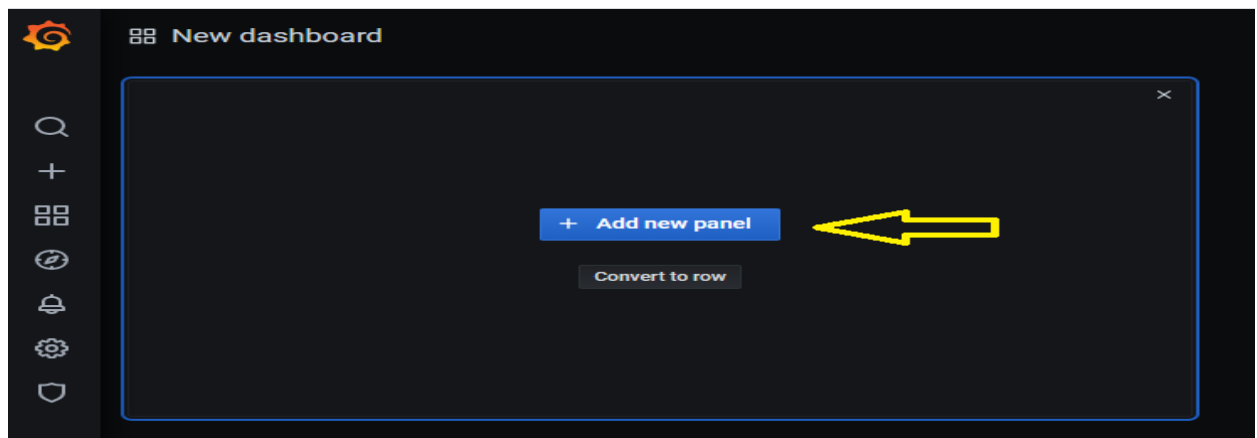
### - **Build Your own Dashboard**

After successfully adding Prometheus as data source to Grafana. You can now create your real time dashboards. Use the following steps to create your dashboard:

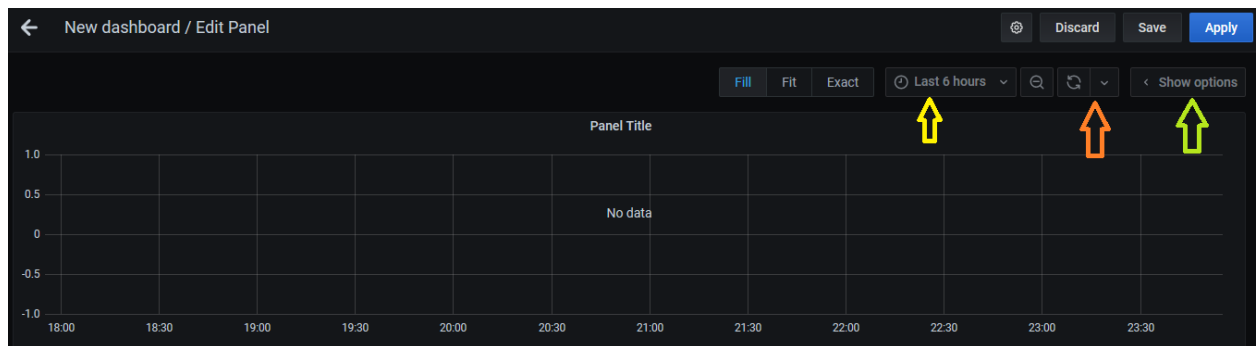
- i- From Grafana homepage, click on create “+” button (as shown by the yellow arrow in the image below)



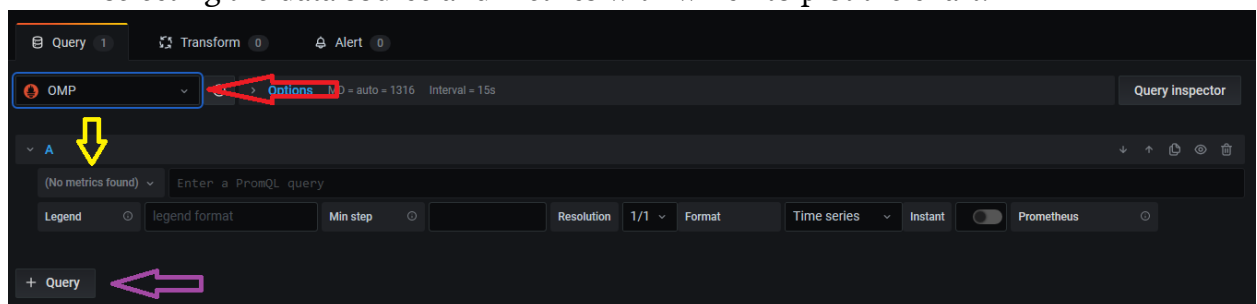
- ii- Click on Add new panel button (as shown by the yellow arrow in the image below)



- iii- This shows a panel with 2 sections. The top section as seen in the image below contains the chart display area and options for controlling the chart displayed.
- The yellow arrow shows a time duration of metrics to plot on the chart.
  - The orange arrow shows the time interval for refreshing the chart
  - The light green arrow points to a button that display more options.



- iv- The image below shows the lower section of the panel. This section allows for selecting the data source and metrics with which to plot the chart.



- The red arrow point to the name of the data source to use. In this case, the Prometheus data source name is OMP.

- The yellow arrow point to the metric selection panel. No metric have been selected.
- The purple arrow points to the query button which lets users add more metrics for plotting the chart.



### Example of Grafana Panel for CPU Total Utilization

#### - Zebra's Grafana Metrics

Zebra provides 4 types of metrics based on a unique prefix system:

##### 1. TOU\_Prefix

Metrics with **TOU\_** Prefix contains values of **Total utilization** of an lpar. E.g. **TOU\_VIRPT** is a metric containing numeric value of **VIRPT** lpar Total Utilization.

##### 2. EFU\_Prefix

Metrics with **EFU\_** Prefix contains values of Effective utilization of an lpar. E.g. **EFU\_VIRPT** is a metric containing numeric value of **VIRPT** lpar Effective Utilization.

##### 3. MSU\_Prefix

Metric with **MSU\_** Prefix contains System's MSU value E.g. **MSU\_VIRPT**.

##### 4. VC\_Prefix

Metrics with **VC\_** Prefix contains SYSCPUVC (Percentage of Maximum general purpose processor capacity spent on behalf of a group/class) value. E.g. **VC\_TSO** is a metric containing numeric value of **TSO** Group/Class.

# OMP Mentorship Program 2020: Project Zebra

---

## App Queries

### 1. Monitor 3

- CPC Report
  - a. <http://localhost:3090/rmfm3/?report=CPC>
  - b. <http://localhost:3090/rmfm3/?report=CPC&parm=CPCHCMSU>
  - c. <http://localhost:3090/rmfm3/?report=CPC&parm=ALL>
  - d. [http://localhost:3090/rmfm3/?report=CPC&lpar\\_parms=ALL\\_CP](http://localhost:3090/rmfm3/?report=CPC&lpar_parms=ALL_CP)
  - e. [http://localhost:3090/rmfm3/?report=CPC&lpar\\_parms=VIRPT](http://localhost:3090/rmfm3/?report=CPC&lpar_parms=VIRPT)
  - f. [http://localhost:3090/rmfm3/?report=CPC&lpar\\_parms=VIRPT&parm=CPCHLMSU](http://localhost:3090/rmfm3/?report=CPC&lpar_parms=VIRPT&parm=CPCHLMSU)
- RMF3 files
  - g. <http://localhost:3090/rmfm3?filename=SYSINFO>
  - h. <http://localhost:3090/rmfm3?filename=USAGE>
- USAGE and PROC Reports
  - i. <http://localhost:3090/rmfm3?report=PROC>
  - j. <http://localhost:3090/rmfm3?report=PROC&job=SDSFAUX>
  - k. [http://localhost:3090/rmfm3?report=PROC&job=ALL\\_JOBS](http://localhost:3090/rmfm3?report=PROC&job=ALL_JOBS)
  - l. <http://localhost:3090/rmfm3?report=PROC&parm=PRCPSVCL>

### 2. Post Processor

- Workload
  - a. <http://localhost:3090/rmfpp?report=WLMGL&date=20200731,20200731>
  - b. <http://localhost:3090/rmfpp?report=WLMGL&date=20200731,20200731&SvcCls=STCHI>
  - c. <http://localhost:3090/rmfpp?report=WLMGL&date=20200731,20200731&SvcCls=STCHIGH&Time=05.30.00>
  - d. <http://localhost:3090/rmfpp?report=WLMGL&date=20200731,20200731&SvcCls=STCHIGH&duration=05.30.00,09.30.00>
  - e. <http://localhost:3090/rmfpp?report=WLMGL&date=20200731,20200731&Wlkd=TSO>
  - f. <http://localhost:3090/rmfpp?report=WLMGL&date=20200731,20200731&Wlkd=TSO&Time=05.30.00>
  - g. <http://localhost:3090/rmfpp?report=WLMGL&date=20200731,20200731&Wlkd=TSO&duration=04.00.00,07.30.00>
- CPU
  - h. <http://localhost:3090/rmfpp?report=CPU&date=20200731,20200731>

### 3. Static

- a. <http://localhost:3090/static?file=C:\Users\Salis\Desktop\rmfpp.xml&type=CPU>

# Contributor Documentation



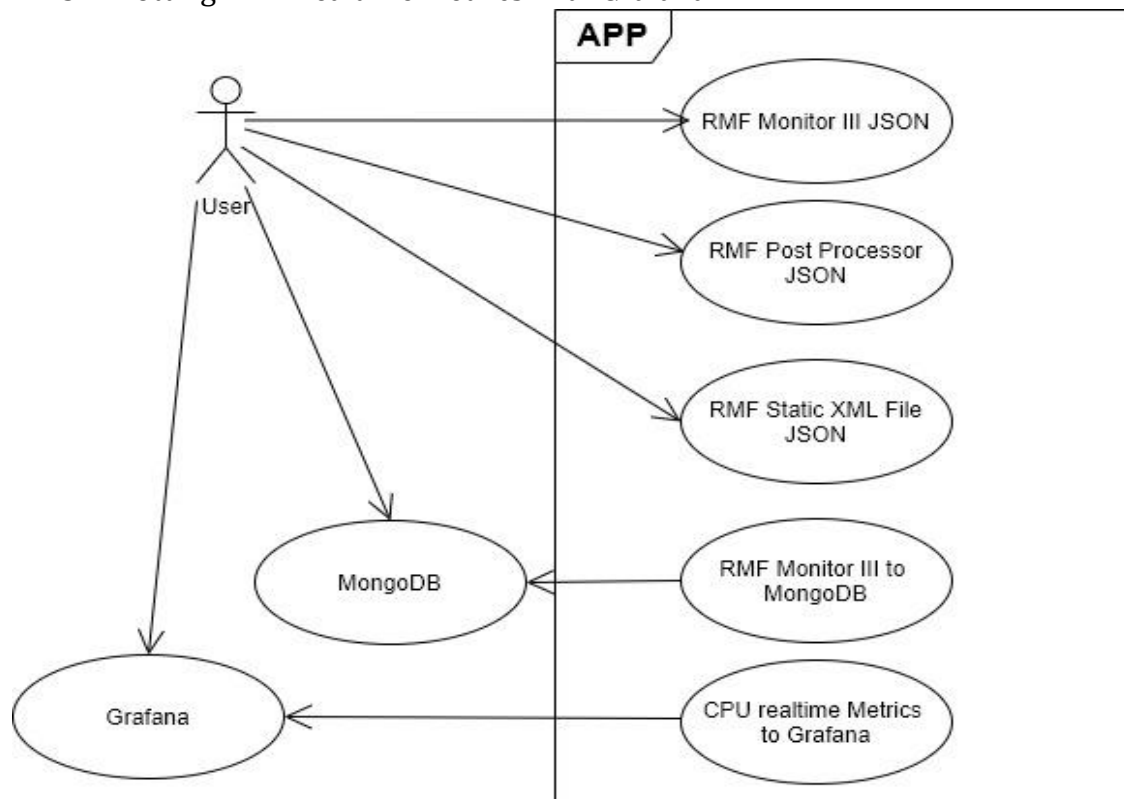
## Project Description

Zowe is a great systems operations tool. One of the systems programmers or performance analyzer's job is to decode SMF/RMF reports to check system's health. Having a generic parser for SMF datasets and/or RMF (or CMF) reports for Zowe would open various opportunities to create/re-use many open-source monitoring tools out there.

## Use Case

The Project have five (5) Use Cases:

1. Parsing RMF Monitor III Report to JSON
2. Parsing RMF Monitor I Report to JSON
3. Parsing RMF static XML File to JSON
4. Saving RMF Monitor III Report to MongoDB
5. Plotting RMF Realtime Metrics with Grafana



**Figure 1: Use Case Diagram**

## User Requirement

- The Project should create a Parsing Engine for SMF/RMF Report.
- Parsing Engine Output should be in JSON/CSV Format.
- Parsing Engine Output should be saved to NoSQL/Streaming database.
- The Project should provide Graphical System performance Monitoring.

## System Environment

- **Development:** Visual Studio Code.
- **Runtime Environment:** Node version 8.11.2 | NPM 5.6.0.
- **Database:** MongoDB.
- **Database Management:** MongoDB Compass.
- **Server:** Ubuntu 18 LTS.
- **Graphical tool:** Grafana.
- **Grafana Data Source:** Prometheus.
- **Unit Testing:** -.
- **Diagrams:** Draw.io

## Design Approach

The design approach used in the project is based on the following:

- **Data Flow Design**

SMF/RMF data is retrieved over the internet in XML format. The XML is then passed to the parsing engine and an output in JSON/CSV format is produced. The outputs for Realtime RMF reports are saved into a NoSQL database. Performance metrics from Realtime reports are exposed through an endpoint and Scraped Using Prometheus.
- **Architecture Design**

The project will follow a Three Layer Architecture so that the objects in the system as a whole can be organized to best separate concerns and prepare for distribution and reuse.
- **Graphical Tool**

Prometheus serves as a data source for building Realtime dashboards using a graphical tool. The project makes use of an open source graphical tool (Grafana) for creating real time monitoring dashboards from parsed RMF data.

## Design Pattern:

The Application Classes were factored into the following 3 layers:

### i. **The App-Server Layer**

This layer consist of the following components:

#### a. **HTTP GET Functions**

These functions send a HTTP GET Request to RMF DDS server for RMF Monitor III or RMF Post Processor data.

#### b. **Parser**

This consist of functions for parsing RMF Monitor III and RMF Post Processor data.

#### c. **Model**

This consist of Schema definitions for data to be saved into MongoDB.

**ii. The data Layer**

This layer contains the two data warehouse for the project:

**a. Prometheus**

Prometheus saves Realtime Performance metrics exposed by the parsing Engine.

**b. MongoDB**

MongoDB saves Realtime data output from the parsing Engine.

**iii. The Presentation Layer**

This layer consist of:

**a. Zowe API Catalog/Browser**

This tool displays JSON output from the parsing Engine.

**b. Grafana**

This tool displays dashboard from real time data output from the parsing Engine.

### **System Design Consideration**

**1. Directories**

**i. Root Directory**

This directory consist of:

- **App.js file**

- **Zconfig.json**

This file contains the configurations of the parsing engine in JSON format.

- **Cpurealtime.js**

This file contains function that expose real time CPU utilization metrics using prom-client library.

- **Mongo.js**

This file contains functions that save real time parsing engine output to MongoDB.

**ii. App Server Directory**

This directory Consist of:

- **Controllers folder**

This folder is made up of files that contain functions controlling the Events/Actions of the parsing Engine.

- **Parser folder**

This folder is made up of files containing functions for parsing real time and post processor data by the parsing Engine.

- **Models folder**

This folder contains Schema files for saving data to MongoDB as well as db.js file which contains functions for connecting to MongoDB.

- **Routes folder**

This folder contains files for mapping URL Endpoints to controller functions of the parsing Engine.

## 2. Exception Handling

Exception Handlers occur at the application level. Errors are displayed in JSON format.

### Architecture

- 1) **User (request):** User send a request to the zebra App using any of its recognized URL(s).
- 2) **GetRequest:** zebra app send a get request to RMF DDS server for post processor or Monitor III data depending on the URL specified by the user. DDS server returns an XML file.
- 3) **Parser:** The XML file returned is feed to RMF post processor parser, RMF monitor III parser or CPU utilization parser depending on the URL specified by the user. The parser returned a JSON.
- 4) **User (response):** User can view parsed RMF report using a browser or Zowe API Catalog.
- 5) **Prom-client:** The JSON returned by CPU utilization parser is used to create custom Prometheus metrics using prom-client library. The custom Prometheus metrics are exposed via /prommetric endpoint by the zebra app.

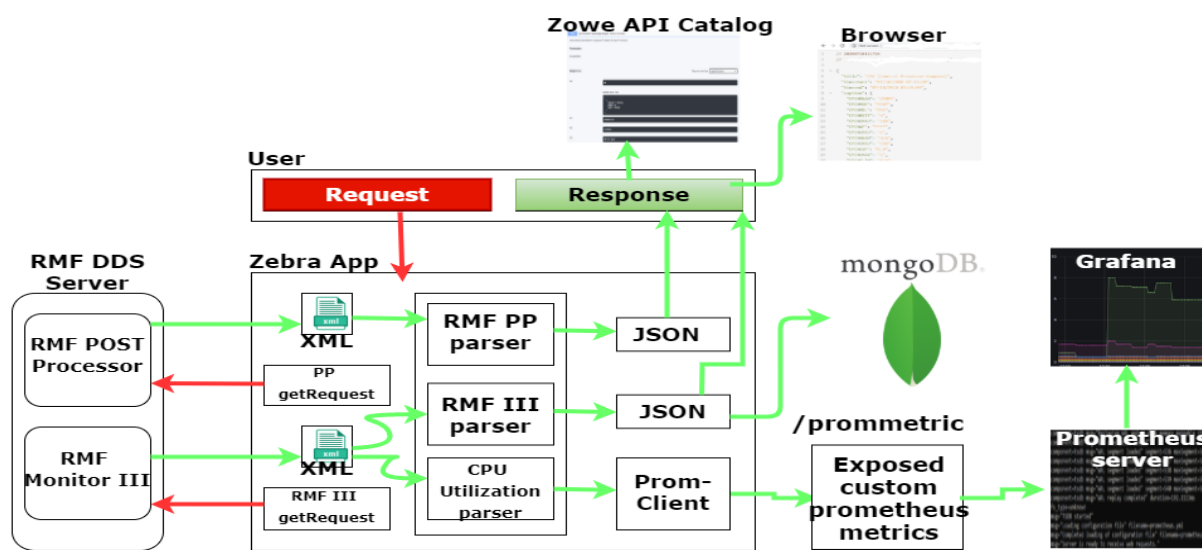
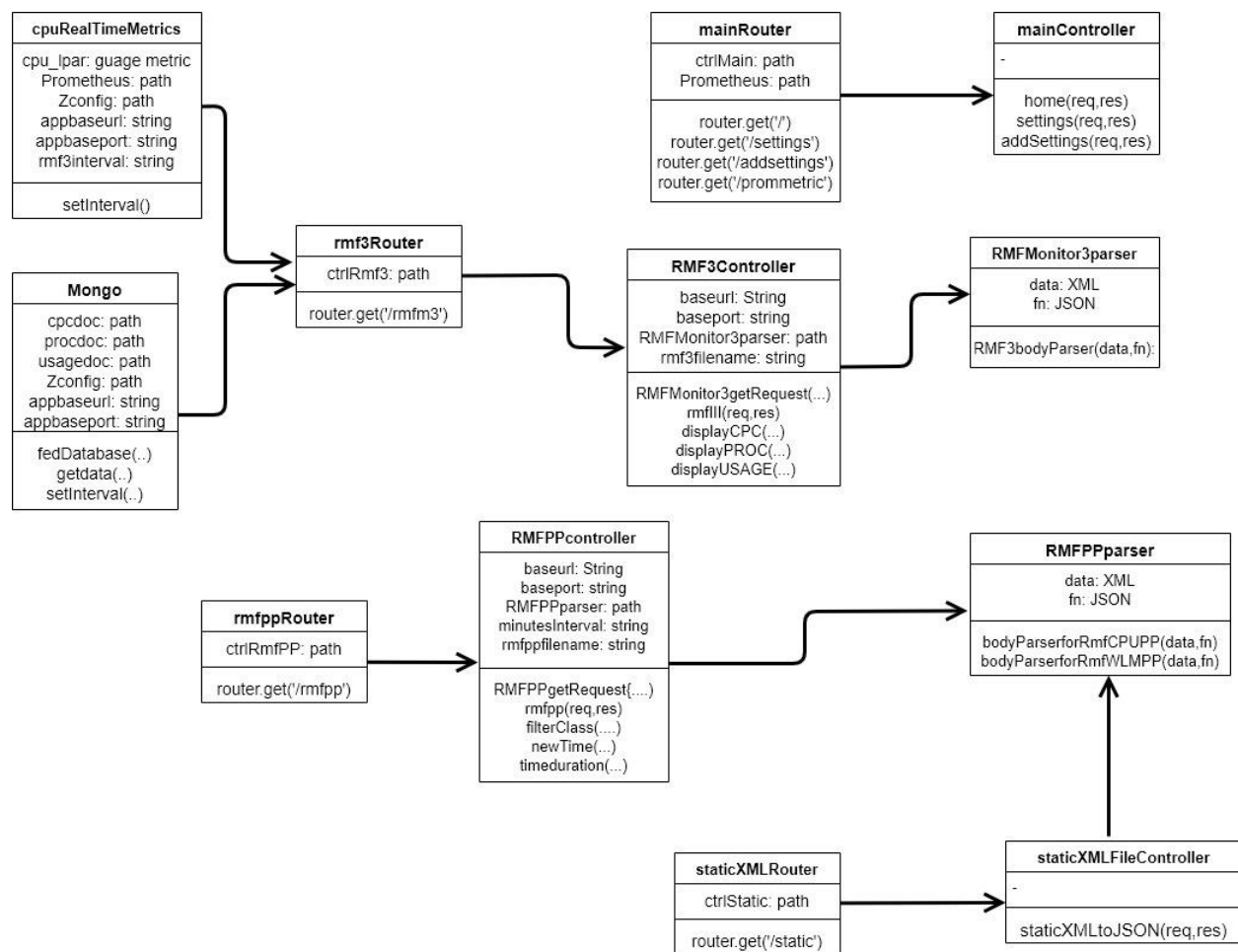


Figure 2: Project Architecture

- 6) **Prometheus server:** this server scrape custom Prometheus metrics from zebra app /prommetric endpoint
- 7) **Grafana:** Grafana dashboards are built by connecting Grafana to Prometheus server. This dashboard shows CPU utilization chart I Realtime.
- 8) **MongoDB:** Realtime data output from the parsing engine is saved to MongoDB database.

## Class Diagram



**Figure 3: Project Class Diagram**

The class diagram shows directional association between the project classes. Classes are represented in the project by files with ".js" as file extension.

Routers classes are the starting point of communication with other classes as they map incoming API request from users to controller and parser classes. The Router class names

contains the word “Router” at the end. These classes are responsible for recognizing the app’s endpoint.

Controller class names contains the word “Controller” at the end and consist of functions for:

- Sending GET Request to DDS server
- Sending XML to Parser
- Filtering parsed JSON based on user specified parameters in the URL
- Displaying the JSON response to User
- Adding Configuration settings to the App

Parser classes consist of functions for parsing XML to JSON. RMF monitor III parser has a single function which can parse all RMF III XML reports due to their format consistency. Monitor I parser (RMFPPparser) has two functions for parsing CPU post processor and Workload Postprocessor data.

**Mongo** and **cpuRealTimeMetrics** classes have a **setInterval()** function which makes them run continuously at an interval specified by the user in the app configuration. They both interact with monitor III router to retrieve a JSON.

**Mongo** class is responsible for retrieving real time **CPC**, **PROC** and **USAGE** monitor III reports and fed them into **MongoDB** while **cpuRealTimeMetrics** class is responsible for retrieving real time **CPC** report **JSON**, filtering the JSON for **Total utilization**, **MSU value** and **Effective Utilization values**. It then use prom client library to create custom gauge metric. The custom metrics are saved into a register. The metrics in the register are exposed via an Endpoint in the **mainRouter** class. The exposed metrics can then be scraped by Prometheus and real time charts can be plotted using Grafana.

### Activity Diagram

#### i. RMF Monitor III Activity Diagram

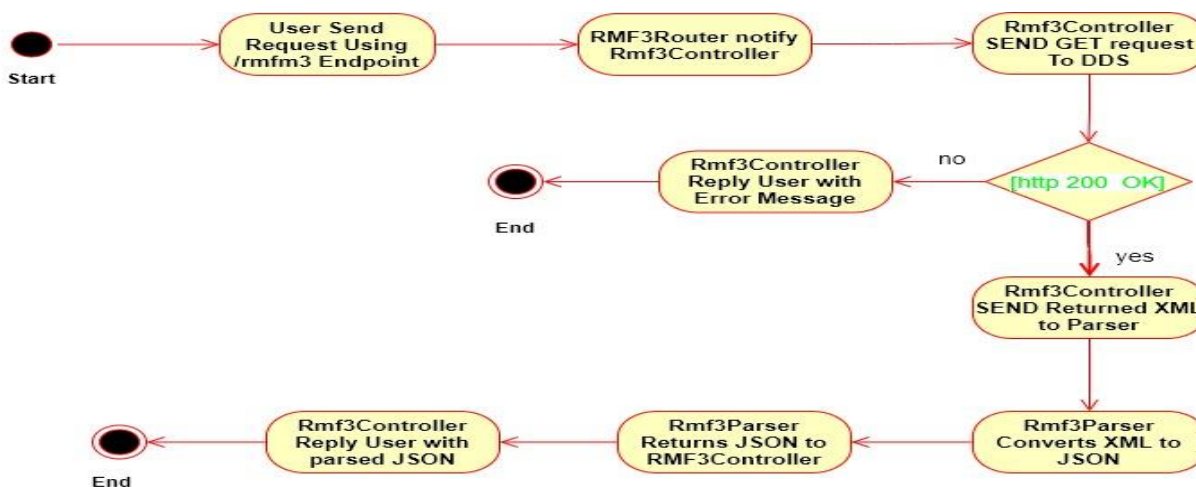
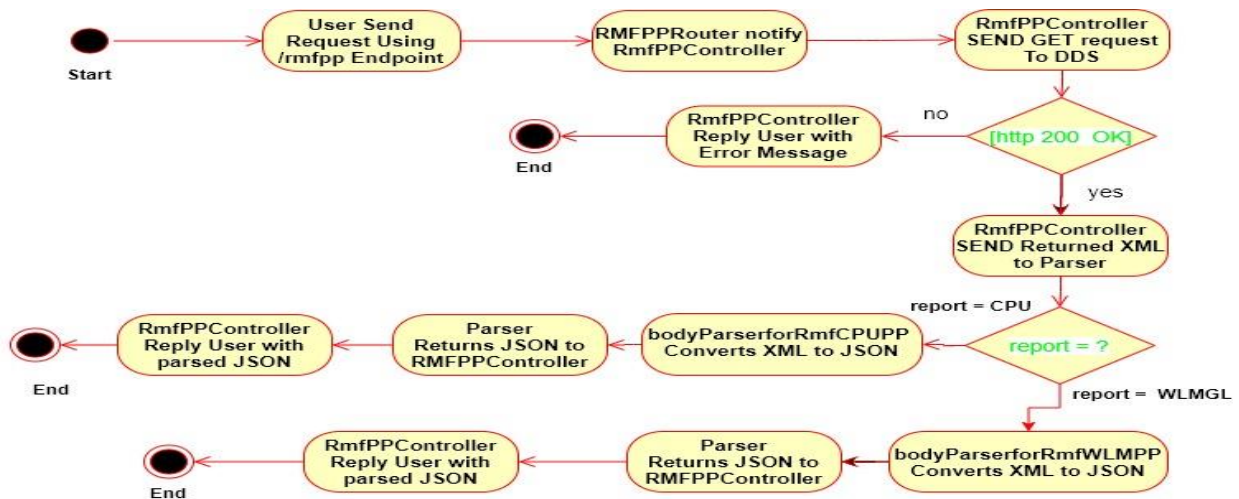


Figure 4: RMF 3 Activity Diagram

RMF Monitor III Activities starts when a User trigger a request using the /rmfm3 Endpoint, This leads to a series of activities involving RMF3Router, RMF3Controller and RMF3Parser. A Condition exist to check is the Request to DDS server is successful, this condition determines the data that get returned to RMF3Controller and finally to the User. In the End, the user receives a JSON Response containing parsed RMF III report or Error Message in case of a failed request to DDS.

### ii. RMF Post Processor Activity Diagram



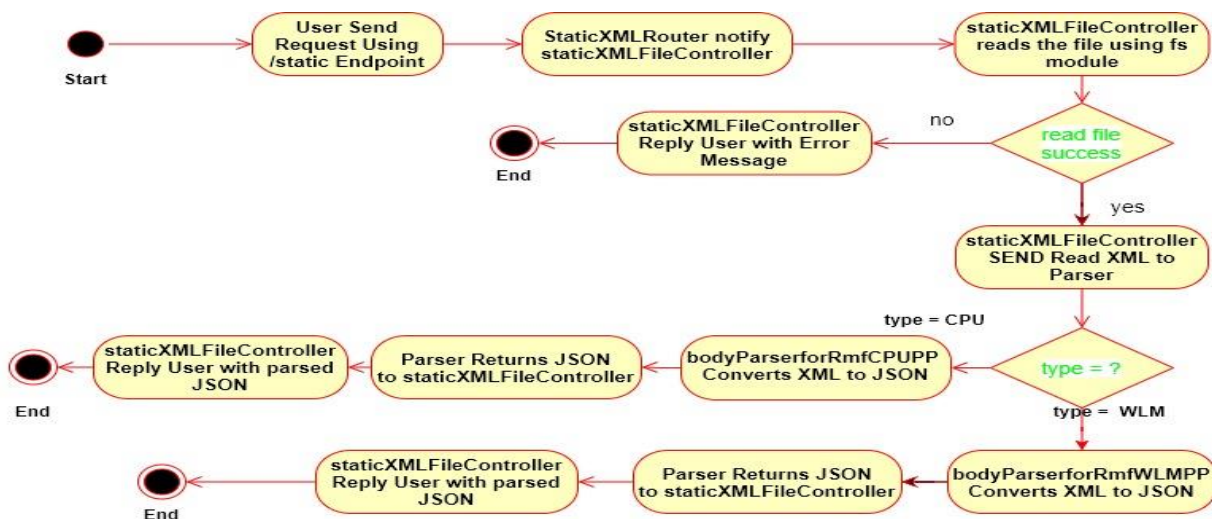
**Figure 5: RMF Post Processors Activity Diagram**

RMF Monitor I Activities starts when a User trigger a request using the /rmfpp Endpoint, This leads to a series of activities involving RMFPPRouter, RMFPPController and RMFPPParser. Two conditions exist in the activity flow of RMF post Processor. First is the condition that checks if the request to DDS server is successful. The Second one checks the value of the Report parameter specified by the user during a request. The value of the report parameter is used to determine the parser for the returned by the request to DDS.

### iii. Static XML File Activity Diagram

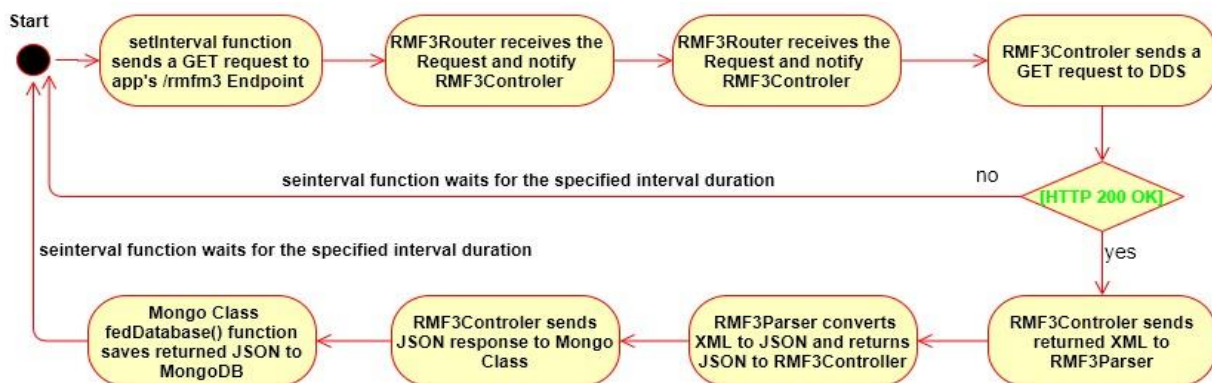
Static XML File to JSON Activities starts when a User trigger a request using the /static Endpoint, This leads to a series of activities involving staticXMLRouter, staticXMLFileController and RMFPPParser. A re-use of the RMFPPParser occurs here. Two conditions exist in the activity flow of static File to JSON as well. First is the condition that checks if reading the static file specified by the user in the URL's file (takes file path as value) parameter is successful. The Second one checks the value of the Report parameter specified by the user during a request. The value of the report parameter is used to determine the parser for the returned by the request to DDS.





**Figure 6: Static XML File Activity Diagram**

## iv. Mongo Activity Diagram



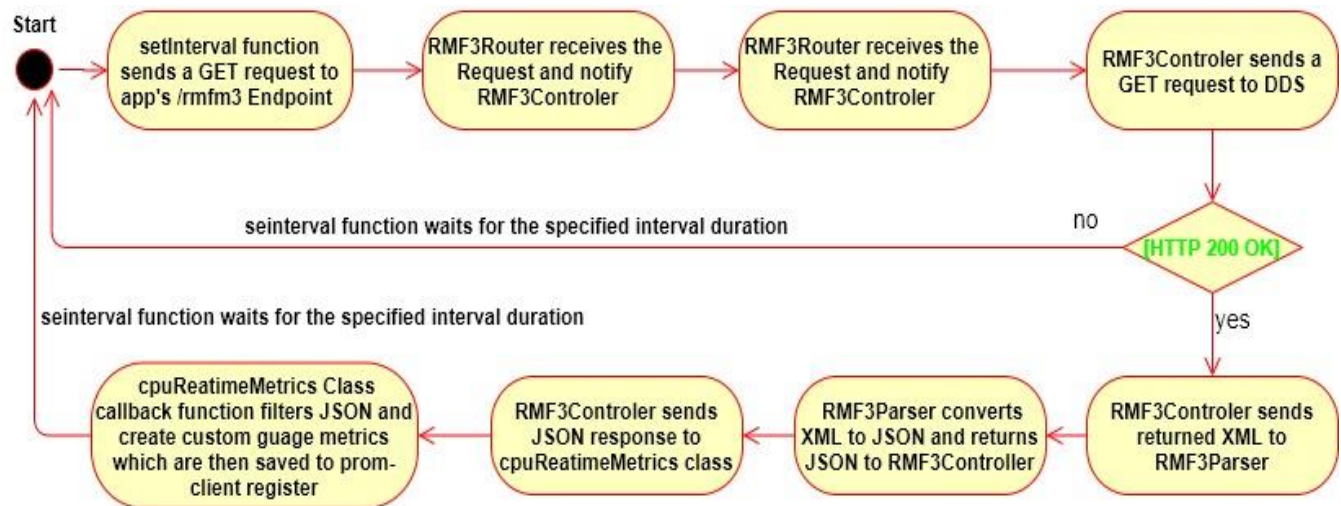
**Figure 7: Mongo Activity Diagram**

Activities of the Mongo class do not require user intervention. A setInterval function runs continuously at an interval specified in the Apps Configuration. This function serves as a trigger and make use of RMF III Activities to retrieve and store JSON into MongoDB.

## v. CPU Realtime Metrics Activity Diagram

Just like in Mongo Class, Activities of the CPU Realtime Metrics (cpuRealTimeMetrics) class do not need user intervention. The setInterval function triggers RMF III Activities which returns a JSON. Through a call back function, CPU Realtime Metrics class filters the JSON and create Prometheus custom gauge metrics for Effective utilization, MSU and Total utilization values. These Metrics are then saved to a prom-client library register.





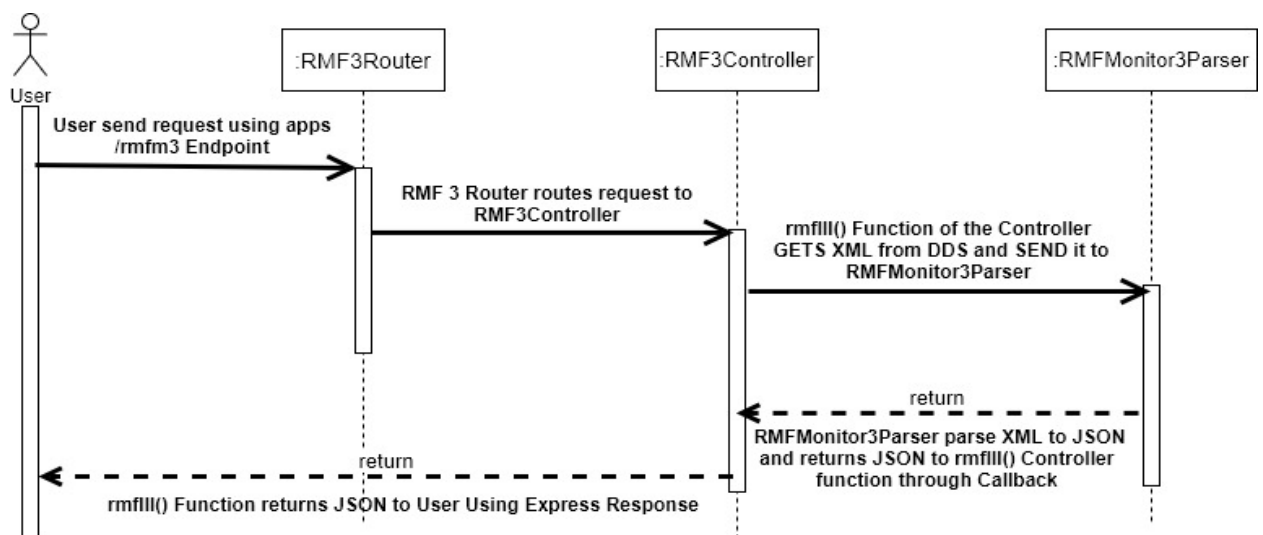
**Figure 8: CPU Realtime Metrics Activity Diagram**

## Sequence Diagram

### i. RMF Monitor III Sequence Diagram

RMF monitor III Service is triggered when user send a request using /rmfm3 Endpoint (e.g. localhost:3000/rmfm3?report=CPC).

RMF3Router receives the request and notify RMF3Controller. rmfIII() Function of the controller sends A HTTP API GET Request to DDS Server. This Returns an XML which is then sent to RMFMonitor3Parser. The parser then parse the XML into JSON and returns the JSON to the controllers rmfIII function. This JSON is finally sent to user through Express Response.

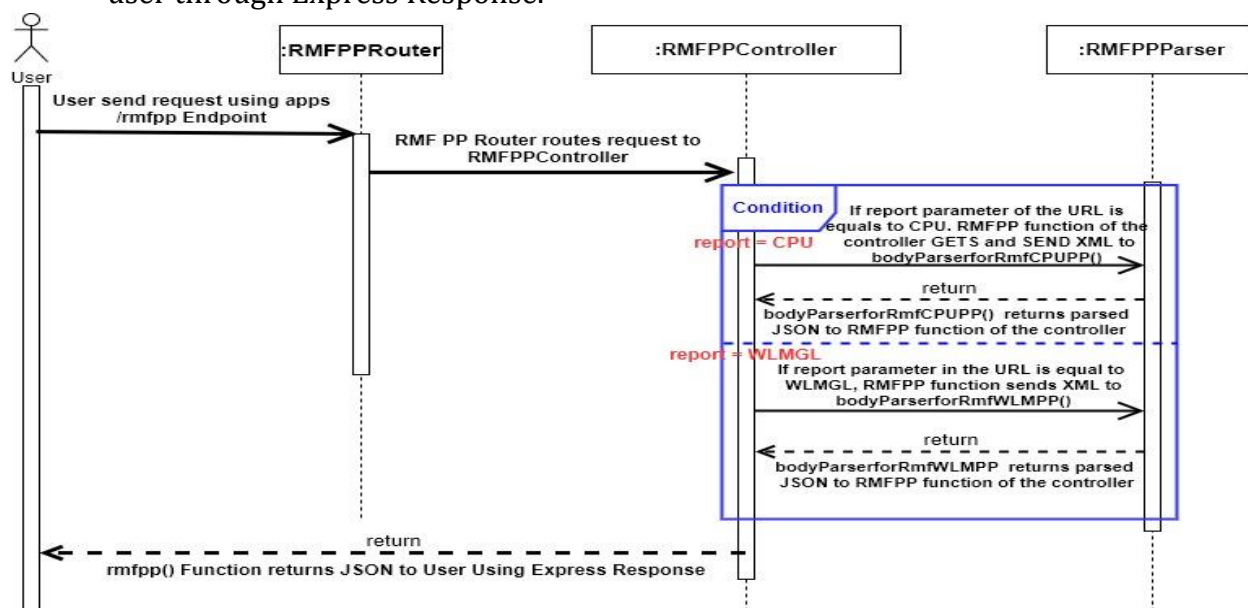


**Figure 9: RMF 3 Sequence Diagram**

## ii. RMF Post Processor Sequence Diagram

RMF pp service is also triggered when a user send a request using /rmfpp Endpoint (e.g. localhost:3000/rmfpp?report=CPU).

RMFPPRouter receives the request and notify the controller. Rmfpp() function of the controller evaluates the value of report parameter(CPU or WLMGL). It then sends a HTTP GET request to DDS server. The returned XML is send to the appropriate parser(bodyParserforRmfWLMPP or bodyParserforRmfCPUPP) based on the value of the report URL parameter. The parser then parse the XML into JSON and returns the JSON to the controllers rmfpp function. This JSON is finally sent to user through Express Response.



**Figure 10: RMF PP Sequence Diagram**

## iii. Static XML File

The static XML file service is triggered when a user sends a request using the apps /static Endpoint (e.g. localhost:3000/static?file=/home/salis).

staticXMLRouter receives the request and notify the controller. staticXMLtoJSON () function of the controller evaluates the value of type parameter(CPU or WLM). The controller re-use rmfpparser for parsing XML to JSON. The XML file specified in the URL file (takes file path) parameter is send to the appropriate parser (bodyParserforRmfWLMPP or bodyParserforRmfCPUPP) based on the value of the type URL parameter. The parser then parse the XML into JSON and returns the JSON to the controllers staticXMLtoJSON function. This JSON is finally sent to user through Express Response.

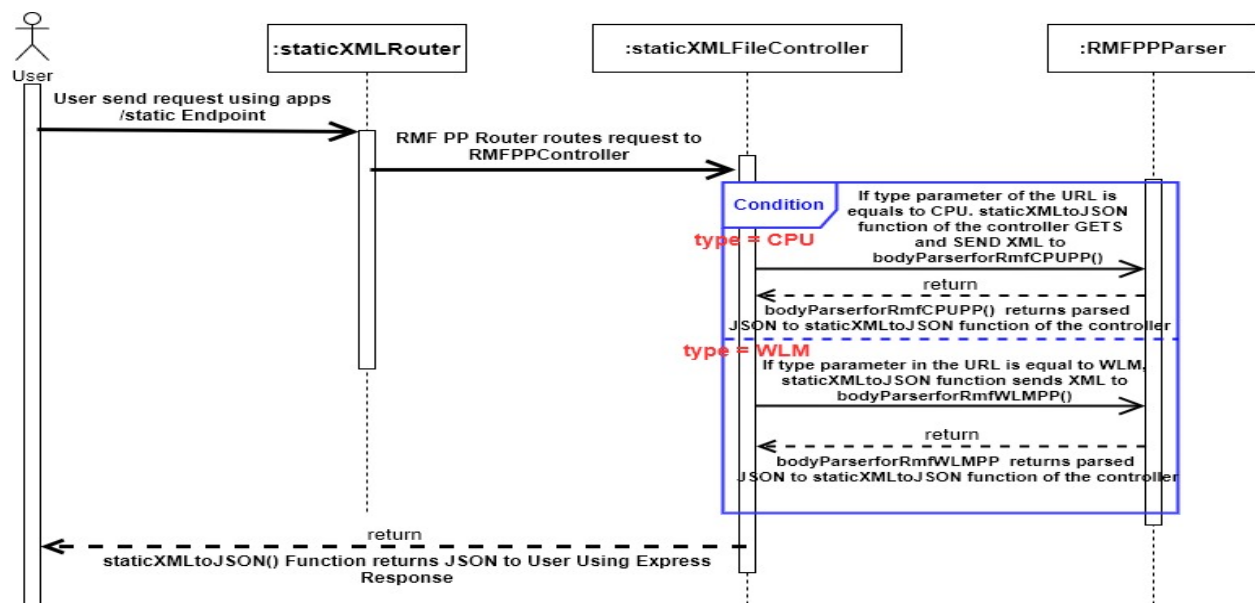


Figure 11: Static XML File Sequence Diagram

## iv. Mongo

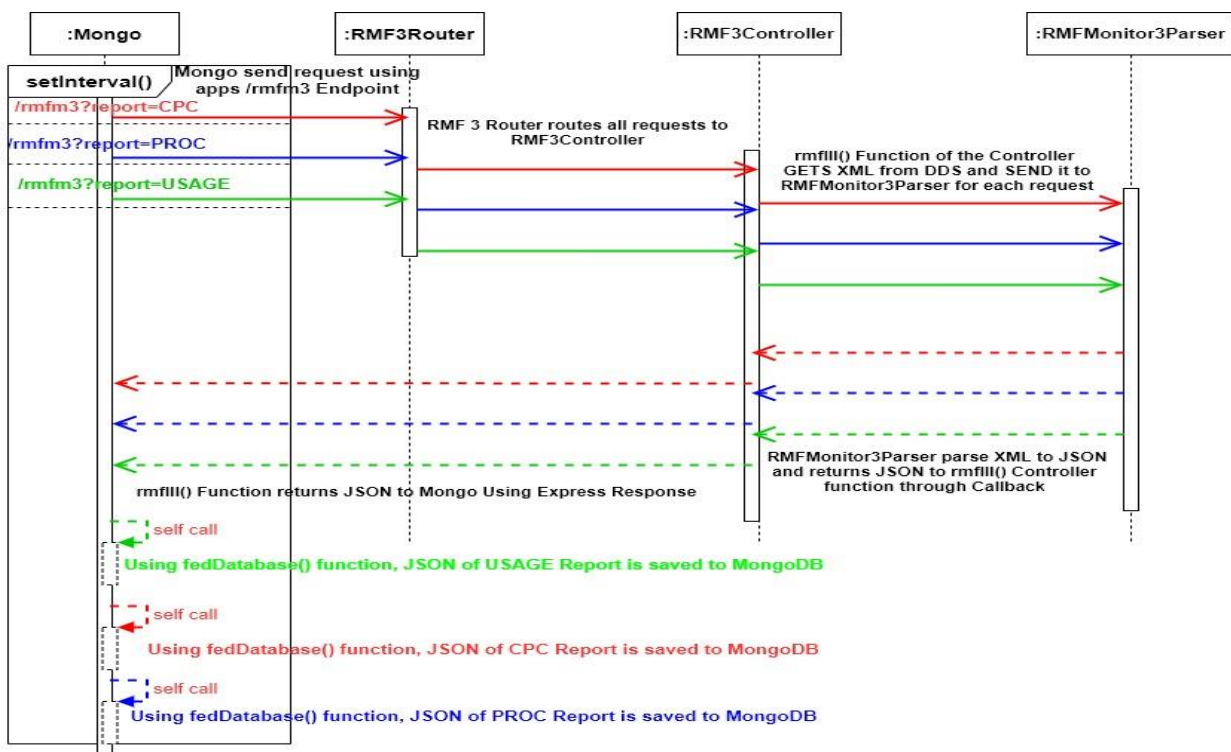


Figure 12: Mongo Sequence Diagram

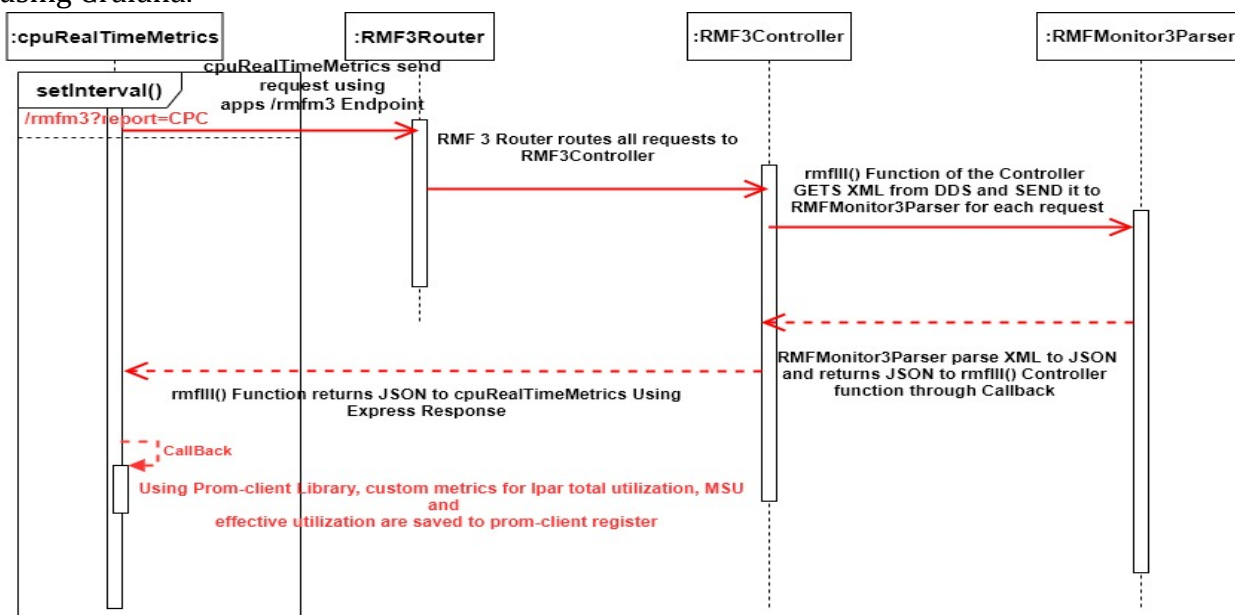
Unlike the first three services, Mongo service does not require a trigger from user. The class contains a `setInterval` function that runs continuously based on the interval specified in the App configuration.

When `setInterval` function is activated, it send three request for CPC, PROC and USAGE JSON report by triggering the apps `/rmfm3` endpoint. For each report type (CPC, PROC And USAGE), RMF3Router sends a notification to RMF3Controller. `rmfIII` function of the controller sends a HTTP GET request to DDS for each report and the returned XML is sent to RMFMonitor3Parser. The parser parse each report and return 3 JSON to the `rmfIII` controller function. All 3 JSON's are sent to Mongo class.

Using the `fedDatabase()` function of the Mongo class, The 3 JSON response are saved to the appropriate documents in MongoDB.

### v. CPU Realtime Metrics

The CPU real-time class also does not need user intervention. It's responsible for creating and saving Prometheus Custom metrics that can be used to plot real-time graphs using Grafana.



**Figure 13: CPU Realtime Metrics Sequence Diagram**

The `setInterval` function of the CPU real-time metrics class sends a request to the app `/rmfm3` for CPC JSON. The process of retrieving the JSON is similar to Mongo class. Data flow from router, controller and parser of the RMF Monitor III service. The JSON returned to CPU real-time metrics class is processed using a callback function. The JSON is filtered for Effective utilization, MSU and Total Utilization Values. These values are used to create custom gauge metrics using `prom-client` library. These metrics are saved into a register.

## Entity Diagram

The Project make use of MongoDB. Mongoose library was used to structure the data. In MongoDB each entry in a database is called a document. In MongoDB a collection of documents is called a collection (think “table” if you’re used to relational databases). In Mongoose the definition of a document is called a schema.

### 1. CPC Activity

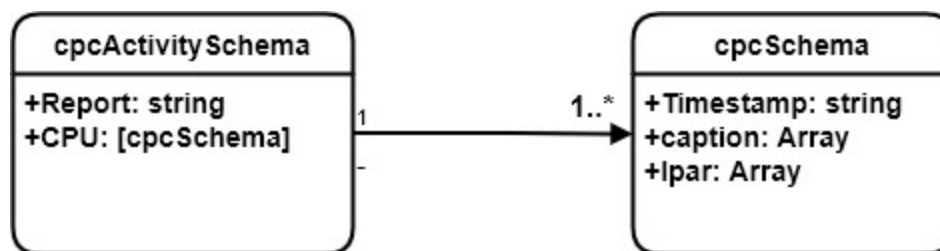


Figure 14: CPC Schema Definition

For CPC report, the main document definition is named cpcActivitySchema. The main document contains two fields:

- **Report:**  
Report is a string that represent the Document title
- **CPU**  
CPU field represent (is populated by) a subdocument. The subdocument is named cpcSchema and have 3 fields:
  - **Timestamp**  
This is a string that represent the CPC report **timestamp**.
  - **Caption**  
This is an array and is populated by CPC report **caption** key.
  - **Lpar**  
This is an array and is populated by an array of the CPC report **table** key

The Mongo Class Create CPC Activity document only once and continue to populate CPU field of the main document by the subdocument.

### 2. PROC Activity

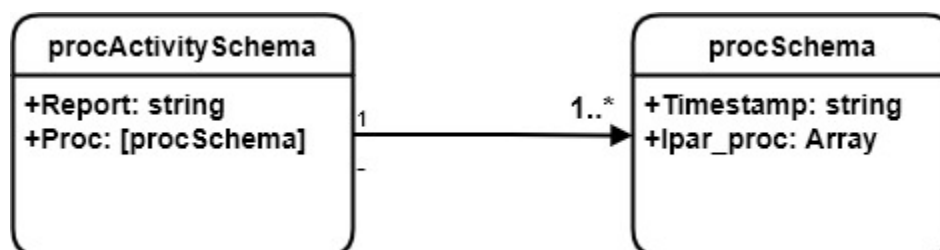


Figure 15: PROC Schema Definition

For PROC report, the main document definition is named procActivitySchema. The main document contains two fields:

- **Report:**  
Report is a string that represent the Document title
- **Proc**  
Proc field represent (is populated by) a subdocument. The subdocument is named procSchema and have 2 fields:
  - **Timestamp**  
This is a string that represent the PROC report **timestamp**.
  - **Lpar\_proc**  
This is an array and is populated by an array of PROC report **table** key

The Mongo Class Create PROC Activity document only once and continue to populate Proc field of the main document by the subdocument.

### 3. USAGE Activity

For USAGE report, the main document definition is named usageActivitySchema. The main document contains two fields:

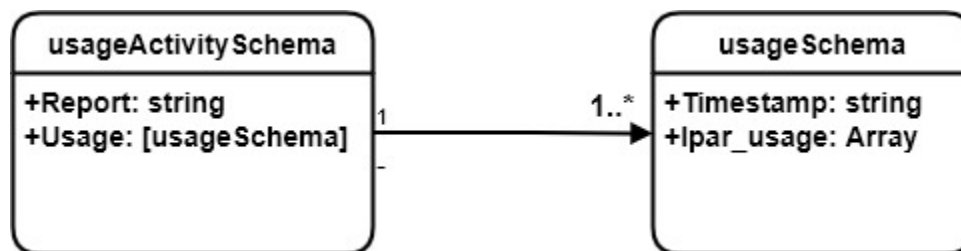


Figure 16: Usage Schema Definition

- **Report:**  
Report is a string that represent the Document title
- **Usage**  
Usage field represent (is populated by) a subdocument. The subdocument is named usageSchema and have 2 fields:
  - **Timestamp**  
This is a string that represent the USAGE report **timestamp**.
  - **Lpar\_usage**  
This is an array and is populated by an array of USAGE report **table** key

The Mongo Class Create USAGE Activity document only once and continue to populate Usage field of the main document by the subdocument.