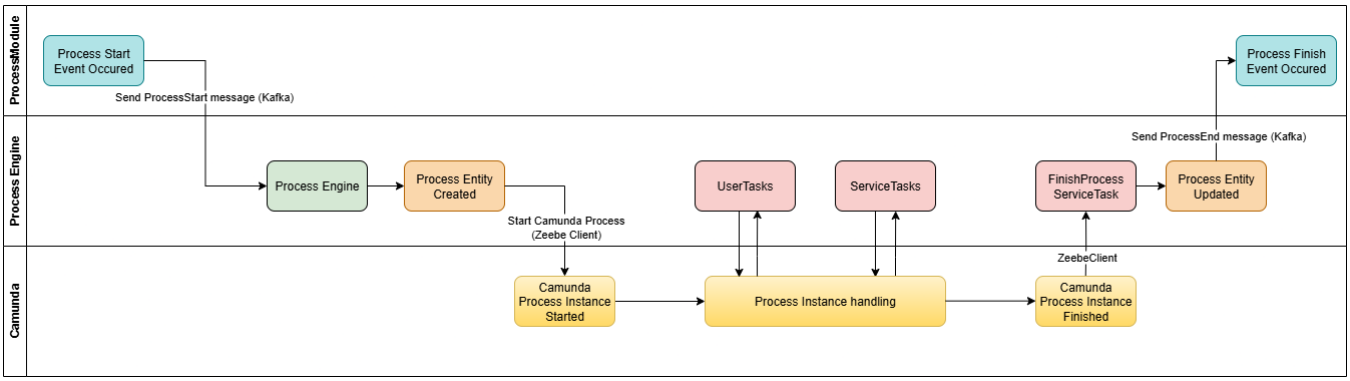
Camunda process life cycle

Falcon application will create some objects that will become managed by Camunda instance via Process instances.

ProcessModule will be responsible for creating such objects. It will also trigger creating ProcessInstance on Camunda engine. To do that it will use ProcessEngine, that will spawn ProcessInstance. Process flow will be continued on Camunda. Camunda will orchestrate the process and will invoke some ServiceTasks / UserTasks during process progress. When Camunda Process instance hit the Process End Event the process will job called "ProcessEndJob" (execution listerner). This job (in fact- service task) will be hadled by ProcessEngine. ProcessEngine will publish the message to ProcessModule to let know, the process has been finished. ProcessModule will consume this message to update it's object.

Idea:

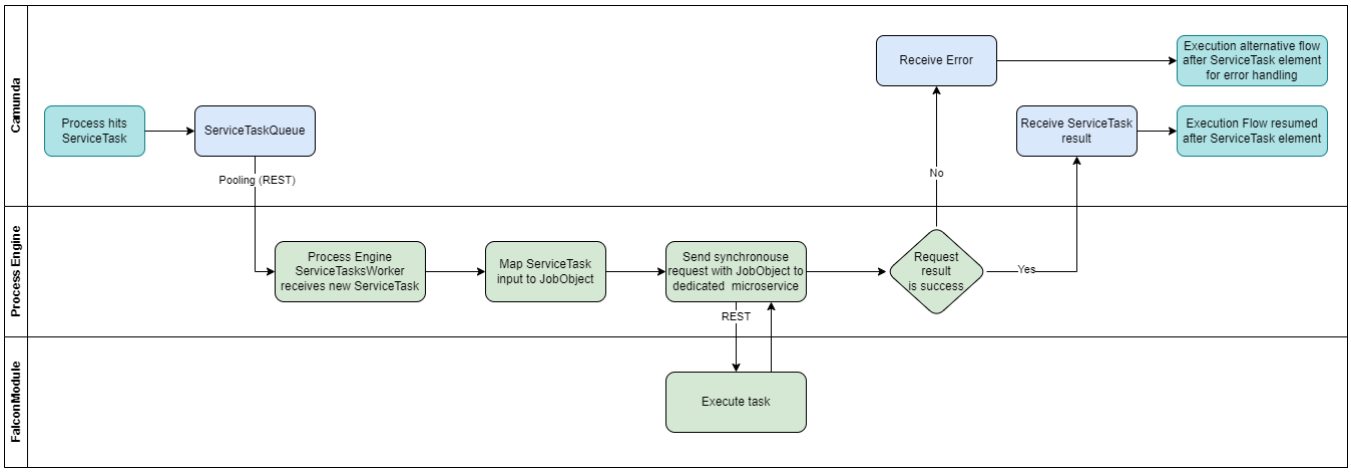
1. ProcessModule creates the object (e.g. ProcessObject). CreditApplication has some BusinessIdentifier, Status etc. properties
2. In order to run process instance via Camunda, ProcessModule generates the message to Kafka.
3. ProcessEngine consumes the message and creates the process instance on camunda. Based on the context data, process engine picks equivalent process definitinon and passes necessary data. ProcessEngine persists some metadata like business identifier, process status etc.
4. Camunda is responsible for orchestrating the process. It may call some external code via ServiceTasks or UserTasks multiple times. These external code may affect some Falcon objects.
5. At some point the process hits the end event. Camunda may start additional "ServiceTask" via execution listener. The job handler updates the ProcessEngine object and sets it's status to complete. Kafka message is being published to update ProcessModule object state.



Process with Service Task (synchronous communication)

1. The process is started in Camunda.
2. The **ProcessEngine** detects a **Service Task** that should be handled synchronously.
3. The **ProcessEngine** immediately delegates the task to the appropriate microservice using REST and waits for the result.
4. The microservice executes the business logic and returns the result to the **ProcessEngine**.
5. The **ProcessEngine** sends the completion information to Camunda, allowing the process to continue to the next step.

This approach is used when the **Service Task** can be completed quickly and does not require asynchronous processing. If a microservice is not available, the **ProcessEngine** will use a retry policy. If the **ProcessEngine** cannot connect to the microservice after the retries, it will return an error to Camunda. The process flow in Camunda should include an additional path to handle such errors.

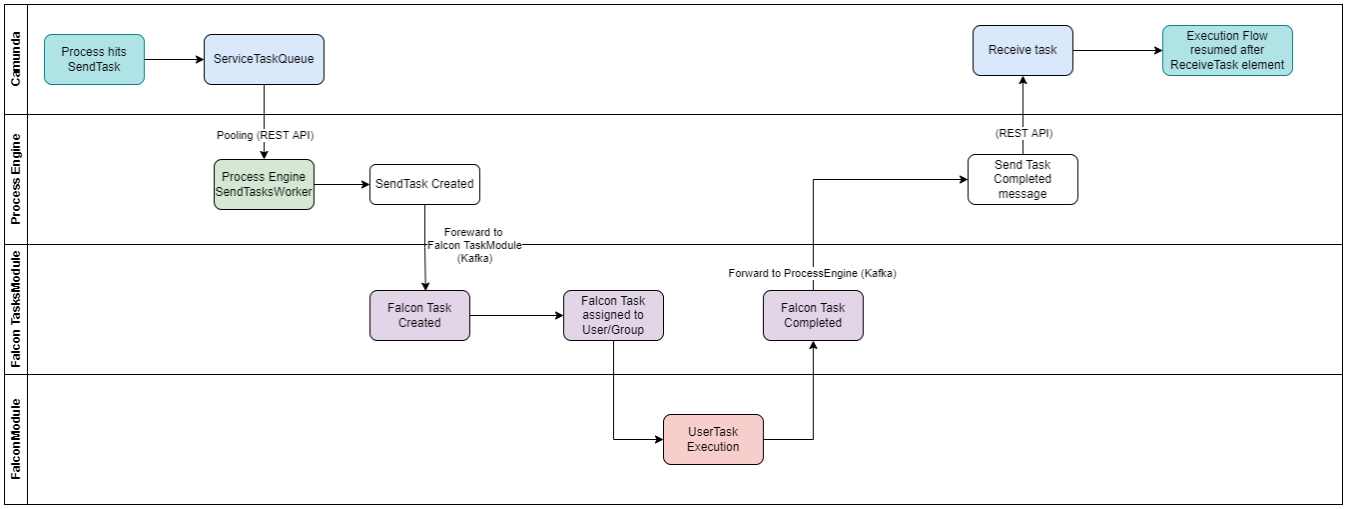


Process with Send Task/Receive Task (asynchronous communication)

This scenario involves a process that uses **Send Task** and **Receive Task** to coordinate asynchronous steps between Camunda and external microservices.

1. The process is started in Camunda.
2. The **ProcessEngine** detects a **Send Task** event and publishes a message to the Kafka.
3. The microservice gets message from Kafka and processes the task asynchronously
4. When task is completed, microservice sends a response message to Kafka.
5. The **ProcessEngine** receives the response and publishes a corresponding message to Camunda, which triggers the continuation of the process at the **Receive Task** event.
6. The process resumes execution from the point where it was waiting for the message.

This approach is suitable for long-running or asynchronous operations where the process should not block while waiting for an external system to complete its work. It also allows for better error handling and retry mechanisms in case the external microservice is temporarily unavailable.



Process with User Task (asynchronous communication)

This scenario involves a process that uses a **User Task** to delegate the task to a microservice.

1. The ProcessEngine through its internal worker periodically queries the Camunda task module to retrieve unassigned user tasks (pooling)
2. After the task detection - the worker assigns this task to a technical user. The task will not be returned again in case of pooling
3. worker creates an obieket representing the Camunda task on his side and saves it. He puts back the context data.
4. Worker publishes the corresponding message to Kafka regarding the Camunda task and the required context data. The consumer of the message is the Falcon Task Module.

5. the Falcon Task Module, based on the context data handles the task. It is responsible for assigning the task to the appropriate user/user group. Based on the context data, the task will be interpreted, delegated for execution (target module). 6 The user who executes the task will, for example, be directed to the corresponding Falcon form, which is associated with the context data of the task). The user executes the task.
6. Information on task completion goes to the Falcon task module. The module marks the task as completed. It publishes a message to Kafka about the completion of the task (the recipient of the message is ProcessEngine).
7. ProcessEngine consumes the message from Kafka about the completion of the task. ProcessEngine, using the REST API, sends a message to Camunda about the completion of the user's task. Optionally, it can also send a payload.
8. Camunda resumes continuing the process flow.