Event Setup Architecture

NOTE: this document was copied from the notion exports

Event Setup is the system maintained by the Event Creation pod that handles the orchestration of the various services used to communicate with the various external components involved in creating templates, updating event sections with template configurations, and duplicating event sections. This includes the EventTemplatesComponent and the EventSetup pipeline.*

Original ADR

Upstream Work

TL;DR

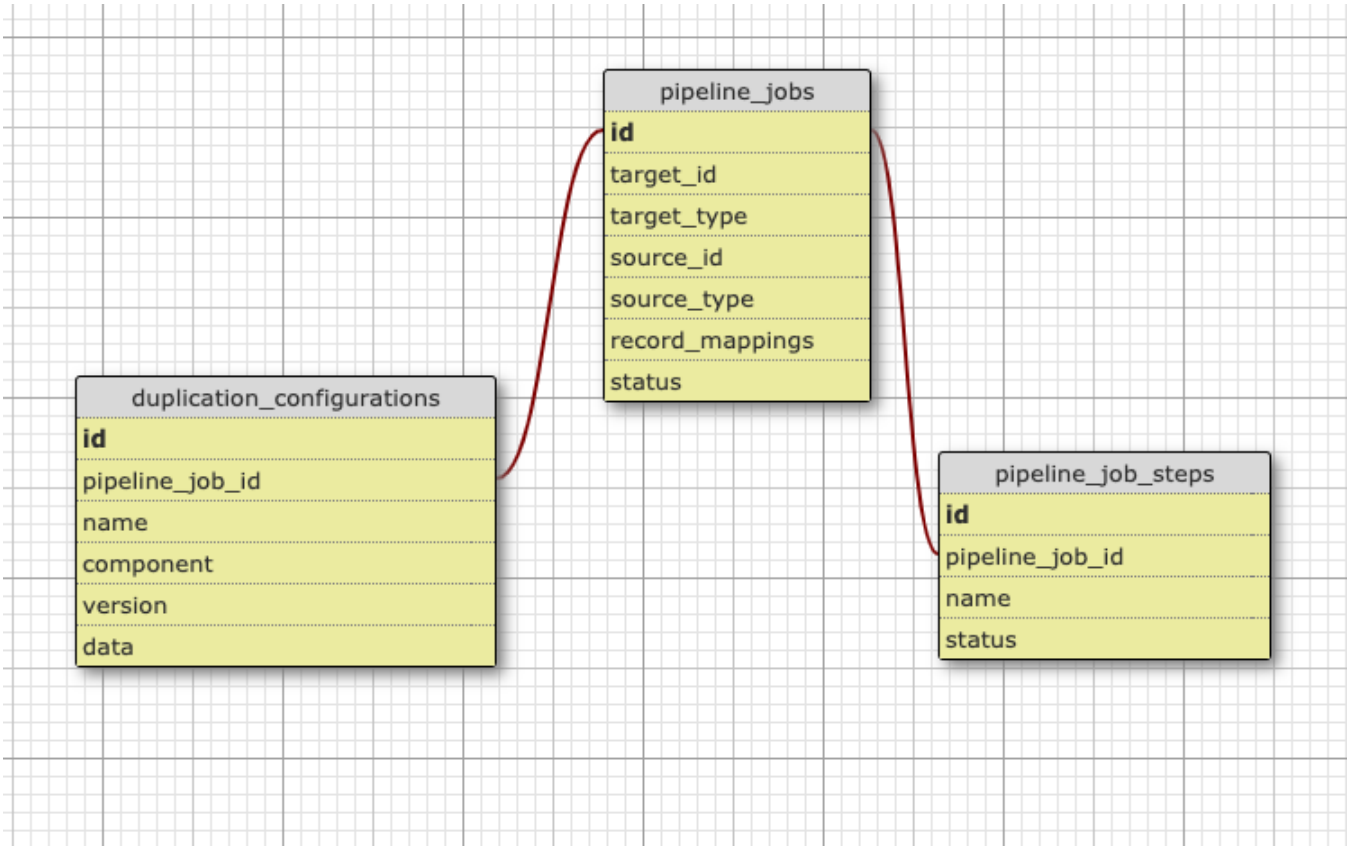
Upon receiving a request to ***use a Template***, the EventTemplatesComponent API will:

- call `CreateEventCommand` to create a *blank* event with status `building`, in the DB
- call `CreatePipelineJobCommand` to create a new `EventTemplatesComponent::PipelineJob` record
- call `StartPipelineRunJob` to enqueue a background job with the id of the above pipeline job

The above process will be similar (with some adaptions on the commands called) for the 3 possible workflows: **Creating an Event from a Template**, **Creating an Event from another Event (*Duplication*),** or **Creating a Template (*from an event*).** In all of these workflows the *event setup pipeline* will have to be prepared before it can run asynchronously.

In all 3 workflows, a *blank* **target** record will need to be created in the DB (a new **Event** or a new **Template**). The actual *pipeline workflow* will run asynchronously and the Event Templates component will keep track of the steps and progress of this workflow involving multiple external component calls.

This requires database persistence. These are the EventTemplates internal tables required for the event setup pipeline :



PipelineJob

Column	Description
target_id	the external_id of the new blank object that was created (Template or Event)
target_type	the record type of the new blank object (Template or Event)
source_id	the external_id of the record that is the basis for setting up the new one (Event or Template)
source_type	the record type of record that is the basis for setting up the new one (Template or Event)
record_mappings	a Hash like structure with the updated record mapping as returned by the various external components
status	values: cancelled, failed, waiting, completed

PipelineJobStep

Column	Description
pipeline_job_id	the pipeline job id
name	the name of the step encoding the method (build or generate), configuration name (eg. basic_settings) and the component (eg. event_dashbaord) - build.basic_settings.event_dashboard
status	values: cancelled, failed, waiting, completed

PipelineJobStep

Column	Description
pipeline_job_id	the pipeline job id
name	the name of the configuration stored
component	the name of the component which generated and builds this configuration
version	the configuration version as generated by the component
data	the internal data extracted by the component

NOTE: DuplicationConfiguration is an ancillary table for **Event Duplication**, as a duplication is a pipeline that hits each component twice - *generate* configuration (and store it in this table) and *build* the configuration (loaded from this table). These records can be marked for destruction once the PipelineJob has completed successfully. Later, once we have confidence in the system, this could be a candidate for a faster storage system such as REDIS if we identify a requirement for faster pipeline runs.

Event Setup Workflows

Name	Type	Target	Source
Template Event	Event Creation	Event	Template
Event Event	Duplication	Event	Event
Event Template	Template Creation	Template	Event

Note: For now we are focusing on the **Template Event** workflow, but the system is designed to be able to handle all 3 workflows.

Hopin::EventSetup library

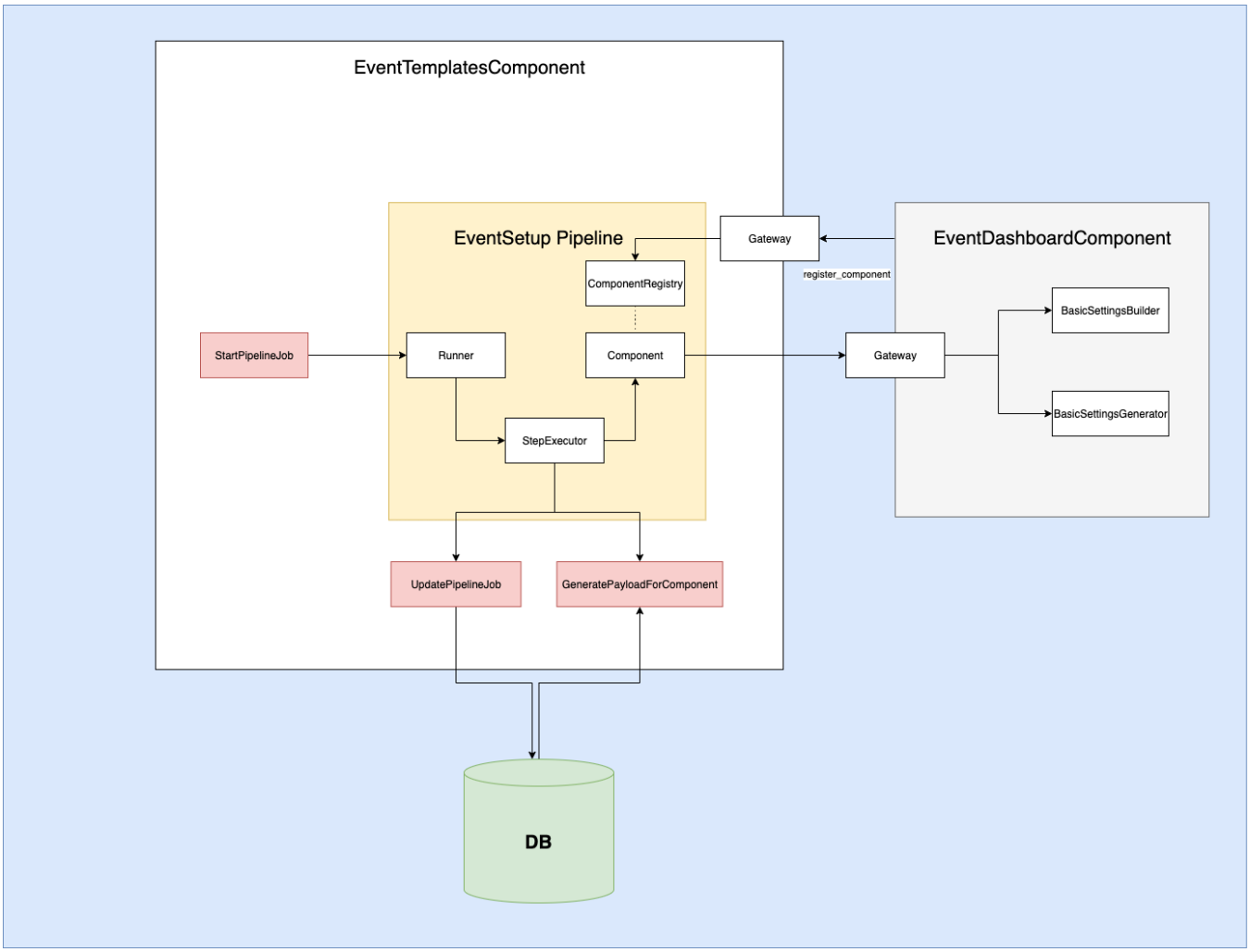
We have extracted this library as a gem in order for the external components to have access to a common set of classes and objects and override some defaults.

https://git.ringcentral.com/events/Product/event-management/hopin-event-setup

The Event Setup System

The following diagram shows a simplified version of the system with only one component. In order to explain how to system is meant to work, let's go imagine that we want to **Create a new Event** using a **Template that only stores basic_settings configuration**.

TODO: *Update diagram*



At the start of the pipeline job, there is a Template with an `external_id` , a target event created in the monolith with an `external_id` , and an **EventSet upPipelineJob** that represents the current state and looks like this:

target_id: `event_external_id`

target_type: `Event`

source_id: `template_external_id`

source_type: `Template`

record_mappings: `{}`

The **EventSetupPipelineJob has** a single **EventSetupPipelineJobStep** with the following name `build.basic_settings.event_dashboad` - This name encodes the following values *`method_name.configuration_name.component_name`*

1. The StartPipelineJobCommand

The background job that was enqueued upstream (in the EventTemplatesComponent API) is picked up for processing. It has a reference to the **EventSetupPipelineJob** `id`

This job will pull out the **PipelineJob** from the DB and call **StartPipelineJobCommand** that takes as an argument the `pipeline_job` object. This command will instantiate a new **Runner** and register the step for basic settings:

```
runner = EventSetup::Pipeline::Runner.new(build_job)
pipeline_job.steps.waiting.each do |step|
  runner.register_step(
    name: step.short_name,
    component: step.component,
    method: step.method,
  )
end
runner.run

private

def build_job
        EventSetup::Pipeline::Job.new(
          id: pipeline_job.id,
    source_id: pipeline_job.source_external_id,
    target_id: pipeline_job.target_external_id,
  )
end
```

NOTE: In this pipeline job there is a single step. As we introduce more steps and different workflows, this class will grow in complexity (we may need to can extract a **WorkflowBuilder**).

2. The Runner

The runner keeps a list of steps for the pipeline workflow. It fetches the **Component** instance from the **ComponentRegistry** and keeps track of which **method** to use on the component: `build` to setup event sections (consumes a **configuration** object) or `generate` to extract configuration settings (takes a **configuration name** and returns a **configuration** object).

```
def register_step(name:, component:, method:)
        component = @registry.find(component)
        return if component.nil?

        step = Step.new(name: name, component: component, method: method)
        steps << step if step.valid?
end
```

When running, it iterates over the steps and calls the `StepExecutor` on each step:

```
def run
  steps.each do |step|
    clock = Clock.new
    clock.measure do
          StepExecutor.new(job, step).execute
             end
  end
end
```

NOTE: The clock is used to measure the time it takes to process each step. This will allow us to identify where the bottlenecks in the system are located.

3. The Job

The Job encapsulates the **PipelineJob** record. It has the same `id`, `source_id`, and `target_id` as the DB record it is representing. The job is responsible for knowing how to **update** the pipeline_job and to **retrie**ve configurations and record mappings (wether from the DB or from in-memory or REDIS repositories if we want to avoid calling the DB during pipeline runs).

```
def update(step)
      UpdatePipelineJobStepCommand.run(
        pipeline_job_id: id,
            step: step,
      )
end

def record_mappings
  LoadRecordMappingsCommand.run!(pipeline_job_id: id)
end

def find_configuration(name)
  LoadConfigurationCommand.run!(
    pipeline_job_id: id,
    configuration_name: name,
  )
end
```

4. The Step

The step is a simple encapsulation of a workflow step. It is **valid** if the respective component knows how to handle the given method (i.e., the component *g ateway* implements said method). It contains the **method, component, and** (configuration) **name** and is able to store the component **response** (should be a `Hopin::EventSetup::Response`.

```
def valid?
      component.handles?(method)
end
```

5. The Component

The component is an abstraction of the external monolith components (packwerk Rails engines). It has a name and a gateway, which is the external public gateway that the external component implements to **build** and **generate** (it ***must*** implement these two methods!) because this internal component will need to call them:

```
# This always returns a Hopin::EventSetup::Response
def call_gateway(method:, payload:)
  if handles?(method.to_sym)
    response = send(method, payload)
    return response if response.is_a? Hopin::EventSetup::Response

    bad_response("#{name} returned invalid response type")
  else
    bad_response("Unknown gateway method `#{method}`")
  end
end

def handles?(method_name)
  gateway.respond_to?(method_name)
end

private

def generate(payload)
  @gateway.generate(payload[:configuration_name], payload[:event_external_id])
end

def build(payload)
  @gateway.build(payload[:configuration], payload[:event_external_id], payload[:record_mappings])
end

def bad_response(message)
  errors = { component: message }
  Hopin::EventSetup::Response.new(success: false, errors: errors)
end
```

Components are registered via the **EventTemplatesComponent** public gateway using the monolith's `event_templates_component_event_setup` initialiser:

```
Rails.application.reloader.to_prepare do
      # register_component receives the name for the component (without the suffix)
      # and the class of the gateway that will receive event setup requests
  EventTemplatesComponent::Gateway::EventSetup.register_component(
    :event_dashboard,
    EventDashboardComponent::Public::EventSetup,
  )
end
```

Each component gateway must implement `build` and `generate` methods.

Here's an example of an external public gateway interface with build:

```
# typed: strict
# frozen_string_literal: true

module EventDashboardComponent
  module Public
    class EventSetup
      class << self
        extend T::Sig

        sig do # rubocop:disable HopinCops/EnforceValueObject
          params(
            configuration: Hopin::EventSetup::Configuration,
            event_external_id: String,
            record_mappings: T::Hash[T.untyped, T.untyped],
          )
          .returns(Hopin::EventSetup::Response)
        end
        def build(configuration, event_external_id, record_mappings = {})
          EventDashboardComponent::EventSetup::ExecuteBuildService.run(
            configuration: configuration,
            event_external_id: event_external_id,
            record_mappings: record_mappings,
          )
        end
      end
    end
  end
end
```

6. The StepExecutor

The Runner will call the `StepExecutor` for each step. The step has information about the **internal component** (abstraction of the external component) and will call the correct method for the step. The `Step` will then store the response and the `Job` will receive the `Step` for updating the state of the pipeline run.

```
def execute
  step.response = step.component.call_gateway(
              method: step_method,
              payload: payload
        )
  job.update(step)
end

private

def payload
  { event_external_id: event_external_id }.tap do |hash|
    if build_step?
      hash[:configuration] = job.find_configuration(step.name)
      hash[:recordmappings] = job.record_mappings
    end

    hash[:configuration_name] = step.name if generate_step?
  end
end
```

NOTE: The payloads for the `build` and `generate` methods are different, but they share the need for an `event_external_id`. This will also be different depending on workflow being run and the `StepExecutor` returns the appropriate value:

```
def event_external_id
  if build_step?
    job.target_id
  elsif generate_step?
    job.source_id
  end
end
```

7. End Pipeline Run

At the end of each pipeline run, the `UpdatePipelineJobCommand` is called to update the status of the **PipelineJob** record and the status of the new **Event**.

NOTE: For now, we want to allow the user to have access to the event regardless of wether the **PipelineJob** was fully successful or not. The idea is to have a fault tolerant system that can handle errors that take place in the components.

FUTURE

Pipeline as a service

Because there is a need for this system to be fault tolerant (i.e., doesn't crash while it's running), there is a high risk that it may crash inside one of the external components. This means that we would not be able to keep a valid state for the pipeline without some sort of clean up worker/job. It's also difficult to implement timeouts under this system or deal with parallelization of *non-codependent components*.

Therefore, we may need to extract the pipeline to a service external to the monolith, so that we could communicate with the various components via API calls/webhooks and therefore implement a timeout system and be free from errors happening inside each component. The pipeline code already removes much of the direct knowledge of DB models and external components, and can perhaps be easily extracted into a Sinatra application that would be run as a micro-service.

REDIS as the pipeline storage

We have noticed that `Sidekiq` will sometimes create new instances of the `ComponentRegistry` without re-initializing the Rails app, which means references to the components is lost. Currently we re-register all components on each run, but that is not ideal.

We believe that we could use an adapter for the `ComponentRegistry` to use REDIS as a store, therefore avoiding any data loss. REDIS could also be used to store any `record_mappings` and the `configuration` objects for a given run, making a pipeline run independent from a connection to the monolith DB.