Workflow Engine Report

Adarsh Das

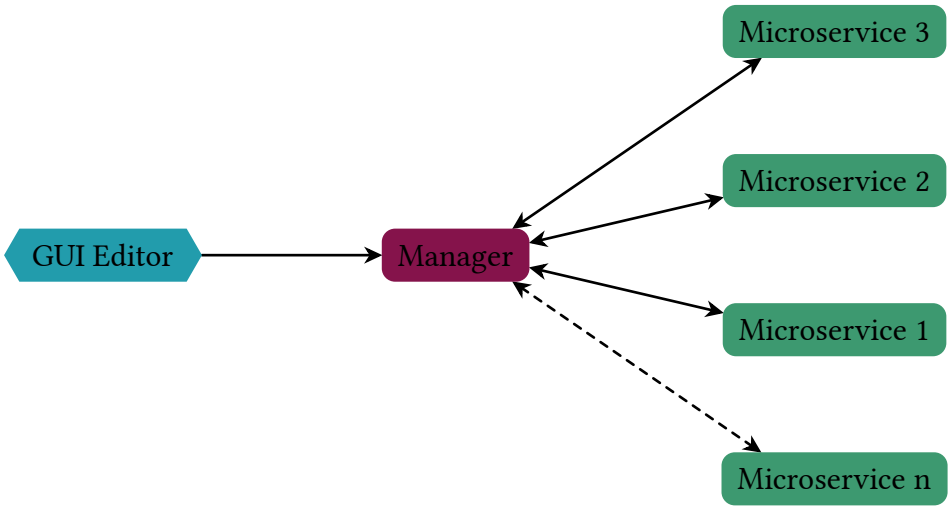
This document is a report on the development of a workflow engine. The report covers the design, implementation, and testing of the engine. The engine is designed to be highly customizable and can be used in a variety of applications.

July 31, 2024

This project was developed under the guidance of Mr. Naresh Krishnamoorthy as part of the internship at Dover Corporation.

Contents

1. The Basic structure of the engine



The workflow engine is designed to be highly modular and customizable. It consists of the follwing components:

- Manager: The manager is responsible for coordinating the flow of data between the microservices.
- Microservices: The microservices are responsible for performing specific tasks.
- GUI Editor: The GUI editor allows users to create and edit workflows.

> **Note**
>
> Communication between the manager, microservices, and the editor is facilitated using Kafka.

2. In-depth look at the components

2.1. GUI Editor

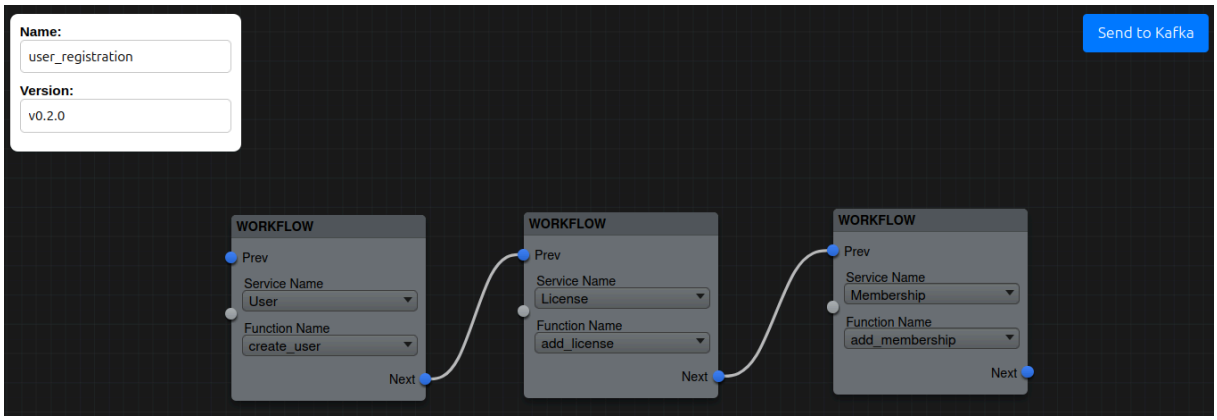


Figure 1: Screenshot of the GUI editor

The GUI editor utilizes the `Flume` Javascript library which enables it to drag and drop components to create workflows. The editor also allows user the edit the metadata of the components, and finally allows to publish the kafka, where the manager can consume to perform the relevant edits.

2.2. Manager

The manager is responsible for coordinating the flow of data between the microservices. Currently it listens to the following topics:

- `toManager` : This topic is used by the microservices to send data to the manager.
- `fromManager` : This topic is used by the manager to send data to the microservices.
- `toManagerEdits` : This topic is used by the editor to send data to the manager.

We can explain each topic logic in detail:

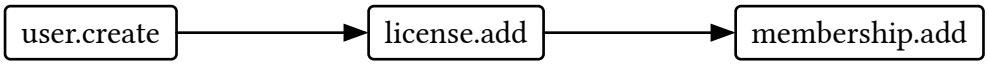
2.2.1. `toManager`

This topic is used by the microservices to send data to the manager. The data is sent in the form of a RON object. An example of the data sent is:

```
ToManager(
    uuid: "585f6cf4-bf0e-4852-8ca3-c6866ce8a752",
    name: "user_registration",
    version: "v0.1.0",
    process: Process(
        service: "user",
        function: "create"
    ),
    status: Success,
    schema: "v0.1.0",
    data: "adarsh"
)
```

> **Tip**
>
> Here the `Process` struct is defined as follows in the code
>
> ```rust
> pub type Service = String; // aliased as a String
> pub type Function = String; // aliased as a String
>
> #[derive(Debug, PartialEq, Eq, Hash, Clone, Serialize, Deserialize)]
> pub struct Process {
>     pub service: Service,
>     pub function: Function,
> }
> ```
>
> And Success is part of an enum defined as follows:
>
> ```rust
> #[derive(Serialize, Deserialize, Debug)]
> pub enum Status {
>     Success,
>     InProgress,
>     Failed,
> }
> ```

Once we receive the data, we process it based on the relevant workflow and send the data to the relevant microservice. For example, if we have the following workflow:



Then the manager will send the data to the `license` microservice which executes the `add` function.

Now what if the status is `Failed` ?, in that case we employ a rollback mechanism, similar to the way we have in SAGA pattern [1]. For example if we get the following data:

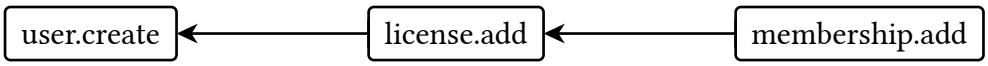
```
ToManager(
    uuid: "585f6cf4-bf0e-4852-8ca3-c6866ce8a752",
    name: "user_registration",
    version: "v0.1.0",
    process: Process(
        service: "membership",
        function: "add"
    ),
    status: Failed,
    schema: "v0.1.0",
    data: "adarsh"
)
```

we can refer to the rollback dictionary, i.e., a dictionary of processes and their corresponding rollback processes.

> **Note**
>
> The rollback for a process can be one or more processes.

```
Process(service:"legal",function:"update")  : Process(service:"legal",function:"revert"),
Process(service:"user",function:"create")   : Process(service:"user",function:"delete"),
Process(service:"membership",function:"add"): Process(service:"membership",function:"remove"),
Process(service:"license",function:"add")   : Process(service:"license",function:"remove")
```

and the anti-workflow, which is same as our workflow but with connections reversed.



Thus as `membership.add` failed, we will use our anti workflow and see that the previous process was `license.add` and thus we will call the `license.remove` function of the `license` microservice. Then we will keep on following the anti-workflow until we reach the `user.create` function. Finally here we will call the `user.delete` function of the `user` microservice.

> **Note**
>
> The `toManager` topic is read using a group id. This enables the manager to read the data from the microservices in a round-robin fashion.

2.2.2. `fromManager`

This topic is used by the manager to send data back to the microservices. An example output is as follows,

```
FromManager(
    uuid: "585f6cf4-bf0e-4852-8ca3-c6866ce8a752",
    name: "user_registration",
    version: "v0.1.0",
    process: Process(
        service: "membership",
        function: "add"
    ),
    schema: "v0.1.0",
    data: "adarsh"
)
```

This signals the reading microservice that this data is for the `membership.add` function.

2.2.3. `toManagerEdits`

This topic is used by the editor to send data to the manager. An example of the data sent is:

```
ToManagerEdits(
    name: "user_registration",
    version: "v0.1.0",
    schema: "v0.1.0",
    workflow: {
```

```
6           Process(service:"license",function:"add"):[
7               Process(service:"legal",function:"update")
8           ],
9           Process(service:"user",function:"create"):[
10              Process(service:"license",function:"add"),
11              Process(service:"membership",function:"add")
12          ]
13      }
14  )
```

This data contains the workflow for the required process. The manager can then use this data to update the workflow in it local storage.

> **Note**
>
> Unlike `toManager` and `fromManager`, the `toManagerEdits` topic is not read using a group id, as we require edits to be propogated to all the managers.

3. Conclusion

In conclusion, the workflow engine is a highly modular and customizable system designed to streamline and automate complex processes. It consists of a manager that coordinates data flow between various microservices, each performing specific tasks. The GUI editor enhances user experience by allowing easy creation and modification of workflows. Communication between components is efficiently managed using Kafka, ensuring seamless data exchange and process execution. This robust design makes the workflow engine adaptable to a wide range of applications, providing a solid foundation for efficient workflow management.

Bibliography

[1] “Pattern: Saga.” Accessed: Jul. 30, 2024. [Online]. Available: https://microservices.io/patterns/data/saga.html°

Index of Figures