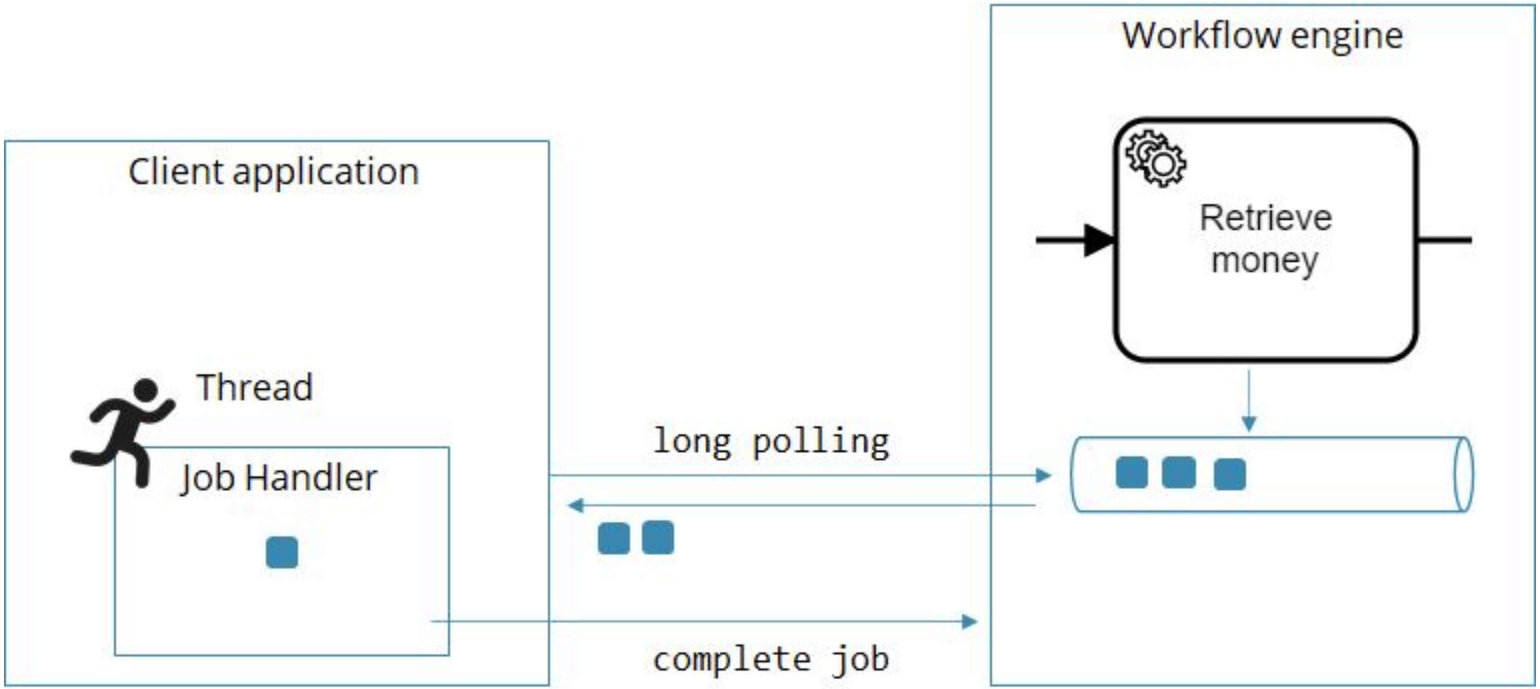
Job Workers

Long Polling vs Job Streaming

Long Polling:

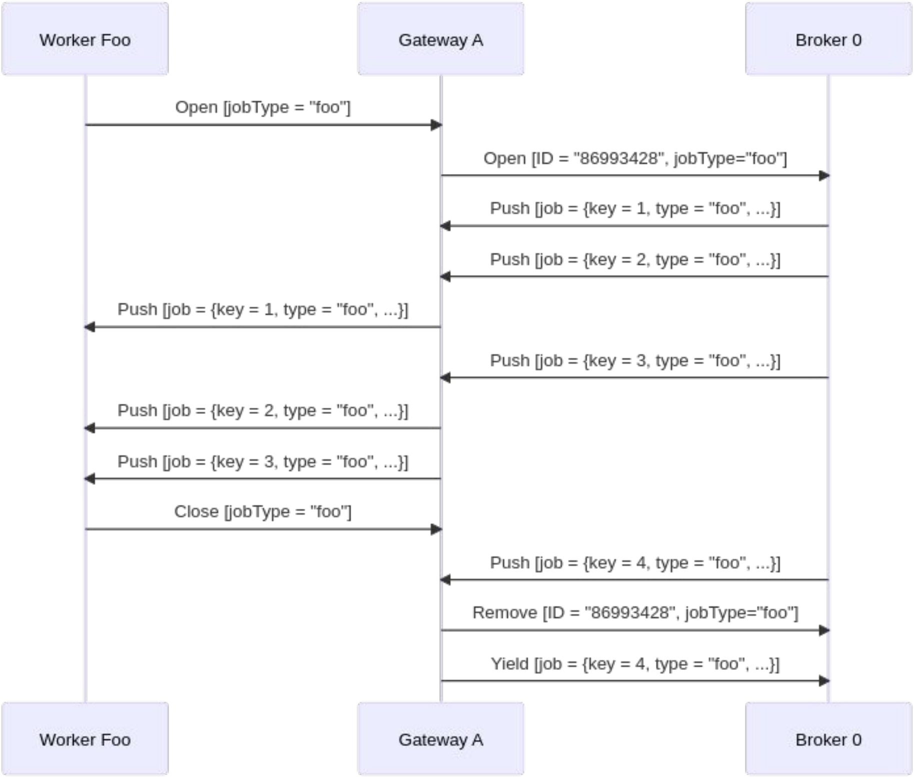
- When execution **reaches a service task** with a given job type, the broker **creates a job**.
- The job is then **queued** in the broker’s state until a worker of the same type activates it.
- If no worker is available, the job remains queued (it won’t be lost).
- Workers use **long polling** to fetch jobs efficiently:
- A worker sends an **ActivateJobs** request: “Give me up to N jobs of type X, wait up to Y ms.”
- If matching jobs are already queued → the broker responds immediately.
- If not → the request stays open until **a job arrives** or **the wait time expires**.
- **After activation**, the worker processes the job and **reports back** (completed / failed / retried), which updates or re-queues it.


Workflow engine
Retrieve money
Client application
Thread
Job Handler
long polling
complete job

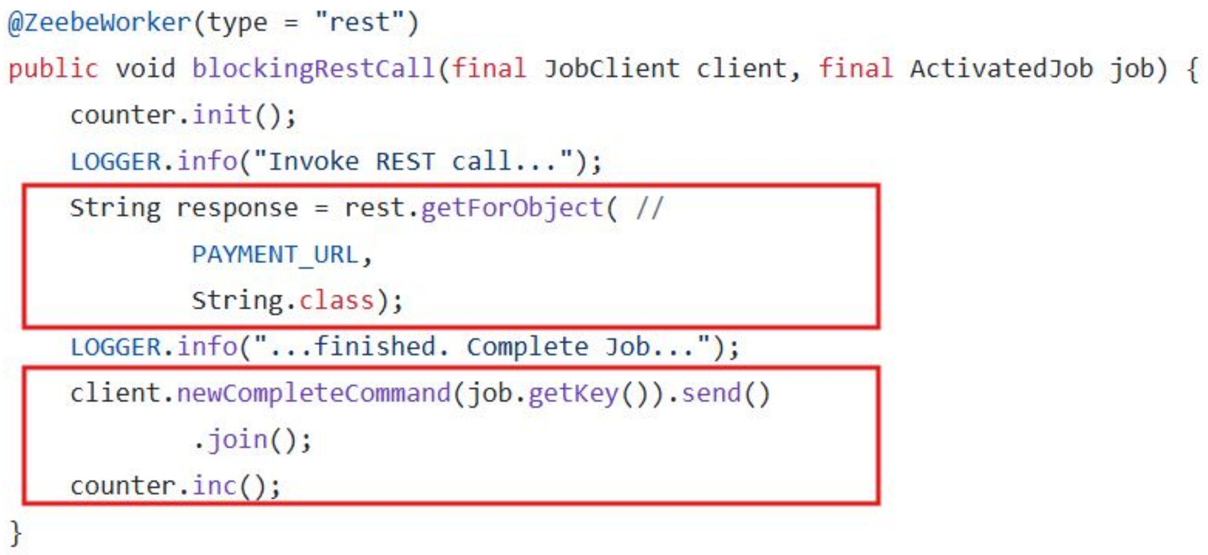
Long Polling vs Job Streaming

Job Streaming:

- When execution **reaches a service task**, the broker **creates a job**.
- Workers **subscribe to a job type** and maintain **an open streaming connection**.
- The broker pushes jobs to **the gateway,** which **buffers** them.
- The gateway enforces **backpressure**: it only sends as many jobs as the worker can consume over the connection.
- Workers complete, fail, or retry jobs, freeing capacity in the gateway buffer, allowing more jobs to flow.


Worker Foo
Gateway A
Broker 0
Open [jobType = "foo"]
Open [ID = "86993428", jobType="foo"]
Push [job = {key = 1, type = "foo", ...}]
Push [job = {key = 2, type = "foo", ...}]
Push [job = {key = 1, type = "foo", ...}]
Push [job = {key = 3, type = "foo", ...}]
Push [job = {key = 2, type = "foo", ...}]
Push [job = {key = 3, type = "foo", ...}]
Close [jobType = "foo"]
Push [job = {key = 4, type = "foo", ...}]
Remove [ID = "86993428", jobType="foo"]
Yield [job = {key = 4, type = "foo", ...}]
Worker Foo
Gateway A
Broker 0

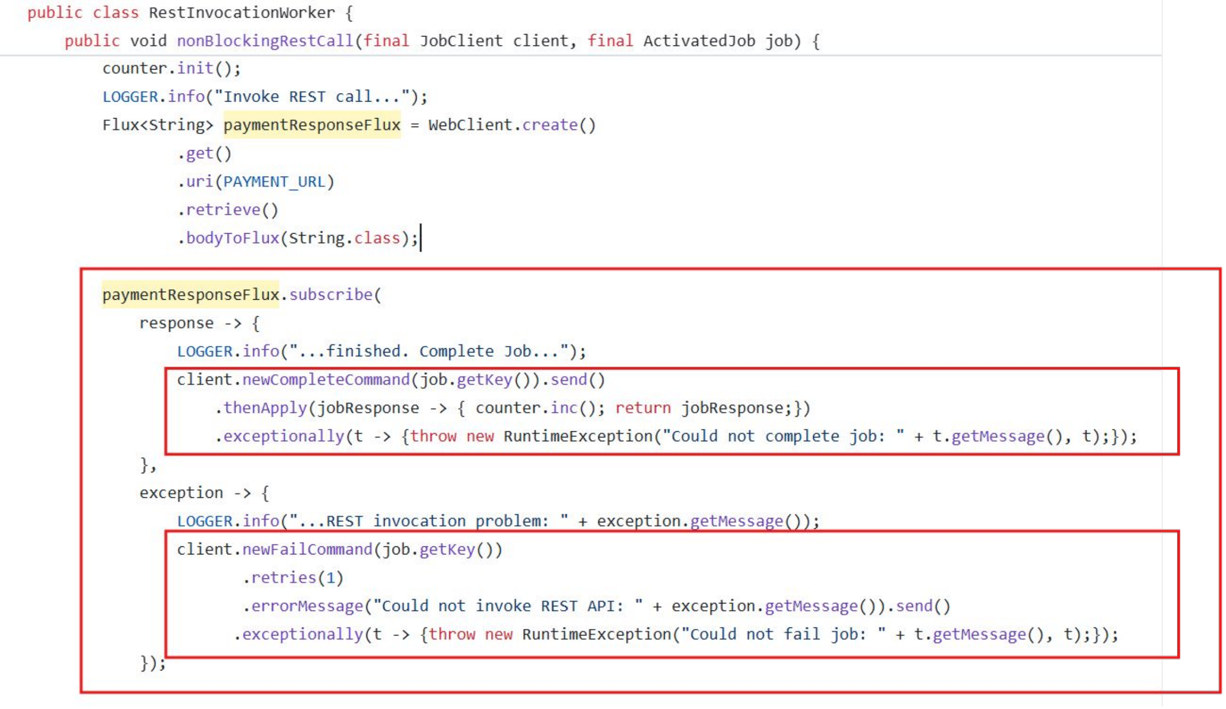
Blocking vs Non-blocking / Reactive code



```
@ZeebeWorker(type = "rest")
public void blockingRestCall(final JobClient client, final ActivatedJob job) {
    counter.init();
    LOGGER.info("Invoke REST call...");
    String response = rest.getForObject( //
            PAYMENT_URL,
            String.class);
    LOGGER.info("...finished. Complete Job...");
    client.newCompleteCommand(job.getKey()).send()
            .join();
    counter.inc();
}
```

https://github.com/berndruecker/camunda-cloud-clients-parallel-job-execution/blob/main/java-worker/src/main/java/io/berndruecker/experiments/cloudclient/java/RestInvocationWorker.java

Non-blocking / Reactive code



```
public class RestInvocationWorker {
    public void nonBlockingRestCall(final JobClient client, final ActivatedJob job) {
        counter.init();
        LOGGER.info("Invoke REST call...");
        Flux<String> paymentResponseFlux = WebClient.create()
                .get()
                .uri(PAYMENT_URL)
                .retrieve()
                .bodyToFlux(String.class);

        paymentResponseFlux.subscribe(
            response -> {
                LOGGER.info("...finished. Complete Job...");
                client.newCompleteCommand(job.getKey()).send()
                    .thenApply(jobResponse -> { counter.inc(); return jobResponse;})
                    .exceptionally(t -> {throw new RuntimeException("Could not complete job: " + t.getMessage(), t);});
            },
            exception -> {
                LOGGER.info("...REST invocation problem: " + exception.getMessage());
                client.newFailCommand(job.getKey())
                       .retries(1)
                       .errorMessage("Could not invoke REST API: " + exception.getMessage()).send()
                      .exceptionally(t -> {throw new RuntimeException("Could not fail job: " + t.getMessage(), t);});
            });
```

https://github.com/berndruecker/camunda-cloud-clients-parallel-job-execution/blob/main/java-worker/src/main/java/io/berndruecker/experiments/cloudclient/java/RestInvocationWorker.java

Connectors

A **Connector** is a pre-built integration with an external system (e.g., REST, SOAP, Kafka, AWS S3, Email).

Camunda ships connectors as part of the **Connectors runtime** (Connectors environment).

Connectors Runtime Environment

Connectors runtime environment is built on top of the same job worker mechanism.

- Connector runtime = special job worker
  - The Connectors runtime is essentially a long-running job worker that subscribes to those connector job types.
  - It polls Zeebe, activates the job, executes the integration logic (e.g. calls the REST API), then reports back.
- Completion or failure
  - If successful, the runtime completes the job with mapped outputs.
  - If there’s an error, it can fail the job (with retries, backoff, incident if exhausted) — exactly like any worker.