Flyte Documentation

Release 0.0.1

Flyte Authors

Jun 25, 2020

Flyte Documentation

Flyte is a structured programming and distributed processing platform created at Lyft that enables highly concurrent, scalable and maintainable workflows for machine learning and data processing. Welcome to the documentation hub for Flyte.

CHAPTER 1

Introduction

Flyte is a structured programming and distributed processing platform created at Lyft that enables highly concurrent, scalable and maintainable workflows for machine learning and data processing. Flyte consists of tasks as fundamental building blocks (like functions in functional programming) that can be linked together in a directed acyclic graph to form workflows (control flow specification). Flyte uses protocol buffers as the specification language to specify these workflows and tasks. The actual implementation of the task can be in any language as the tasks themselves are executed as containers. Flyte comes with Flytekit, a python SDK to develop applications on Flyte (authoring workflows, tasks etc). Want to learn more, Read ON!

1.1 What is Flyte?

Workflows are ideal for organizing complex processes. As Lyft has evolved, developers have turned to workflows to solve an increasing number of business problems. Flyte was created as an end-to-end solution for developing, executing, and monitoring workflows reliably at scale.

The system powers products all over Lyft, including Data Science, Pricing, Fraud, Driver Engagement, Locations, ETA, and Autonomous.

Flyte handles all the overhead involved in executing complex workflows, including hardware provisioning, scheduling, data storage, and monitoring. This allows developers to focus solely on writing their workflow logic.

1.1.1 Development

In Flyte, workflows are expressed as a graph. To create a workflow, you must compose a graph that conforms to our strict specification. We provide developer tools that make it easy to define robust workflows that conform to this spec. The spec allows us to perform up-front validation and ensure data is passed properly from one task to the next. This practice has proven extremely useful at Lyft in time and cost savings.

In as few as 20 lines of code, a user can define a custom workflow, submit the graph to Flyte, and snooze while Flyte fulfills the execution on a schedule, or on-demand.

1.1.2 Execution

When fulfilling a workflow execution, Flyte will dynamically provision the resources necessary to complete the work. Tasks are launched using docker containers to ensure consistency across environments. The system can distribute workloads across 100s of machines, making it useful for production model training and batch compute at Lyft.

As tasks complete, unused resources will be deprovisioned to control costs.

Workflows come with a variety of knobs such as automatic retries, and custom notifications on success or failure.

1.1.3 Monitoring

The Flyte system continuously monitors workflow executions throughout their lifecycle. You can visit the Flyte web interface at any time to track the status of a workflow.

The web interface provides the ability to drill down into the individual workflow steps, and get rich error messaging to triage any issues.

1.2 How to read these docs?

Flyte can be interacted with and operated in 3 ways, or by 3 personas: **user**, **administrator**, and **contributor**. The documentation is organized in a way so that the reading flow is optimized for each of these personas.

1.2.1 User: I want to write Flyte Workflows

A user refers to anyone who wants to:

- Explore how Flyte works and try it out before installing, operating, or using it
- Use a hosted Flyte deployment available at her organization
- Deploy a local cluster for debugging or testing

Jump to User Docs

1.2.2 Administrator: I want to manage a Flyte installation at my company

An administrator is someone who wants to deploy, manage, and scale a Flyte installation for his or her organization. The administrator is not interested in altering or modifying any code, only using the system off the shelf, and configuring the various available knobs and settings. This section also talks about the typical installation structure and scalability primitives available in the system.

Jump to Administrator Docs

1.2.3 Contributor: I want to help make Flyte even more awesome!

A contributor is someone who wants to:

- Extend Flyte's capabilities by contributing new plugins
- Fix a bug in the source code
- Gain a deep understanding of Flyte's architecture
- Add new functionality or alter the source code for their organization

- Contribute to this documentation

We view Flyte enthusiasts as potential contributors and recommend they read through these docs as well.

Jump to Contributor Docs

CHAPTER 2

User docs

This doc guides you using Flyte locally or a version of Flyte hosted by your organization.

2.1 Getting Started

Warning: In order to try Flyte, you must first install it. If you do not already have access to a Flyte installation, you can install the Flyte sandbox to try Flyte. Refer to the *Installation Guide* to set up the Flyte sandbox.

2.1.1 Quick Start Examples

Before you can run any workflows, you'll first need to register a workflow in Flyte. We've written an example Flyte workflow, and placed it into a docker image called `flytesnacks`.

The example workflow takes in an image url as input, does edge detection, and produces an image showing the edges in the photo.

Workflow Setup in Flyte

Registration

If you're using the `sandbox` flyte installation, you can use the following command to register our example workflow with Flyte

```
docker run --network host -e FLYTE_PLATFORM_URL='127.0.0.1:30081' lyft/flytesnacks:v0.
↪1.0 pyflyte -p flytesnacks -d development -c sandbox.config register workflows
```

This command will register the workflow with your Flyte app under the `development` domain of the project `flytesnacks`.

NOTE: if your Flyte endpoint is something other than `localhost:30081`, change the `FLYTE_PLATFORM_URL` value accordingly.

Running Workflows in Flyte

Creating an execution

Now that your workflow is registered, you can visit the Flyte console to run the workflow.

From the flyte console homepage http://localhost:30081/console, click the "development" link under `flytesnacks`. This will show you the list of workflows registered under the `development` domain of the `flytesnacks` project.

Click on the `workflows.edges.EdgeDetectorWf` workflow. This will take you to the Workflow Details page.

Click the "Launch Workflow" button. This will open the Launch Workflow Form.

Leave the `Workflow Version` and `Launch Plan` default values, but insert any publicly accessible image url in the `image_input` section. For example, you can use https://images.ctfassets.net/q8mvene1wzq4/1OQ8OBLzXGv8pVjFTLf0QF/9a7b8cdb982161daebd5618fc7cb5041/Car_blue_L.png.

Click "Launch" at the bottom of the form (you may need to scroll down).

In a few moments, you'll see the execution getting fulfilled by the Flyte system. Observe as Flyte runs through the workflow steps.

When the workflow is complete, click the "run-edge-detection" link under "Node", this will show you some details about the execution. Click the "Outputs" tab. You should see that the workflow produced an output telling you where it stored the produced image.

To find this image, visit the minio UI at http://localhost:30081/minio (the sandbox username is `minio` and the sandbox password is `miniostorage`).

Follow the path given in your workflow output. If you download the file, you'll see that the workflow produced the edge-detected version of the image url input.

2.1.2 Writing Your First Workflow

The easiest way to author a Flyte Workflow is using the provided python SDK called "FlyteKit".

You can save some effort by cloning the `flytesnacks` repo, and re-initializing it as a new git repository

```
git clone git@github.com:lyft/flytesnacks.git myflyteproject
cd myworkflow
rm -rf .git
git init
```

now open the "Makefile" and change the first line to `IMAGE_NAME=myflyteproject`

Let's also remove the existing python task so we can write one from scratch.

```
rm workflows/edges.py
```

Creating a Project

In Flyte, workflows are organized into namespaces called "Projects". When you register a workflow, it must be registered under a project.

Lets create a new project called `myflyteproject`. Use the project creation endpoint to create the new project

```
curl -X POST localhost:30081/api/v1/projects -d '{"project": {"id": "myflyteproject",
↪"name": "myflyteproject"} }'
```

Writing a Task

The most basic Flyte primitive is a "task". Flyte Tasks are units of work that can be composed in a workflow. The simplest way to write a Flyte task is using the FlyteSDK.

Start by creating a new file

```
touch workflows/first.py
```

This directory has been marked in the configuration file as the location to look for workflows and tasks. Begin by importing some of the libraries that we'll need for this example.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import urllib.request as _request

import cv2
from flytekit.common import utils
from flytekit.sdk.tasks import python_task, outputs, inputs
from flytekit.sdk.types import Types
from flytekit.sdk.workflow import workflow_class, Output, Input
```

From there, we can begin to write our first task. It should look something like this.

```
@inputs(image_location=Types.String)
@outputs(parsed_image=Types.Blob)
@python_task
def edge_detection_canny(wf_params, image_location, parsed_image):
    with utils.AutoDeletingTempDir('test') as tmpdir:
        plane_fname = '{}/plane.jpg'.format(tmpdir.name)
        with _request.urlopen(image_location) as d, open(plane_fname, 'wb') as opfile:
            data = d.read()
            opfile.write(data)

        img = cv2.imread(plane_fname, 0)
        edges = cv2.Canny(img, 50, 200)  # hysteresis thresholds

        output_file = '{}/output.jpg'.format(tmpdir.name)
        cv2.imwrite(output_file, edges)

        parsed_image.set(output_file)
```

Some of the new concepts demonstrated here are:

- `wf_params`: The first argument to a python task is a Flyte SDK defined object that offers handlers like logging.
- Inputs and outputs are first defined in the decorator, and then passed into the argument of the function. Note that the names in the function signature need to match those in the decorator arguments.
- A `Blob` is a Flyte Kit type that represents binary data. It is used to offload data to a storage location like S3. Here we use it to store an image.

Writing a Workflow

Next you need to call that task from a workflow. In the same file, add these lines.

```
@workflow_class
class EdgeDetectorWf(object):
    image_input = Input(Types.String, required=True, help="Image to run for")
    run_edge_detection = edge_detection_canny(image_location=image_input)
    edges = Output(run_edge_detection.outputs.parsed_image, sdk_type=Types.Blob)
```

This code block creates a workflow, with one task. The workflow itself has an input (the link to an image) that gets passed into the task, and an output, which is the processed image.

Interacting with Flyte

Flyte fulfills tasks using docker images. You'll need to build a docker image from this code before it can run in Flyte. The repo has a make target to build the docker image for you

```
make docker_build
```

If you have the flyte sandbox installed on your local machine, the image will be accessible to to your Flyte system. If you're running a remote Flyte instance, you'll need to upload this image to a remote registry such as Dockerhub, Amazon ECR, or Google Container Registry, so that it can be used by the Flyte system.

To upload to a remote registry, use

```
DOCKER_REGISRY_USERNAME={username} DOCKER_REGISTRY_PASSWORD={pass} REGISTRY=docker.io␣
↪make docker_build
```

Replace the values above with your registry username, password, and registry endpoint.

You may need to change the `IMAGE_NAME` in the Makefile to reflect your namespace in the docker registry. (ex `{{my docker username}}/myflyteproject`)

With the image built, we just need to register the tasks and workflows. The process is the same as what we had done previously.

```
docker run --network host -e FLYTE_PLATFORM_URL='127.0.0.1:30081' {{ your docker␣
↪image }} pyflyte -p myflyteproject -d development -c sandbox.config register␣
↪workflows
```

After this, you should be able to visit the Flyte UI, and run the workflow as you did with `flytesnacks` previously.

2.2 Flyte Concepts

This section provides a quick overview of some important concepts in Flyte with links to deeper dives.

Tasks are at the core of Flyte. A Task is any independent unit of processing. Tasks can be pure functions or functions with side-effects. Tasks also have configuration and requirements specification associated with each definition of the task.

Workflows are programs that are guranteed to reach a terminal state eventually. They are represented as Directed Acyclic Graphs (DAGs) expressed in protobuf. The Flyte specification language expresses DAGs with branches, parallel steps and nested Workflows. Workflow can optionally specify typed inputs and produce typed outputs, which are captured by the framework. Workflows are composed of one or more *Nodes*. A Node is an encapsulation of an instance of a Task.

Executions are instances of workflows, nodes or tasks created in the system as a result of a user-requested execution or a scheduled execution.

Projects are a multi-tenancy primitive in Flyte that allow logical grouping of Flyte workflows and tasks. Projects often correspond to source code repositories. For example the project *Save Water* may include multiple *Workflows* that analyze wastage of water etc.

Domains enable workflows to be executed in different environments, with separate resource isolation and feature configuration.

Launchplans provide a mechanism to specialize input parameters for workflows associated different schedules.

Registration is the process of uploading a workflow and its task definitions to the *FlyteAdmin* service. Registration creates an inventory of available tasks, workflows and launchplans declared per project and domain. A scheduled or on-demand execution can then be launched against one of the registered entities.

2.2.1 Tasks

Definition

Tasks encapsulate fully independent units of execution. Flyte language exposes an extensible model to express that in an execution-independent language. Flyte contains first class task plugins that take care of executing these tasks.

Real world examples

Query a data store Using *Hive tasks* to retrieve data into dataframes so that subsequent tasks can process them.

Transform data Using *Container tasks* to transform data collected/computed earlier, users can develop a task as a simple Lambda function with specified inputs and outputs represented as a container with entrypoints, with specific compute, memory and gpu requirements.

Map-reduce massive jobs Using *Spark programs* with their cluster configuration and compute requirements

Hyperparameter tuning task Using a system like Katib, users can execute a task that needs multiple iterations and leads to multiple other containers to execute.

A distributed or single container Tensorflow Job

Characteristics

In abstract, a task in the system is characterized by:

1. A *project and domain* combination,
2. A unique unicode name (we recommend it not to exceed 32 characters), and
3. A version string.
4. *Optional* Task interface definition

   In order for tasks to exchange data with each other, a task can define a signature (much like a function/method signature in programming languages). A task interface defines the *input and output variables* as well as their *types*.

Requirements

When deciding whether a unit of execution conistitutes a Flyte Task or not. Consider the following:

- Is there a well-defined graceful/successful exit criteria for the task? A task is expected to exit after finishing processing its inputs.
- Is it repeatable? Under certain circumstances, a task might be retried, rerun… etc. with the same inputs. It's expected to produce the same outputs every single time. For example, avoid using random number generators with current clock as seed but opt to using a system-provided clock as the seed.
- Is it a pure function? i.e. does it have side effects that are not known to the system (e.g. calls a web-service). It's strongly advisable to avoid side-effects in tasks. When side-effects are required, ensure that those operations are **idempotent_**.

Types

Since it's impossible to define the unit of execution of a task the same way for all kinds of tasks, Flyte allows different task types in the system. Flyte comes with a set of defined, battle tested task types but also allows for a very flexible model to introducing new *user-defined task types*. Read more about various supported task types *here*.

Fault tolerance

In any distributed system failure is inevitable, allowing users to design a fault-tolerant system (e.g. workflow) is an inherent goal of Flyte. At a high level, tasks offer two parameters to control how to handle that:

Retries Tasks can define a retry strategy to let the system know how to handle failures (e.g. retry 3 times on any errors).

Timeouts In order for the system to ensure it's always making progress, tasks must be guaranteed to end. The system defines a default timeout period for tasks. It's also possible for task authors to define a timeout period after which the task is marked as failure. Note that a timed-out task will be retried if it has a retry strategy defined.

Memoization

Flyte supports memoization for task outputs to ensure identical invocations of a task are not repeatedly executed wasting compute resources. For more information on memoization please refer to *Task Cache*.

2.2.2 Workflows

A workflow is a directed acyclic graph (DAG) of units of work encapsulated by nodes. Specific instantiations of a workflow (commonly with bound input arguments) are referred to as workflow executions, or just executions. In other words, a workflow is a template for ordered task execution.

Flyte workflows are defined in protobuf and the Flyte SDK facilitates writing workflows. Users can define workflows as a collection of nodes. Nodes within a workflow can produce outputs that subsequent nodes depend on as inputs. These dependencies dictate the workflow structure. Workflows written using the SDK do not need to explicitly define nodes to enclose execution units (tasks, sub-workflows, launch plans), these will be injected by the SDK and captured at registration time.

Structure

Workflows can accept inputs and/or produce outputs and re-use task definitions across *projects* and *domains*. Workflow structure is flexible - nodes can be executed in parallel, the same task definition can be re-used within the same workflow. A single workflow can contain any combination of task types. A workflow can contain a single functional node, it can contain many nodes in all sorts of arrangements and even launch other workflows.

At execution time, node executions will be triggered as soon as their inputs are available. Workflow nodes are naturally run in parallel when possible.

Flyte-specific structure

During registration, the Flyte platform will validate workflow structure and save the workflow. The registration process also updates the workflow graph. A compiled workflow will always have a start and end node injected into the workflow graph. In addition a failure handler will catch and process execution failures.

Versioning

Like *tasks*, workflows are versioned. Registered workflows are also immutable meaning that an instance of a workflow defined by project, domain, name and version cannot be updated. The tasks referenced in a workflow version are themselves immutable and are tied to specific tasks' versions.

Executions

A workflow can only be executed through a launch plan. A workflow can be launched many times with a variety of launch plans and inputs. Workflows that produce inputs and outputs can take advantage of *task caching* to cache intermediate inputs and outputs and speed-up subsequent executions.

2.2.3 Nodes

A node represents a unit of execution or work within a workflow. Ordinarily a node will encapsulate an instance of a *task* but a node can also contain an entire subworkflow or trigger a child workflow. Nodes can have inputs and outputs, which are used to coordinate task inputs and outputs. Node outputs can be used as inputs to other nodes within a workflow.

Tasks are always encapsulated within a node, however, like tasks, nodes can come in a variety of flavors determined by their *target*. These targets include *task nodes*, *workflow nodes*, and *branch nodes*.

Task Nodes

Tasks referenced in a workflow are always enclosed in nodes. This extends to all task types, for example an array task will be enclosed by a single node. For more information about different task types, refer to *task types*.

Workflow Nodes

A node can contain an entire sub-workflow. Because workflow executions always require a launch plan, workflow nodes have a reference to a launch plan used to trigger their enclosed workflows.

Branch Nodes

Branch nodes alter the flow of the workflow graph. Conditions at runtime are evaluated to determine control flow.

2.2.4 Overview of the Execution of a Workflow

Typical flow using flyte-cli

- When you request an execution of a Workflow using the UI, Flyte CLI or other stateless systems, the system first calls the getLaunchPlan endpoint and retrieves a Launch Plan matching the name for a version. The Launch Plan definition includes the definitions of all the input variables declared for the Workflow.
- The user-side component then ensures that all required inputs are supplied and requests the FlyteAdmin service for an execution
- The Flyte Admin service validates the inputs, making sure that they are all specified and, if required, within the declared bounds.
- Flyte Admin then fetches the previously validated and compiled workflow closure and translates it to an executable format with all of the inputs.
- This executable Workflow is then launched on Kubernetes with an execution record in the database.

2.2.5 Projects

A project in Flyte is a grouping of workflows and tasks to achieve a particular goal. A Flyte project can map to an engineering project or everything that's owned by a team or an individual. There cannot be multiple projects with the same name in Flyte. Since the fully-qualified name for tasks and workflows include project name and domain name, the task/workflow names are only required to be unique within a project. The workflows in a project A can refer tasks and workflows from other projects using the fully-qualified name.

Flyte allows users to set resource limit for each project. Flyte provides basic reports and dashboards automatically for each project. The information captured in these reports include workflow/task level insights, resource usage and billing information.

2.2.6 Domains

Domains provide an abstraction to isolate resources and feature configuration for different deployment environments. For example at Lyft we develop and deploy Flyte workflows in development, staging, and production. We configure Flyte domains with those names, and specify lower resource limits on the development and staging domains than production domains. We also use Domains to disable Launchplans and schedules from development and staging domains, since those features are typically meant for production deployments.

2.2.7 Understanding Registration process

Typical Flow Using Flytekit

- A user defines tasks using the Flytekit Task Definition language
- A User defines a workflow using either Flytekit Workflow definition lanugage.
- The user then uses flytekit's register cli to compile the tasks into their serialized representation as described in *Flyte Specification language*. During this time the task representation is also bound to a container that contains the code for the task. This associated entity is registered with FlyteAdmin using the registerTask api.
- The user then uses flytekit's register cli to compile the workflow into their serialized representation as described in *Flyte Specification language*. The referenced tasks are replaced by their Flyte Admin registered Identifier, obtained in the previous step. This associated entity is registered with FlyteAdmin using the registerWorkflow api.
- She can then launch an execution using the FlyteAdmin launch execution API, which requires the necessary inputs provided. This is automatically done if she uses the Flyte-CLI to launch the execution.
- She can later use the FlyteAdmin read API's to get details of the execution, monitor it to completion or retrieve a historical execution
- OR alternatively she can use the FlyteConsole to visualize the execution in realtime as it progresses or visualize any historical execution. The console also makes it easy to view debugging information for the execution.
- She can set specific rules like *notification* on failure or success or publish all events in an execution to a pub-sub system.
- She can also query the datastore to get a summary of all the executions and the compute resources consumed.

Typical Flow without Flytekit

It is possible to achieve the exact same workflow as above in case Flytekit is not available. Workflows and tasks are purely specifications and can be provided using any tool like YAML, JSON, protobuf binary or any other programming language. Contributions welcome.

Registration in the Backend

When FlyteAdmin receives a workflow registration request, it uses the workflow compiler to compile and validate the workflow. It also fetches all the referenced tasks and creates a complete workflow closure, which is stored in the metastore. If the workflow compilation fails, the compiler returns an error to the client.

2.3 Flyte Features

2.3.1 FlyteCLI

A command-line interface for interacting with Flyte

The FlyteCLI is a command-line tool that allows users to perform administrative tasks on their Flyte workflows and executions. It is an independent module but installed as part of the *Flyte Kit <components-flytekit>*. It primarily iteracts with the *FlyteAdmin <components-flyteadmin>* service over its gRPC interface, allowing users to list registered workflows, or get a currently running execution.

Installation

The easist way to install FlyteCLI is using virtual environments. Follow the official doc to install the `virtualenv` package if you don't already have it in your development environment.

Install from source

Now that you have virtualenv, you can either install flyte-cli from source. To do this first clone the git repository and after setting up and activating your virtual environment, change directory to the root directory of the flytecli package, and install the dependencies with `pip install -e .`.

Install from pypi [recommended]

Another option is to just install flyte-cli from prebuilt binaries

Testing if you have a working installation

To test whether you have a successful installation of flytecli, run `flyte-cli` or `flyte-cli --help`.

If you see the following output, you have installed the FlyteCLI successfully.

```
Usage: flyte-cli [OPTIONS] COMMAND [ARGS]...

    Command line tool for interacting with all entities on the Flyte Platform.

Options:
    -n, --name TEXT     [Optional] The name to pass to the sub-command (if
...

Commands:
    execute-launch-plan        Kick off a launch plan.
...
```

Terminology

This section introduces and explains the most commonly used terms and concepts the users will see in FlyteCLI.

Host

`Host` refers to your running Flyte instance and is a common argument for the commands in FlyteCLI. The FlyteCLI will only be interacting with the Flyte instance at the URL you specify with the `host` argument. parameter. This is a required argument for most of the FlyteCLI commands.

Project

`Project` is a multi-tenancy primitive in Flyte and allows logical grouping of instances of Flyte entities by users. Within Lyft's context, this term usually refers to the name of the Github repository in which your workflow code resides.

For more information see *Projects*

Domain

The term `domain` refers to development environment (or the service instance) of your workflow/execution/launch plan/etc. You can specify it with the `domain` argument. Values can be either `development`, `staging`, or `production`. See *Domains*

Name

The `name` of a named entity is a randomly generated hash assigned automatically by the system at the creation time of the named entity. For some commands, this is an optional argument.

Named Entity

`Name Entity` is a primitive in Flyte that allows logical grouping of processing entities across versions. The processing entities to which this term can refer include unversioned `launch plans`, `workflows`, `executions`, and `tasks`. In other words, an unversioned `workflow` named entity is essentially a group of multiple workflows that have the same `Project`, `Domain`, and `Name`, but different versions.

URN

Note: URN is a FlyteCLI-only concept. You won't see this term in other flyte modules.

URN stands for "unique resource name", and is the identifier of a version of a given named entity, such as a workflow, a launch plan, an execution, or a task. Each URN uniquely identifies a named entity. URNs are often used in FlyteCLI to interact with specific named entities.

The URN of a version of a name entity is composible from the entity's attributes. For example, the URN of a workflow can be composed of a prefix *wf* and the workflow's `project`, `domain`, `name`, and `version`, in the form of `wf:<project>:<domain>:<name>:<version>`.

Note that execution is the sole exception here as an execution does not have versions. The URN of an execution, therefore, is in the form of `ex:<project>:<domain>:<name>`.

Getting help

FlyteCLI uses subcommands. Whenever you feel unsure about the usage or the arguments of a command or a subcommand, get help by running `flyte-cli --help` or `flyte-cli <subcommand> --help`

Commands

For information on available commands in FlyteCLI, refer to FlyteCLI's help message.

2.3.2 Metrics for your executions

Flyte-Provided Metrics

Whenever you run a workflow, Flyte Platform automatically emits high-level metrics. These metrics follow a consistent schema and aim to provide visibility into aspects of the Platform which might otherwise be opaque. These metrics will help users diagnose whether an issue is inherent to the Platform or to one's own task or workflow implementation. We will be adding to this set of metrics as we implement the capabilities to pull more data from the system, so keep checking back for new stats!

At a highlevel, workflow execution goes through the following discrete steps:

Description of main events for workflow execution

Events	Description
Acceptance	Measures the time between when we receive service call to create an Execution (Unknown) and when it has moved to Queued.
Transition Latency	Measures the latency between two consecutive node executions, the time spent in Flyte engine.
Queuing Latency	Measures the latency between the time a node's been queued to the time the handler reported the executable moved to running state.
Task Execution	Actual time spent executing user code
Repeat steps 2-4 for every task	
Transition Latency	See #2
Completion Latency	Measures the time between when the WF moved to succeeding/failing state and when it finally moved to a terminal state.

Flyte Stats Schema

Prefix	Type	Description
`propeller.all.workflow.acceptance-latency-ms`	Timer (ms)	Measures the time between when we receive service call to create an Execution (Unknown) and when it has moved to Queued.
`propeller.all.node.queueing-latency-ms`	Timer (ms)	Measures the latency between the time a node's been queued to the time the handler reported the executable moved to running state.
`propeller.all.node.transition-latency-ms`	Timer (ms)	Measures the latency between two consecutive node executions, the time spent in Flyte engine.
`propeller.all.workflow.completion-latency-ms`	Timer (ms)	Measures the time between when the WF moved to succeeding/failing state and when it finally moved to a terminal state.
`propeller.all.node.success-duration-ms`	Timer (ms)	Actual time spent executing user code (when the node ends with success phase).
`propeller.all.node.success-duration-ms-count`	Counter	Count of how many times a node success was reported.
`propeller.all.node.failure-duration-ms`	Timer (ms)	Actual time spent executing user code (when the node ends with failure phase).
`propeller.all.node.failure-duration-ms-count`	Counter	Count of how many times a node failure was reported.

All the above stats are automatically tagged with the following fields for further scoping. This includes user-produced stats. Users can also provide additional tags (or override tags) for custom stats.

Tag	Description
wf	This is the name of the workflow that was executing when this metric was emitted. `{{project}}:{{domain}}:{{workflow_name}}`

Flyte Task Container Metrics

To be built.

Default Project Dashboard

To be built.

User Stats With Flyte

The workflow parameters object that the SDK injects into the various tasks has a statsd handle that users should call to emit stats related to their workflows not captured by the default metrics. Please see the tutorial for more information. The usual caveats around cardinality apply of course.

Users are encouraged to avoid creating their own stats handlers. These can pollute the general namespace if not done correctly, and also can accidentally interfere with production stats of live services, causing pages and wreaking havoc in general. In fact, if you're using any libraries that emit stats, it's best to turn them off if possible.

2.3.3 Notifications

Notifications are customizable events triggered on workflow termination.

Notifications can be of three flavors:

- *Email*
- *PagerDuty*
- *Slack*

You can combine notifications in a single workflow to trigger for different combinations of recipients, domains and terminal phases.

Terminal workflow phases include:

- `WorkflowExecution.Phase.FAILED`
- `WorkflowExecution.Phase.TIMED_OUT`
- `WorkflowExecution.Phase.ABORTED`
- `WorkflowExecution.Phase.SUCCESS`

Future work

Currently the notification email subject and body are hard-coded. In the future passing these as customizable parameters in the notification proto message would be helpful.

2.3.4 Labels and Annotations

In Flyte, workflow executions are created as kubernetes resources. These can be extended with labels and annotations.

Labels and **annotations** are key value pairs which can be used to identify workflows for your own uses. Labels are meant to be used as identifing attributes whereas annotations are arbitrary, *non-identifying* metadata.

Using labels and annotations is entirely optional. They can be used to categorize and identify workflow executions.

Labels and annotations are optional parameters to launch plan and execution invocations. In the case an execution defines labels and/or annotations *and* the launch plan does as well, the execution spec values will be preferred.

Launch plan usage example

```
from flytekit.models.common import Labels, Annotations

@workflow
class MyWorkflow(object):
    ...

my_launch_plan = MyWorkflow.create_launch_plan(
    labels=Labels({"myexecutionlabel": "bar", ...}),
    annotations=Annotations({"region": "SEA", ...}),
    ...
)

my_launch_plan.execute(...)
```

Execution example

```
from flytekit.models.common import Labels, Annotations

@workflow
class MyWorkflow(object):
    ...

my_launch_plan = MyWorkflow.create_launch_plan(...)

my_launch_plan.execute(
    labels=Labels({"myexecutionlabel": "bar", ...}),
    annotations=Annotations({"region": "SEA", ...}),
    ...
)
```

2.3.5 Task Cache

Flyte provides the ability to cache the output of task executions in order to make subsequent executions faster. A well-behaved Flyte Task should generate deterministic output given the same inputs and task functionality. This is useful in situations where a user knows that many executions with the exact same inputs can occur. For example, your task may be periodically run on a schedule, run multiple times when debugging workflows, or commonly shared across different workflows but receive the same inputs.

Enable Caching For a Task

In order to enable your task to be cached, mark `cache=True` below:

```
@inputs(original=Types.String)
@outputs(hash=Types.String)
@python_task(cache=True, cache_version='1.0.0')
def hash_string_task(wf_params, original, hash)
   ...
```

A task execution is cached based on the **Project, Domain, cache_version, the task signature and inputs** associated with the execution of the task.

- *Project:* A task run under one project cannot use the cached task execution from another project. This could cause inadvertent results between project teams that could cause data corruption.
- *Domain:* For separation of test, staging, and production data, task executions are not shared across these environments.
- *cache_version:* When task functionality changes, you can change the cache_version of the task. Flyte will know not to use older cached task executions and create a new cache entry on the next execution.
- *Task signature:*: The cache is specific to the task signature that is associated with the execution. The signature is made up of task name, input parameter names/types and also the output parameter name/types.
- *Task input values*: A well-formed Flyte Task always produces deterministic outputs. This means given a set of input values, every execution should produce identical outputs. When a task execution is cached, the input values are part of the cache key.

Notice that task executions can be cached across different versions of the task. This is because a change in SHA does not neccessarily mean that it correlates to a change in task functionality.

Flyte provides several ways to break the old task execution cache, and cache new output:

- `cache_version`: this field indicates that the task functionality has changed. Flyte users can manually update this version and Flyte will cache the next execution instead of relying on the old cache.
- Task signature: If a Flyte user changes the task interface in any way (such as by adding, removing, or editing inputs/outputs), Flyte will treat that as a task functionality change. On the next execution, Flyte will run the task and store the outputs as new cached values.

2.3.6 Why roles?

Roles can be applied to launch plans (and the executions they create) to determine execution privileges. Out of the box, Flyte offers a way to configure permissions via IAM role or Kubernetes service account.

IAM role examples

Configure project-wide IAM roles by adding the following to your config:

```
[auth]
assumable_iam_role=arn:aws:iam::123accountid:role/my-super-cool-role
```

Alternatively, pass the role as an argument to `create_launch_plan`:

```
my_lp = MyWorkflow.create_launch_plan(assumable_iam_role=
↪'arn:aws:iam::123accountid:role/my-super-cool-role')
```

Kubernetes serviceaccount examples

Configure project-wide kubernetes serviceaccounts by adding the following to your config:

```
[auth]
kubernetes_service_account=my-kube-service-acct
```

Alternatively, pass the role as an argument to `create_launch_plan`:

```
my_lp = MyWorkflow.create_launch_plan(kubernetes_service_account='my-kube-service-acct
↪')
```

2.4 Flyte Task Types

2.4.1 Container Task

This is one of the low-level task types. It belongs to task category "SingleStepTask", and is defined using a container image and task interface (TypedInterface). TypedInterface consists of an input map and an output map. Python Task is one implementation of this type in Flyte Python SDK. The task name and inputs are passed to the container when it is started.

ContainerTask definition also includes RuntimeMetadata which points to the specific implementation of the "Container Task"(FlyteSDK or other).

Python Task

Python tasks are implemented on top of container tasks. It is a python function that needs to specify all its inputs using flyte input annotation @inputs and outputs using flyte output annotation @outputs. The function is not supposed to return a value but set all the outputs before exiting. The user container built during the build step is used for registration of this container task.

Usage

The following is an example of a simple Python task.

Listing 1: Python task example with inputs and outputs

```
@inputs(value_to_print=Types.Integer)
@outputs(out=Types.Integer)
@python_task(cache_version='1')
def add_one_and_print(workflow_parameters, value_to_print, out):
    added = value_to_print + 1
    workflow_parameters.logging.info("My printed value: {}".format(added))
    out.set(added)
```

This does a couple things.

- SDK uses the current container image to register this task
- SDK registers the container with special container entrypoint that knows how to run this task.

2.4.2 Hive Tasks

Hive tasks are an example of dynamic tasks. That is, they are a two-step task where the workflow container is first run to produce the queries, which are later executed using a Flyte plugin. This means that the text of the queries as well as the number of queries can be dynamic.

See the Hive tasks discussion in the generated API documentation for more information.

Usage

Basic Query Execution

The following is an example of a simple Hive query.

Listing 2: Simple Hive query example

```
@qubole_hive_task
def generate_simple_queries(wf_params):
    q1 = "SELECT 1"
    q2 = "SELECT 'two'"
    return [q1, q2]
```

This is a pretty simple query. Your queries will be run on Qubole, but nothing will happen with the output.

Query with Schema Integration

A more common and powerful pattern is to integrate querying along with the Flyte `Schema` type.

Listing 3: Hive query example with Schema integration

```
@outputs(hive_results=[Types.Schema()])
@qubole_hive_task(tags=['mytag'], cluster='flyte')
def generate_queries(wf_params, hive_results):
    q1 = "SELECT 1"
    q2 = "SELECT 'two'"
    schema_1, formatted_query_1 = Schema.create_from_hive_query(select_query=q1)
    schema_2, formatted_query_2 = Schema.create_from_hive_query(select_query=q2)

    hive_results.set([schema_1, schema_2])
    return [formatted_query_1, formatted_query_2]
```

This does a couple things.

- Your queries will be amended by the SDK before they are executed. That is, instead of `SELECT 1`, the actual query that will be run will be something like

  ```
  CREATE TEMPORARY TABLE 1757c8c0d7a149b79f2c202c2c78b378_tmp AS SELECT 1;
  CREATE EXTERNAL TABLE 1757c8c0d7a149b79f2c202c2c78b378 LIKE␣
  ↪1757c8c0d7a149b79f2c202c2c78b378_tmp STORED AS PARQUET;
  ALTER TABLE 1757c8c0d7a149b79f2c202c2c78b378 SET LOCATION
      \'s3://my-s3-bucket/ec/b78e1502cef04d5db8bef64a2226f707/\';
  INSERT OVERWRITE TABLE 1757c8c0d7a149b79f2c202c2c78b378
  SELECT *
  FROM 1757c8c0d7a149b79f2c202c2c78b378_tmp;
  DROP TABLE 1757c8c0d7a149b79f2c202c2c78b378;
  ```

  When a user's query runs, it's first selected into a temporary table, and then copied from the temporary table into the permanent external table. The external table is then dropped, which doesn't actually delete the just-queried data, but rather alleviates pressure on the Hive metastore.

- The task's output will have been bound a priori to the location that the Qubole Hive query will end up writing to, so that the rest of Flyte (downstream tasks for instance) will know about them.

Miscellaneous

Hive Execution Environment

Qubole is currently the primary execution engine for Hive queries, though it doesn't have to be. In fact, the `qubole_hive_task` decorator in the SDK is a refinement on the broader `hive_task` decorator.

2.4.3 Spark Task

Spark Task Type allows users to write a Spark Job which are then executed on the Kubernetes cluster via the Spark-Operator.

Setup

Spark on Kubernetes is containerized i.e. your own repo image is run as Spark driver and executors hence we need to have Spark and it's dependencies installed in the user's container.

One way to achieve this is to build relevant images for your use-case using Dockerfile which already exist as part of the Spark distribution. Flytekit also provides *flytekit_install_spark.sh* and *flytekit_spark_entrypoint.sh* which can be used to install spark dependencies/set the correct entrypoint for Spark Tasks in an user image.

Please refer to Spark Documentation for details on Spark on K8s.

Usage

The following is an example of a simple Spark job which calculates the value of Pi.

Listing 4: Simple Spark Job example

```
@spark_task(
    spark_conf={
      'spark.driver.memory': "1000M",
      'spark.executor.memory': "1000M",
      'spark.executor.cores': '1',
      'spark.executor.instances': '2',
    },
 )
 def hello_spark(workflow_parameters, spark_context, partitions, out):
     print("Starting Spark with Partitions: {}".format(partitions))
     n = 100000 * partitions
     count = spark_context.parallelize(range(1, n + 1), partitions).map(f).reduce(add)
     pi_val = 4.0 * count / n
     print("Pi val is :{}".format(pi_val))
     out.set(pi_val)
```

This is a pretty simple job. It calculates the value of Pi and sets it as the output for the task. The task takes in *spark_conf* and *hadoop_conf* where the user can provide any custom spark and hadoop config required for the Spark Task.

Flyte Workflow Demo - Spark Workflow is a Spark Workflow example which uses the Spark Task defined above.

2.4.4 Dynamic Tasks

Flyte offers many task types to support a varity of use-cases that are all based on statically defining what those tasks do beforehand. Dynamic Task type is the only exception to this rule where the system knows very little about what the nature of the workload that will be executed until execution time.

A Dynamic task is executed as follows:

- A generator step; This step runs like a *Container Task*. The expected outcome of this step is a dynamic job spec.
- Execute the dynamic job spec; The spec can contain any of the flyte supported workflow nodes. One or more of these nodes might contain other dynamic tasks. In which case, it'll get recursively executed.
- Assemble final outputs of the task.

Some of the potential use cases:

- Launching arbitrary workflows/tasks: You might build a workflow where one of its tasks chooses and executes a launchplan based on values passed as inputs to the task.

- Dynamically creating a workflow If the desired set of nodes/tasks that need to be executed is controlled by inputs, you can use dynamic tasks to build a workflow spec at execution time and yield that for it to be executed.

Listing 5: Dynamic task example

```
@inputs(tasks_count=Types.Integer)
@outputs(out=[Types.Integer])
@dynamic_task(cache_version='1')
def generate_others(workflow_parameters, tasks_count, out):
  res = []
  # Launch an arbitrary number of tasks based on the input tasks_count
  for i in range(0, tasks_count):
    task = other_task(index=i)
    yield task
    # Define how to assemble the final result
    res.append(task.outputs.other_task_out)

  # Define how to set the final result of the task
  out.set(res)
```

A few notes about how this task will run:

- This code will be executed only once. Based on the way outputs are declared, *Bindings* will be created to instruct the system on how to assemble the final outputs after all yielded tasks are executed.
- Manipulating Outputs of the yielded tasks is not supported. Think of this step as a Map Concept. If a simple reduce is required, it'll have to happen as a separate task that consumes the assembled outputs here.
- There is a restriction on the size of individual tasks outputs as well as on the final output of this task. If large outputs are desired, consider using Blob types.

2.4.5 Sidecar Tasks

Sidecar tasks can be used to bring up multiple containers within a single task. Sidecar tasks are defined using a Kubernetes pod spec but differ slightly in that the plugin task executor solely monitors the status of a user-specified primary container over the task lifecycle.

Installation

If you haven't already, install the sidecar extra from flytekit like so:

```
pip install flytekit[sidecar]
```

And assert that you have a dependency in your project on

```
k8s-proto>=0.0.2
```

Usage

Sidecar accept all arguments that ordinary *container tasks* accept. However, sidecar tasks require two additional arguments, `pod_spec` and `primary_container_name`

Pod Spec

Using the generated python protobuf code in flyteproto, a task can define a completely kubernetes-native pod spec that will be deployed as part of the sidecar task execution.

Primary container

This is a required name you use to distinguish your primary container. The code in the body of the task definition will be injected in the primary container. The pod spec you pass to the task definition does not necessarily need to include a container definition with the primary container, but if you'd like to modify the primary container by setting a shared volume mount for example, you can do so in the pod spec.

For primary containers defined in the pod spec, a few caveats apply. The following container fields will be overwritten at registration time and therefore are meaningless if set:

- image
- command
- args
- resource requirements
- env

Resource requirements and env will use the values set in the sidecar task definition and are therefore still customizable.

For example:

```
def get_pod_spec():
    my_pod_spec = generated_pb2.PodSpec()
    my_container = generated_pb2.Container(name="primary")
    # apply more customization to my_container as desired

    my_pod_spec.containers.extend([my_container])
    # apply more customization to my_pod_spec as desired

    return my_pod_spec


@inputs(in1=Types.String)
@outputs(out1=Types.String)
@sidecar_task(
    cpu_request='10',
    cpu_limit='20',
    environment={"foo": "bar"},
    pod_spec=get_pod_spec(),
    primary_container_name="primary",
)
def simple_sidecar_task(wf_params, in1, out1):
    print("Hi, {} I'll be run in a sidecar task!".format(in1))
    ...
```

For a trivial example of configuring multiple containers so that one writes to a shared volume mount and the second waits until it receives the message, see:

```
from __future__ import absolute_import
from __future__ import print_function

```

(continues on next page)

(continued from previous page)

```
import time
import os

from flytekit.sdk.tasks import sidecar_task
from k8s.io.api.core.v1 import generated_pb2


# A simple pod spec in which a shared volume is mounted in both the primary and␣
↪secondary containers. The secondary
# writes a file that the primary waits on before completing.
def generate_pod_spec_for_task():
    pod_spec = generated_pb2.PodSpec()

    primary_container = generated_pb2.Container(name="primary")

    secondary_container = generated_pb2.Container(
              name="secondary",
              image="alpine",
    )
    secondary_container.command.extend(["/bin/sh"])
    secondary_container.args.extend(["-c", "echo hi sidecar world > /data/message.txt
↪"])
    shared_volume_mount = generated_pb2.VolumeMount(
              name="shared-data",
              mountPath="/data",
    )
    secondary_container.volumeMounts.extend([shared_volume_mount])
    primary_container.volumeMounts.extend([shared_volume_mount])

    pod_spec.volumes.extend([generated_pb2.Volume(
               name="shared-data",
               volumeSource=generated_pb2.VolumeSource(
                   emptyDir=generated_pb2.EmptyDirVolumeSource(
                              medium="Memory",
                   )
               )
    )])
    pod_spec.containers.extend([primary_container, secondary_container])
    return pod_spec


@sidecar_task(
    pod_spec=generate_pod_spec_for_task(),
    primary_container_name="primary",
)
def my_sidecar_task(wfparams):
    # The code defined in this task will get injected into the primary container.
    while not os.path.isfile('/data/message.txt'):
        time.sleep(5)
```

CHAPTER 3

Administrator Docs

This section provides details on how to manage and scale Flyte deployments within your organization. To manage Flyte, we will first provide an overview of the architecture, follow that up with an installation section and then describe how to manage and scale it for your users or yourself.

3.1 Architecture

This document aims to demystify how Flyte's major components `FlyteIDL`, `FlyteKit`, `FlyteCLI`, `FlyteConsole`, `FlyteAdmin`, `FlytePropeller`, and `FlytePlugins` fit together at a high level.

3.1.1 FlyteIDL

In Flyte, entities like "Workflows", "Tasks", "Launch Plans", and "Schedules" are recognized by multiple system components. In order for components to communicate effectively, they need a shared understanding about the structure of these entities.

The Flyte IDL (Interface Definition Language) is where shared Flyte entities are defined. This IDL also defines the RPC service definition for the core Flyte API.

FlyteIDL uses the protobuf schema to describe entities. Clients are generated for Python, Golang, and JavaScript and imported by Flyte components.

3.1.2 Planes

Flyte components are separated into 3 logical planes. The planes are summarized here and explained in further detail below. The goal is that any of these planes can be replaced by an alternate implementation.

User Plane	The User Plane consists of all user tools that assist in interacting with the core Flyte API. These tools include the FlyteConsole, FlyteKit, and FlyteCLI.
Control Plane	The Control Plane implements the core Flyte API. It serves all client requests coming from the User Plane. It stores information such as current and past running workflows, and provides that information upon request. It also accepts requests to execute workflows, but offloads the work to the Data Plane.
Data Plane	The sole responsibility of the the Data Plane is to fulfill workflows. It accepts workflow requests from the Control Plane and guides the workflow to completion, launching tasks on a cluster of machines as necessary based on the workflow graph. It sends status events back to the control plane so the information can be stored and surfaced to end-users.

User Plane

In Flyte, workflows are represented as a Directed Acyclic Graph (DAG) of tasks. While this representation is logical for services, managing workflow DAGs in this format is a tedious exercise for humans. The Flyte User Plane provides tools to create, manage, and visualize workflows in a format that is easily digestible to users.

These tools include:

FlyteKit FlyteKit is an SDK that helps users design new workflows using the Python programming language. FlyteKit can parse the python code, compile it into a valid Workflow DAG, and submit it to Flyte to be executed.

FlyteConsole Flyte console provides the Web interface for Flyte. Users and administrators can use the console to view workflows, launch plans, schedules, tasks, and individual task executions. The console provides tools to visualize workflows, and surfaces relevant logs for debugging failed tasks.

FlyteCLI Flyte Command Line Interface provides interactive access to Flyte to launch and access Flyte workflows via terminal.

Control Plane

The Control Plane supports the core REST/gRPC API defined in FlyteIDL. User Plane tools like FlyteConsole and FlyteKit contact the control plane on behalf of users to store and retrieve information.

Currently, the entire control plane is handled by a single service called **FlyteAdmin**.

FlyteAdmin is stateless. It processes requests to create entities like Tasks, Workflows, and Schedules by persisting data in a relational database.

While FlyteAdmin serves the Workflow Exeuction API, it does not, itself, execute workflows. To launch workflow executions, FlyteAdmin sends the workflow DAG off to the DataPlane. For added scalability and fault-tolerance, FlyteAdmin can be configured to load-balance workflows across multiple isolated data-plane clusters.

Data Plane

The Data Plane is the engine that accepts DAGs, and fulfills workflow executions by launching tasks in the order defined by the graph. Requests to the Data Plane generally come via the control plane, and not from end-users.

In order to support compute-intensive workflows at massive scale, the Data Plane needs to launch containers on a cluster of machines. The current implementation leverages kubernetes for cluster management.

Unlike the user-facing control-plane, the Data Plane does not expose a traditional REST/gRPC API. To launch an execution in the Data Plane, you create a "flyteworkflow" resource in kubernetes. A "flyteworkflow" is a kubernetes Custom Resource (CRD) created by our team. This custom resource represents the flyte workflow DAG.

The core state machine that processes flyteworkflows is worker we call **FlytePropeller**.

FlytePropeller leverages the kubernetes operator pattern. It polls the kubernetes API, looking for newly created flyteworkflow resources. FlytePropeller understands the workflow DAG, and launches the appropriate kubernetes pods as needed to complete tasks. It periodically checks for completed tasks, launching downstream tasks until the workflow is complete.

Plugins

Each task in a flyteworkflow DAG has a specified **type**. The logic for fulfilling a task is determined by its task type. In the most basic case, FlytePropeller launches a single kubernetes pod to fulfill a task. More complex task types require workloads to be distributed across hundreds of pods.

The type-specific task logic is separated into isolated code modules that we call **plugins**. Each task type has an associated plugin that is responsible for handling tasks of its type. For each task in a workflow, FlytePropeller activates the appropriate plugin based on the task type in order to fullfill the task.

The Flyte team has pre-built plugins for Hive, Spark, and AWS Batch, and more. To support new use-cases, developers can create their own plugins and bundle them in their FlytePropeller deployment.

3.2 Installing Flyte

This doc explains how to install Flyte, starting with the simplest method, and expanding on it to increase scale and reliability.

3.2.1 Getting Started

Prerequisites

Kubernetes and its `kubectl` client are the only strict prerequisites to installing Flyte.

Kubernetes can be installed on your local machine to run Flyte locally, or in the cloud for a scalable multi-user setup. Some installation options are listed below.

Local:

- Minikube
- Docker for Mac

Cloud Providers:

- AWS EKS (Amazon)
- GCP GKE (Google)
- Azure AKS (Microsoft)

Once you have kubernetes set up and can access it with `kubectl cluster-info`, you're ready to deploy flyte.

Flyte has a few different deployment configurations. We'll start with the easiest, and expand on it to increase scale and reliability.

Sandbox Deployment

The simplest Flyte deployment is the "sandbox" deployment, which includes everything you need in order to use Flyte. The Flyte sandbox can be deployed with a single command

```
kubectl create -f https://raw.githubusercontent.com/lyft/flyte/master/deployment/
↪sandbox/flyte_generated.yaml
```

This deployment uses a kubernetes NodePort for Flyte ingress. Once deployed, you can access the Flyte console on any kubernetes node at `http://{{ any kubernetes node }}:30081/console` (note that it will take a moment to deploy).

For local deployments, this endpoint is typically http://localhost:30081/console.

WARNING: The sandbox deployment is not well suited for production use. Most importantly, Flyte needs access to an object store, and a PostgreSQL database. In the sandbox deployment, the object store and PostgreSQL database are each installed as a single kubernetes pod. These pods are sufficient for testing and playground purposes, but they not designed to handle production load. Read on to learn how to configure Flyte for production.

3.2.2 Handling Production Load

In order to handle production load, you'll want to replace the sandbox's object store and PostgreSQL database with production grade storage systems. To do this, you'll need to modify your Flyte configuration to remove the sandbox datastores and reference new ones.

Flyte Configuration

A Flyte deployment contains around 50 kubernetes resources. The Flyte team has chosen to use the "kustomize" tool to manage these configs. Take a moment to read the kustomize docs. Understanding kustomize will be important to modifying Flyte configurations.

The `/kustomize` directory in the flyte repository is designed for use with `kustomize` to tailor Flyte deployments to your needs. Important subdirectories are described below.

base The base directory contains minimal configurations for each Flyte component.

dependencies The dependencies directory contains deploy configurations for components like `PostgreSQL` that Flyte depends on.

These directories were designed so that you can modify them using `kustomize` to generate a custom Flyte deployment. In fact, this is how we create the `sandbox` deployment.

Understanding the sandbox deployment will help you to create your own custom deployments.

Understanding the Sandbox

The sandbox deployment is managed by a set of kustomize overlays that alter the `base` configurations to compose the sandbox deployment.

The sandbox overlays live in the kustomize/overlays/sandbox directory. There are overlays for each component, and a "flyte" overlay that aggregates the components into a single deploy file.

Component Overlays For each modified component, there is an kustomize overlay at `kustomize/overlays/sandbox/{{ component }}`. The overlay will typically reference the `base` for that component, and modify it to the needs of the sandbox.

Using kustomize "patches", we add or override specific configs from the `base` resources. For example, in the "console" overlay, we specify a patch in the kustomization.yaml. This patch adds memory and cpu limits to the console deployment config.

Each Flyte component requires at least one configuration file. The configuration files for each component live in the component overlay. For example, the FlyteAdmin config lives at kustomize/overlays/sandbox/admindeployment/flyteadmin_config.yaml. These files get included as Kubernetes configmaps and mounted into pods.

Flyte Overlay The `flyte` overlay is meant to aggregate the components into a single deployment file. The kustomization.yaml in that directory lists the components to be included in the deploy.

We run `kustomize build` against the `flyte` directory to generate the complete sandbox deployment yaml we used earlier to deploy Flyte sandbox.

Creating Your Own Deployment

Before you create a custom deployment, you'll need to install kustomize.

The simplest way to create your own custom deployment is to clone the sandbox deploy and modify it to your liking.

To do this, check out the `flyte` repo

```
git clone https://github.com/lyft/flyte.git
```

Copy the sandbox configuration to a new directory on your machine, and enter the new directory

```
cp -r flyte/kustomize/overlays/sandbox my-flyte-deployment
cd my-flyte-deployment
```

Since the `base` files are not in your local copy, you'll need to make some slight modifications to reference the `base` files from our GitHub repository.

```
find . -name kustomization.yaml -print0 | xargs -0 sed -i.bak 's~../../../base~github.
↪com/lyft/flyte/kustomize/base~'
find . -name kustomization.yaml -print0 | xargs -0 sed -i.bak 's~../../../
↪dependencies~github.com/lyft/flyte/kustomize/dependencies~'
find . -name *.bak | xargs rm
```

You should now be able to run kustomize against the `flyte` directory

```
kustomize build flyte > flyte_generated.yaml
```

This will generate a deployment file identical to the sandbox deploy, and place it in a file called `flyte_generated.yaml`

Going Beyond the Sandbox

Let's modify the sandbox deployment to use cloud providers for the database and object store.

Production Grade Database

The `FlyteAdmin` and `DataCatalog` components rely on PostgreSQL to store persistent records.

In this section, we'll modify the Flyte deploy to use a remote PostgreSQL database instead.

First, you'll need to set up a reliable PostgreSQL database. The easiest way achieve this is to use a cloud provider like AWS RDS, GCP Cloud SQL, or Azure PostgreSQL to manage the PostgreSQL database for you. Create one and make note of the username, password, endpoint, and port.

Next, remove old sandbox database by opening up the `flyte/kustomization.yaml` file and deleting database component.

```
- github.com/lyft/flyte/kustomize/dependencies/database
```

With this line removed, you can re-run `kustomize build flyte > flyte_generated.yaml` and see that the the postgres deployment has been removed from the `flyte_generated.yaml` file.

Now, let's re-configure `FlyteAdmin` to use the new database. Edit the `admindeployment/flyteadmin_config.yaml` file, and change the `storage` key like so

```
database:
  host: {{ your-database.endpoint }}
  port: {{ your database port }}
  username: {{ your_database_username }}
  password: {{ your_database_password }}
  dbname: flyteadmin
```

Do the same thing in `datacatalog/datacatalog_config.yaml`, but use the dbname `datacatalog`

```
database:
  host: {{ your-database.endpoint }}
  port: {{ your database port }}
  username: {{ your_database_username }}
  password: {{ your_database_password }}
  dbname: datacatalog
```

Note: *You can mount the database password into the pod and use the "passwordPath" config to point to a file on disk instead of specifying the password here*

Next, remove the "check-db-ready" init container from admindeployment/admindeployment.yaml. This check is no longer needed.

Production Grade Object Store

`FlyteAdmin`, `FlytePropeller`, and `DataCatalog` components rely on an Object Store to hold files.

In this section, we'll modify the Flyte deploy to use AWS S3 for object storage. The process for other cloud providers like GCP GCS should be similar.

To start, create an s3 bucket.

Next, remove the old sandbox object store by opening up the `flyte/kustomization.yaml` file and deleting the storage line.

```
- github.com/lyft/flyte/kustomize/dependencies/storage
```

With this line gone, you can re-run `kustomize build flyte > flyte_generated.yaml` and see that the sandbox object store has been removed from the `flyte_generated.yaml` file.

Next, open the configs `admindeployment/flyteadmin_config.yaml`, `propeller/config.yaml`, `datacatalog/datacatalog_config.yaml` and look for the `storage` configuration.

Change the `storage` configuration in each of these configs to use your new s3 bucket like so

```
storage:
  type: s3
  container: {{ YOUR-S3-BUCKET }}
```

(continues on next page)

(continued from previous page)

```
  connection:
    auth-type: accesskey
    access-key: {{ YOUR_AWS_ACCESS_KEY }}
    secret-key: {{ YOUR_AWS_SECRET_KEY }}
    region: {{ YOUR-AWS-REGION }}
```

Note: *To use IAM roles for authentication, switch to the "iam" auth-type.*

Next, open `propeller/plugins/config.yaml` and remove the default-env-vars (no need to replace them, the default behavior is sufficient).

Now if you re-run `kustomize build flyte > flyte_generated.yaml`, you should see that the configmaps have been updated.

Run `kubectl apply -f flyte_generated.yaml` to deploy these changes to your cluster for a production-ready deployment.

Dynamically Configured Projects

As your Flyte user-base evolves, adding new projects is as simple as registering them through the cli

```
flyte-cli -h {{ your-flyte-admin-host.com }} register-project --identifier␣
↪myuniqueworkflow --name FriendlyWorkflowName
```

A cron which runs at the cadence specified in flyteadmin config will ensure that all the kubernetes resources necessary for the new project are created and new workflows can successfully be registered and executed under the new project.

3.2.3 Scaling Beyond Kubernetes

As described in the high-level architecture doc, the Flyte Control Plane sends workflows off to the Data Plane for execution. The Data Plane fulfills these workflows by launching pods in kubernetes.

At some point, your total compute needs could exceed the limits of a single kubernetes cluster. To address this, you can deploy the Data Plane to several isolated kubernetes clusters. The Control Plane (FlyteAdmin) can be configured to load-balance workflows across these isolated Data Planes. This protects you from a failure in a single kubernetes cluster, and increases scalability.

First, you'll need to create additional kubernetes clusters. For this example, we'll assume you have 3 kubernetes clusters, and can access them all with `kubectl`. We'll call these clusters "cluster1", "cluster2", and "cluster3".

We want to deploy **just** the Data Plane to these clusters. To do this, we'll remove the DataPlane components from the `flyte` overlay, and create a new overlay containing **ony** the dataplane resources.

Data Plane Deployment

To create the "Data Plane only" overlay, lets make a `dataplane` subdirectory inside our main deployment directory (my-flyte-deployment). This directory will contain contain only the dataplane resources.

```
mkdir dataplane
```

Now, lets copy the `flyte` config into the dataplane config

```
cp flyte/kustomization.yaml dataplane/kustomization.yaml
```

Since the dataplane resources will live in the new deployment, they are no longer needed in the main `flyte` deployment. Remove the Data Plane resources from the flyte deploy by opening `flyte/kustomization.yaml` and removing everything in the `DATA PLANE RESOURCES` section.

Likewise, the User Plane / Control Plane resources are not needed in the dataplane deployment. Remove these resources from the dataplane deploy by opening `dataplane/kustomization.yaml` and removing everything in the `USER PLANE / CONTROL PLANE RESOURCES` section.

Now Run

```
kustomize build dataplane > dataplane_generated.yaml
```

You will notice that the only the Data Plane resources are included in this file.

You can point your `kubectl` context at each of the 3 clusters and deploy the dataplane with

```
kubectl apply -f dataplane_generated.yaml
```

User and Control Plane Deployment

In order for FlyteAdmin to create "flyteworkflows" on the 3 remote clusters, it will need a secret `token` and `cacert` to access each cluster.

Once you have deployed the dataplane as described above, you can retrieve the "token" and "cacert" by pointing your `kubectl` context each dataplane cluster and executing the following commands.

token `kubectl get secrets -n flyte | grep flyteadmin-token | awk '{print $1}' | xargs kubectl get secret -n flyte -o jsonpath='{.data.token}'`

cacert `kubectl get secrets -n flyte | grep flyteadmin-token | awk '{print $1}' | xargs kubectl get secret -n flyte -o jsonpath='{.data.ca\.crt}'`

These credentials will need to be included in the Control Plane. Create a new file `admindeployment/secrets.yaml` that looks like this

```
apiVersion: v1
kind: Secret
metadata:
  name: cluster_credentials
  namespace: flyte
type: Opaque
data:
  cluster_1_token: {{ cluster 1 token here }}
  cluster_1_cacert: {{ cluster 1 cacert here }}
  cluster_2_token: {{ cluster 2 token here }}
  cluster_2_cacert: {{ cluster 2 cacert here }}
  cluster_3_token: {{ cluster 3 token here }}
  cluster_3_cacert: {{ cluster 3 cacert here }}
```

Include the new `secrets.yaml` file in the `admindeployment` by opening `admindeployment/kustomization.yaml` and add the following line under `resources:` to include the secrets in the deploy

```
- secrets.yaml
```

Next, we'll need to attach these secrets to the FlyteAdmin pods so that FlyteAdmin can access them. Open `admindeployment/deployment.yaml` and add an entry under `volumes:`

```
volumes:
- name: cluster_credentials
  secret:
    secretName: cluster_credentials
```

Now look for the container labeled `flyteadmin`. Add a `volumeMounts` to that section.

```
volumeMounts:
- name: cluster_credentials
  mountPath: /var/run/credentials
```

This mounts the credentials inside the FlyteAdmin pods, however, FlyteAdmin needs to be configured to use these credentials. Open the `admindeployment/configmap.yaml` file and add a `clusters` key to the configmap, with an entry for each cluster

```
clusters:
- name: "cluster_1"
  endpoint: {{ your-cluster-1-kubeapi-endpoint.com }}
  enabled: true
  auth:
    type: "file_path"
    tokenPath: "/var/run/credentials/cluster_1_token"
    certPath: "/var/run/credentials/cluster_1_cacert"
- name: "cluster_2"
  endpoint: {{ your-cluster-2-kubeapi-endpoint.com }}
  auth:
    enabled: true
    type: "file_path"
    tokenPath: "/var/run/credentials/cluster_2_token"
    certPath: "/var/run/credentials/cluster_2_cacert"
- name: "cluster_3"
  endpoint: {{ your-cluster-3-kubeapi-endpoint.com }}
  enabled: true
  auth:
    type: "file_path"
    tokenPath: "/var/run/credentials/cluster_3_token"
    certPath: "/var/run/credentials/cluster_3_cacert"
```

Now re-run

```
kustomize build flyte > flyte_generated.yaml
```

You will notice that the Data Plane resources have been removed from the `flyte_generated.yaml` file, and your new configurations have been added.

Deploy the user/control plane to one cluster (you could use one of 3 existing clusters, or an entirely separate cluster).

```
kubectl apply -f flyte_generated.yaml
```

3.2.4 Optional Components

Scheduling

Scheduling component of Flyte allows workflows to run at a configured cadence. Currently, Flyte does not have a built in cron style scheduler. It uses external services to manage these schedules and launch executions of scheduled workflows at specific intervals.

3.2.5 Data Catalog

The Data Catalog component enables features like *Data Provenance*, *Data Lineage* and *Data Caching*. Data Catalog is enabled in Flyte by default.

3.2.6 Configure Flyte backend

Flyte has been designed to be extremely configurable. The entire backend has been written in Golang and uses flytestdlib module to manage configuration.

Common configuration across all backend components

All backend Flyte components are written in GoLang and use flytestdlib. Flytestdlib uses the common configuration system and is shared across the entire flyte-backend. The configuration used to configure the logging and metadata storage systems.

Logger Configuration

Logger can be configured using a common configuration that is shared across all components. The variables available for configuration are described in the **Config** struct in the source code. Documentation for this is available @ godoc:flytestdlib/logger.

Config Val	Type	Description
logger.formatter.type	string	Sets logging format type. (default "json")
logger.level	int	Sets the minimum logging level. (default 4)
logger.mute	Bool	Mutes all logs regardless of severity. Intended for benchmarks/tests only.
logger.show-source	Bool	Includes source code location in logs.

Metadata storage configuration

Metadata Storage can be configured using a common configuration that is shared across all components. The variables available for configuration are described in the **Config** struct in the code. Documentation for this is available @ godoc:flytestdlib/storage

Config Val	Type	Description
storage.cache.max_size_mbs	int	Maximum size of the cache where the Blob store data is cached in-memory. If not specified or set to 0, cache is not used
storage.cache.target_gc_percent	int	Sets the garbage collection target percentage.
storage.connection.access-key	string	Access key to use. Only required when authtype is set to accesskey.
storage.connection.auth-type	string	Auth Type to use [iam, accesskey]. (default "iam")
storage.connection.disable-ssl	Disables	SSL connection. Should only be used for development.
storage.connection.endpoint	string	URL for storage client to connect to.
storage.connection.region	string	Region to connect to. (default "us-east-1")
storage.connection.secret-key	string	Secret to use when accesskey is set.
storage.container	string	Initial container to create -if it doesn't exist-.'
storage.limits.maxDownloadMBs	int	Maximum allowed download size (in MBs) per call. (default 2)
storage.type	string	Sets the type of storage to configure [s3/minio/local/mem]. (default "s3")

Propeller Configuration

Admin Service Configuration

Propeller uses the following configuration to talk with FlyteAdmin service (control plane). It needs to talk to FlyteAdmin Servic

- Publishing events of progress in workflow execution
- Launching new launch plan executions as child executions

Propeller assumes a grpc endpoint available on the flyteadmin service, which is defined in flyteidl

Config Val	Type	Description
admin.endpoint	string	For admin types, specify where the uri of the service is located.
admin.insecure	Bool	Use insecure connection.

Event configuration

Propeller publishes events for every workflow execution and to prevent a large request spikes to FlyteAdmin service has a configurable tocken-bucket implementation. It can be configured using these parameters.

Config Val	Type	Description
event.capacity	int	The max bucket size for event recording tokens. (default 1000)
event.file-path	string	For file types, specify where the file should be located.
event.rate	int	Max rate at which events can be recorded per second. (default 500)
event.type	string	Sets the type of EventSink to configure [log/admin/file].

Propeller External Workflow Launcher configuration

Propeller can launch child-workflow/launch plan executions using a configurable flyteadmin service endpoint.

Config Val	Type	Description
propeller.admin-launcher.burst	int	Maximum burst for throttle (default 10)
propeller.admin-launcher.cacheSize	int	Maximum cache in terms of number of items stored. (default 10000)
propeller.admin-launcher.tps	int	The maximum number of transactions per second to flyte admin from this client. (default 5)
propeller.enable-admin-launcher	Bool	remote Workflow launcher to Admin

Catalog Service configuration

The Catalog service is an optional add-on to Flyte. It enables various features - refer to Catalog Service to understand the usecases. If available, it can be configured using.

Config Val	Type	Description
propeller.catalog-cache.endpoint	string	Endpoint for catalog service
propeller.catalog-cache.type	string	Catalog Implementation to use (default “noop”)

Propeller Core Configuration

This is the core configuration for propeller. It controls,

- Garbage collection
- Scale in terms of number of workers
- namespaces to observe for new workflows
- handling of system failures
- and other system level configuration

Config Val	Type	Description
propeller.downstream-eval-duration	string	Frequency of re-evaluating downstream tasks (default "60s")
propeller.gc-interval	string	Run periodic GC every 30 minutes (default "30m")
propeller.kube-config	string	Path to kubernetes client config file.
propeller.limit-namespace	string	Namespaces to watch for this propeller (default "all")
propeller.master	string	.
propeller.max-ttl-hours	int	Maximum number of hours a completed workflow should be retained. Number between 1-23 hours (default 23)
propeller.max-workflow-retries	int	Maximum number of retries per workflow (default 50)
propeller.metadata-prefix	string	MetadataPrefix should be used if all the metadata for Flyte executions should be stored under a specific prefix in CloudStorage. If not specified, the data will be stored in the base container directly.
propeller.metrics-prefix	string	An optional prefix for all published metrics. (default "flyte:")
propeller.prof-port	string	Profiler port (default "10254")
propeller.publish-k8s-events	Bool	events publishing to K8s events API.
propeller.workers	int	Number of threads to process workflows (default 2)
propeller.workflow-reeval-duration	string	Frequency of re-evaluating workflows (default "30s")

Propeller Leader election configuration

Config Val	Type	Description
propeller.leader-election.enabled	Bool	s/Disables leader election.
propeller.leader-election.lease-duration	string	Duration that non-leader candidates will wait to force acquire leadership. This is measured against time of last observed ack. (default "15s")
propeller.leader-election.lock-config-map.Name	string	Name of config-map used for leader.
propeller.leader-election.lock-config-map.Namespace	string	Namespace of config-map used for leader.
propeller.leader-election.renew-deadline	string	Duration that the acting master will retry refreshing leadership before giving up. (default "10s")
propeller.leader-election.retry-period	string	Duration the LeaderElector clients should wait between tries of actions. (default "2s")

Propeller Queue configuration

Propeller uses workqueues to manage the event loops and these queues can be configured using the following parameters.

Config Val	Type	Description
propeller.queue.batch-size	int	Number of downstream triggered top-level objects to re-enqueue every duration. -1 indicates all available. (default -1)
propeller.queue.batching-interval	string	Duration for which downstream updates are buffered (default "1s")
propeller.queue.queue.base-delay	string	base backoff delay for failure (default "10s")
propeller.queue.queue.capacity	int	Bucket capacity as number of items (default 100)
propeller.queue.queue.max-delay	string	Max backoff delay for failure (default "10s")
propeller.queue.queue.rate	int	Bucket Refill rate per second (default 10)
propeller.queue.queue.type	string	Type of RateLimiter to use for the WorkQueue (default "default")
propeller.queue.sub-queue.base-delay	string	base backoff delay for failure (default "10s")
propeller.queue.sub-queue.capacity	int	Bucket capacity as number of items (default 100)
propeller.queue.sub-queue.max-delay	string	Max backoff delay for failure (default "10s")
propeller.queue.sub-queue.rate	int	Bucket Refill rate per second (default 10)
propeller.queue.sub-queue.type	string	Type of RateLimiter to use for the WorkQueue (default "default")
propeller.queue.type	string	Type of composite queue to use for the WorkQueue (default "simple")

Propeller Commandline only parameters

These parameters are available when starting a propeller process.

Config Val	Type	Description
alsologtostderr	Bool	log to standard error as well as files
config	string	config file (default is $HOME/config.yaml)
errors.show-source	Bool	Indicates whether errors should include source code location information.
kubeconfig	string	Paths to a kubeconfig. Only required if out-of-cluster.
log_backtrace_at	Bool	trace location when logging hits line file:N, emit a stack trace (default :0)
log_dir	string	If non-empty, write log files in this directory
logtostderr	Bool	log to standard error instead of files
master	string	The address of the Kubernetes API server. Overrides any value in kubeconfig. Only required if out-of-cluster.
stderrthreshold	severity	logs at or above this threshold go to stderr (default 2)

Example Configuration

```
propeller:
  metadata-prefix: metadata/propeller
  workers: 4
  max-workflow-retries: 30
  workflow-reeval-duration: 30s
  downstream-eval-duration: 30s
  limit-namespace: "all"
  prof-port: 10254
  metrics-prefix: flyte
  enable-admin-launcher: true
  leader-election:
    lock-config-map:
      name: propeller-leader
      namespace: flyte
    enabled: true
    lease-duration: 15s
    renew-deadline: 10s
    retry-period: 2s
  queue:
    type: batch
    batching-interval: 2s
    batch-size: -1
    queue:
      type: bucket
      rate: 10
      capacity: 100
    sub-queue:
      type: bucket
      rate: 10
      capacity: 100
logger:
  show-source: true
  level: 5
storage:
  connection:
    access-key: minio
    auth-type: accesskey
    disable-ssl: true
    endpoint: http://minio.flyte.svc.cluster.local:9000
    region: us-east-1
    secret-key: miniostorage
  type: minio
  container: my-s3-bucket
event:
  type: admin
  rate: 500
  capacity: 1000
admin:
  endpoint: flyteadmin:81
  insecure: true
catalog-cache:
  endpoint: datacatalog:89
  type: datacatalog
  insecure: true
```

Plugin Configuration

Controlling the plugins available

Config Val	Type	Description
plugins.enabled-plugins	strings	List of enabled plugins, default value is to enable all plugins. (default [*])

Controlling log-links for user containers

Config Val	Type	Description
plugins.logs.cloudwatch-enabled	Bool	Cloudwatch Logging
plugins.logs.cloudwatch-log-group	string	Log group to which streams are associated.
plugins.logs.cloudwatch-region	string	AWS region in which Cloudwatch logs are stored.
plugins.logs.kubernetes-enabled	Bool	Kubernetes Logging
plugins.logs.kubernetes-url	string	Console URL for Kubernetes logs

Default values to provide kubernetes container executions

Individual plugins configuration

Refer to plugin documentation for understanding the configuration alternatives

Example Configuration

Top level Plugin configuration example

```
plugins:
  enabled-plugins:
    - container
    - spark
    - waitable
    - hiveExecutor
    - sidecar
  logs:
    kubernetes-enabled: true
    kubernetes-url: "http://localhost:30082"
  k8s:
    default-env-vars:
      - FLYTE_AWS_ENDPOINT: "http://minio.flyte:9000"
      - FLYTE_AWS_ACCESS_KEY_ID: minio
      - FLYTE_AWS_SECRET_ACCESS_KEY: miniostorage
```

Individual plugin configuration example

Spark

```
plugins:
  spark:
    spark-config-default:
      - spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version: "2"
      - spark.kubernetes.allocation.batch.size: "50"
      - spark.hadoop.fs.s3a.acl.default: "BucketOwnerFullControl"
      - spark.hadoop.fs.s3n.impl: "org.apache.hadoop.fs.s3a.S3AFileSystem"
      - spark.hadoop.fs.AbstractFileSystem.s3n.impl: "org.apache.hadoop.fs.s3a.S3A"
      - spark.hadoop.fs.s3.impl: "org.apache.hadoop.fs.s3a.S3AFileSystem"
      - spark.hadoop.fs.AbstractFileSystem.s3.impl: "org.apache.hadoop.fs.s3a.S3A"
      - spark.hadoop.fs.s3a.impl: "org.apache.hadoop.fs.s3a.S3AFileSystem"
      - spark.hadoop.fs.AbstractFileSystem.s3a.impl: "org.apache.hadoop.fs.s3a.S3A"
      - spark.hadoop.fs.s3a.multipart.threshold: "536870912"
      - spark.blacklist.enabled: "true"
      - spark.blacklist.timeout: "5m"
      - spark.task.maxfailures: "8"
```

Qubole

```
plugins:
  qubole:
    # Either create this file with your username with the real token, or set the␣
↪QUBOLE_API_KEY environment variable
    # See the secrets_manager.go file in the plugins repo for usage.  Since the dev/
↪test deployment of
    # this has a dummy QUBOLE_API_KEY env var built in, this fake path won't break␣
↪anything.
    quboleTokenPath: "/Path/To/QUBOLE_CLIENT_TOKEN"
    resourceManagerType: redis
    redisHostPath: redis-resource-manager.flyte:6379
    redisHostKey: mypassword
    quboleLimit: 10
```

Waitable

```
plugins:
  waitable:
    console-uri: http://localhost:30081/console
```

FlyteAdmin Configuration

Example config

```
logger:
  show-source: true
  level: 5
application:
  httpPort: 8088
  grpcPort: 8089
flyteadmin:
  roleNameKey: "iam.amazonaws.com/role"
  profilerPort: 10254
  metricsScope: "flyte:"
  metadataStoragePrefix:
    - "metadata"
```

(continues on next page)

(continued from previous page)

```
    - "admin"
  testing:
    host: http://flyteadmin
database:
  port: 5432
  username: postgres
  host: postgres
  dbname: postgres
  options: sslmode=disable
remoteData:
  region: "us-east-1"
  scheme: "local"
  signedUrls:
    durationMinutes: 3
storage:
  type: minio
  connection:
    access-key: minio
    auth-type: accesskey
    secret-key: miniostorage
    disable-ssl: true
    endpoint: http://minio.flyte.svc.cluster.local:9000
    region: us-east-1
  container: "my-s3-bucket"
task_resources:
  defaults:
    cpu: 100m
    memory: 100Mi
    storage: 5Mi
  limits:
    cpu: 2
    memory: 2G
    storage: 20Mi
task_type_whitelist:
  spark:
    - project: flytekit
    - project: flytetester
domains:
  - id: development
    name: development
  - id: staging
    name: staging
  - id: production
    name: production
  - id: domain
    name: domain
cluster_resources:
  templatePath: "/etc/flyte/clusterresource/templates"
  refresh: 5m
```

CHAPTER 4

Contributor Docs

Before reading the contributor docs, it's best to read the *Administrator Docs* docs to understand how Flyte's major components fit together.

The contributor provide low level implimentation details about how each Flyte component is architected, and explain how to extend the system with custom behavior.

4.1 Flyte System Components

These sections provide implementation details about specific flyte components. This information will be useful when contributing to a specific component.

4.1.1 FlyteAdmin

Admin Structure

FlyteAdmin serves the main Flyte API. It processes all client requests to the system. Clients include the FlyteConsole, which calls FlyteAdmin to list workflows, get execution details, etc., and FlyteKit, which calls FlyteAdmin to registering workflows, launch workflows etc.

Below, we'll dive into each component defined in admin in more detail.

RPC

FlyteAdmin uses the grpc-gateway library to serve incoming gRPC and HTTP requests with identical handlers. For a more detailed overview of the API, including request and response entities, see the admin service definition. The RPC handlers are a thin shim that enforce some request structure validation and call out to appropriate *manager*. methods to process requests.

Managers

The Admin API is broken up into entities:

- Executions
- Launch plans
- Node Executions
- Projects (and their respective domains)
- Task Executions
- Tasks
- Workflows

Each API entity has an entity manager in FlyteAdmin reposible for implementing business logic for the entity. Entity managers handle full validation for create, update and get requests, and data persistence in the backing store (see the *repository* section).

Additional Components

The managers utilize additional components to process requests. These additional components include:

- **:ref:'workflow engine <components-admin-workflowengine>'**: compiles workflows and launches workflow executions from launch plans.
- **:ref:'data <components-admin-data>' (remote cloud storage)**: offloads data blobs to the configured cloud provider.
- **:ref:'runtime <components-admin-config>'**: loads values from a config file to assign task resources, initialization values, execution queues and more.
- **:ref:'async processes <components-admin-async>'**: provides functions for scheduling and executing workflows as well as enqueing and triggering notifications

Repository

Serialized entities (tasks, workflows, launch plans) and executions (workflow-, node- and task-) are stored as protos defined here. We use the excellent gorm library to interface with our database, which currently supports a postgres implementation. The actual code for issuing queries with gorm can be found in the gormimpl directory.

Models

Database models are defined in the models directory and correspond 1:1 with database tables[0].

The full set of database tables includes:

- executions
- execution_events
- launch_plans
- node_executions

[0] Unfortunately, given unique naming constraints, some models are redefined in migration_models to guarantee unique index values.

- node_execution_events
- tasks
- task_executions
- workflows

These database models inherit primary keys and indexes as defined in the corresponding models file.

The repositories code also includes transformers. These convert entities from the database format to a response format for the external API. If you change either of these structures, you will find you must change the corresponding transformers.

Component Details

This section dives into detail for each top-level directories defined in `pkg/`.

Asynchronous Components

Notifications and schedules are handled by async routines that are reponsible for enqueing and subsequently processing dequeued messages.

Flyteadmin uses the gizmo toolkit to abstract queueing implementation. Gizmo's pubsub library offers implementations for Amazon SNS/SQS, Google's Pubsub, Kafka topics and publishing over HTTP.

For the sandbox development, no-op implementations of the notifications and schedule handlers are used to remove external cloud dependencies.

Common

As the name implies, `common` houses shared components used across different flyteadmin components in a single, top-level directory to avoid cyclic dependencies. These components include execution naming and phase utils, query filter definitions, query sorting definitions, and named constants.

Data

Data interfaces are primarily handled by the storage library implemented in flytestdlib. However, neither this nor the underlying stow library expose HEAD support so the data package in admin exists as the layer responsible for additional, remote data operations.

Errors

The errors directory contains centrally defined errors that are designed for compatibility with gRPC statuses.

Runtime

Values specific to the flyteadmin application as well as task and workflow registration and execution are configured in the runtime directory. These interfaces expose values configured in the `flyteadmin` top-level key in the application config.

Workflowengine

This directory contains interfaces to build and execute workflows leveraging flytepropeller compiler and client components.

Admin Service

Making Requests to FlyteAdmin

Adding request filters

We use gRPC Gateway to reverse proxy http requests into gRPC. While this allows for a single implementation for both HTTP and gRPC, an important limitation is that fields mapped to the path pattern cannot be repeated and must have a primitive (non-message) type. Unfortunately this means that repeated string filters cannot use a proper protobuf message. Instead use the internal syntax shown below:

```
func(field,value) or func(field, value)
```

For example, multiple filters would be appended to an http request:

```
?filters=ne(version, TheWorst)+eq(workflow.name, workflow)
```

Timestamp fields use the RFC3339Nano spec (ex: “2006-01-02T15:04:05.999999999Z07:00”)

The fully supported set of filter functions are

- contains
- gt (greater than)
- gte (greter than or equal to)
- lt (less than)
- lte (less than or equal to)
- eq (equal)
- ne (not equal)
- value_in (for repeated sets of values)

“value_in” is a special case where multiple values are passed to the filter expression. For example:

```
value_in(phase, 1;2;3)
```

Filterable fields vary based on entity types:

- Task
  - project
  - domain
  - name
  - version
  - created_at
- Workflow

  - project
  - domain
  - name
  - version
  - created_at
- Launch plans
  - project
  - domain
  - name
  - version
  - created_at
  - updated_at
  - workflows.{any workflow field above} (for example: workflow.domain)
  - **state (you must use the integer enum e.g. 1)**
    * States are defined in launch_plan.proto.
- Executions (Workflow executions)
  - project
  - domain
  - name
  - workflow.{any workflow field above} (for example: workflow.domain)
  - launch_plan.{any launch plan field above} (for example: launch_plan.name)
  - **phase (you must use the upper-cased string name e.g. RUNNING)**
    * Phases are defined in execution.proto.
  - execution_created_at
  - execution_updated_at
  - duration (in seconds)
  - **mode (you must use the integer enum e.g. 1)**
    * Modes are defined in execution.proto.
- Node Executions
  - node_id
  - execution.{any execution field above} (for example: execution.domain)
  - **phase (you must use the upper-cased string name e.g. QUEUED)**
    * Phases are defined in execution.proto.
  - started_at
  - node_execution_created_at
  - node_execution_updated_at

  - duration (in seconds)
- Task Executions
  - retry_attempt
  - task.{any task field above} (for example: task.version)
  - execution.{any execution field above} (for example: execution.domain)
  - node_execution.{any node execution field above} (for example: node_execution.phase)
  - **phase (you must use the upper-cased string name e.g. SUCCEEDED)**
    * Phases are defined in execution.proto.
  - started_at
  - task_execution_created_at
  - task_execution_updated_at
  - duration (in seconds)

Putting it all together

If you wanted to do query on specific executions that were launched with a specific launch plan for a workflow with specific attributes, you could do something like:

```
gte(duration, 100)+value_in(phase,RUNNING;SUCCEEDED;FAILED)+eq(lauch_plan.project,␣
↪foo)
+eq(launch_plan.domain, bar)+eq(launch_plan.name, baz)
+eq(launch_plan.version, 1234)
+lte(workflow.created_at,2018-11-29T17:34:05.000000000Z07:00)
```

Adding sorting to requests

Only a subset of fields are supported for sorting list queries. The explicit list is below:

- ListTasks
  - project
  - domain
  - name
  - version
  - created_at
- ListTaskIds
  - project
  - domain
- ListWorkflows
  - project
  - domain
  - name

  - version
  - created_at
- ListWorkflowIds
  - project
  - domain
- ListLaunchPlans
  - project
  - domain
  - name
  - version
  - created_at
  - updated_at
  - **state (you must use the integer enum e.g. 1)**
    * States are defined in launch_plan.proto.
- ListWorkflowIds
  - project
  - domain
- ListExecutions
  - project
  - domain
  - name
  - **phase (you must use the upper-cased string name e.g. RUNNING)**
    * Phases are defined in execution.proto.
  - execution_created_at
  - execution_updated_at
  - duration (in seconds)
  - **mode (you must use the integer enum e.g. 1)**
    * Modes are defined execution.proto.
- ListNodeExecutions
  - node_id
  - retry_attempt
  - **phase (you must use the upper-cased string name e.g. QUEUED)**
    * Phases are defined in execution.proto.
  - started_at
  - node_execution_created_at
  - node_execution_updated_at

    - duration (in seconds)

- ListTaskExecutions
    - retry_attempt
    - **phase (you must use the upper-cased string name e.g. SUCCEEDED)**
        * Phases are defined in execution.proto.
    - started_at
    - task_execution_created_at
    - task_execution_updated_at
    - duration (in seconds)

Sorting syntax

Adding sorting to a request requires specifying the `key`, e.g. the attribute you wish to sort on. Sorting can also optionally specify the direction (one of `ASCENDING` or `DESCENDING`) where `DESCENDING` is the default.

Example sorting http param:

```
sort_by.key=created_at&sort_by.direction=DESCENDING
```

Alternatively, since descending is the default, the above could be rewritten as

```
sort_by.key=created_at
```

4.1.2 What is Data Catalog?

Data Catalog (https://github.com/lyft/datacatalog) is a service for indexing parameterized, strongly-typed data artifacts across revisions. It allows for clients to query artifacts based on meta information and tags.

How Flyte Memoizes Task Executions on Data Catalog

Flyte memoizes task executions by creating artifacts in Data Catalog and associating meta information regarding the execution with the artifact. Let's walk through what happens when a task execution is cached on Data Catalog.

Every task instance is represented as a DataSet:

```
Dataset {
   project: Flyte project the task was registered in
   domain: Flyte domain for the task execution
   name: flyte_task-<taskName>
   version: <cache_version>-<hash(input params)>-<hash(output params)>
}
```

Every task execution is represented as an Artifact in the Dataset above:

```
Artifact {
   id: uuid
   Metadata: [executionName, executionVersion]
   ArtifactData: [List of ArtifactData]
}
```

(continues on next page)

(continued from previous page)

```
ArtifactData {
   Name: <output-name>
   value: <offloaded storage location of the literal>
}
```

To retrieve the Artifact, we tag the Artifact with a hash of the input values for the memoized task execution.

```
ArtifactTag {
   Name: flyte_cached-<unique hash of the input values>
}
```

When caching an execution, Flyte propeller will:

1. Create a dataset for the task
2. Create an artifact that represents the execution, along with the artifact data that represents the execution output
3. Tag the artifact with a unique hash of the input values

When checking to see if the task execution is memoized Flyte Propeller will:

1. Compute the tag by computing the hash of the input
2. Check if a tagged artifact exists with that hash
   a. If it does we have a cache hit and Propeller can skip the task execution.
   b. If an artifact is not associated with the tag, Flyte Propeller needs to run the task.

4.1.3 Flyte Console

This is the web UI for the Flyte platform.

Running flyteconsole

Install Dependencies

Running flyteconsole locally requires NodeJS and yarn. Once these are installed, all of the dependencies can be installed by running `yarn` in the project directory.

Environment variables

Before we can run the server, we need to set up an environment variable or two.

`ADMIN_API_URL` (default: window.location.origin)

The Flyte console displays information fetched from the Flyte Admin API. This environment variable specifies the host prefix used in constructing API requests.

Note: this is only the host portion of the API endpoint, consisting of the protocol, domain, and port (if not using the standard 80/443).

This value will be combined with a suffix (such as `/api/v1`) to construct the final URL used in an API request.

Default Behavior

In most cases, `flyteconsole` will be hosted in the same cluster as the Admin API, meaning that the domain used to access the console is the same value used to access the API. For this reason, if no value is set for `ADMIN_API_URL`, the default behavior is to use the value of *window.location.origin*.

`BASE_URL` (default: `undefined`)

This allows running the console at a prefix on the target host. This is necessary when hosting the API and console on the same domain (with prefixes of `/api/v1` and `/console` for example). For local development, this is usually not needed, so the default behavior is to run without a prefix.

`CORS_PROXY_PREFIX` (default: `/cors_proxy`)

Sets the local endpoint for *CORS request proxying*.

Run the server

To start the local development server, run `yarn start`. This will spin up a Webpack development server, compile all of the code into bundles, and start the NodeJS server on the default port (3000). All requests to the NodeJS server will be stalled until the bundles have finished. The application will be accessible at http://localhost:3000 (if using the default port).

Development

Storybook

This project has support for Storybook. Component stories live next to the components they test, in a `__stories__` directory, with the filename pattern `{Component}.stories.tsx`.

You can run storybook with `npm run storybook`, and view the stories at http://localhost:9001.

Protobuf and the Network tab

Communication with the Flyte Admin API is done using Protobuf as the request/response format. Protobuf is a binary format, which means looking at responses in the Network tab won't be very helpful. To make debugging easier, each network request is logged to the console with it's URL followed by the decoded Protobuf payload. You must have debug output enabled (on by default in development) to see these messages.

Debug Output

This application makes use of the debug libary to provide namespaced debug output in the browser console. In development, all debug output is enabled. For other environments, the debug output must be enabled manually. You can do this by setting a flag in localStorage using the console: `localStorage.debug = 'flyte:*'`. Each module in the application sets its own namespace. So if you'd like to only view output for a single module, you can specify that one specifically (ex. `localStorage.debug = 'flyte:adminEntity'` to only see decoded Flyte Admin API requests).

CORS Proxying

In the common hosting arrangement, all API requests will be to the same origin serving the client application, making CORS unnecessary. For any requests which do not share the same `origin` value, the client application will route requests through a special endpoint on the NodeJS server. One example would be hosting the Admin API on a different domain than the console. Another example is when fetching execution data from external storage such as S3. This is done to minimize the amount of extra configuration required for ingress to the Admin API and data storage, as well as to simplify local development of the console without the need to grant CORS access to `localhost`.

The requests and responses are piped through the NodeJS server with minimal overhead. However, it is still recommended to host the Admin API and console on the same domain to prevent unnecessary load on the NodeJS server and extra latency on API requests due to the additional hop.

4.2 Flyte Specification Language

Flyte at the core consists of a specification language that all the components of the system can interact with. The specification language is written in Protocol Buffers. This allows for a very portable language model. An implementation of the language makes interaction with the system easier and brings the power of Flyte to the user. A full implementation of the language is provided in Python SDK as well as generated Golang types, C++ and JAVA.

This is a reference manual for Flyte Specification Language. For more information on how to use the provided SDK, please refer to SDK manual.

4.2.1 Elements

Types Flyte Spec Language defines a flexible typing system. There are broadly three sets of types; primitives (e.g. ints, float, bool, string), collections (lists and maps) and use-case specific types (e.g. Schema, Blob, Struct).

Collection types support any dimensionality (e.g. a list of map of list of strings).

Note: Flyte collection types are invariant. i.e. a list of Cat is not a subtype of list of Animal.

For reference documentation on types, refer to Types

Literals Literals are bound values for specific types. E.g. *5* is a literal of primitive type int with value *5*

For reference documentation on literals, refer to Types

Variables Variables are named entities that have a type. Variables are used to name inputs and outputs of execution units (tasks, wfs… etc.) as well as referencing outputs of other execution units.

For reference documentation on variables, refer to Variable

Interface Every execution unit in Flyte can optionally define an interface. The interface holds information about the inputs and output variables. In order for **task B** to consume output **X** of **task A**, it has to declare an input variable of the same type (not necessarily the same name).

For reference documentation on interfaces, refer to Interfaces

Tasks TaskTemplate is a system entity that describes common information about the task for the rest of the system to reason about. TaskTemplates contain:

- An *identifier* that globally identifies this task and allows other entities to reference it.
- A string type that is used throughout the system to customize the behavior of the task execution (e.g. launch a container in K8s, execute a SQL query… etc.) as well as how it gets rendered in the UI.
- An *interface* that can be used to type check bindings as well as compatibility of plugging the task in a larger workflow.
- An optional container that describes the versioned container image associated with the task.

- An optional custom Struct used by various plugins to carry information used to customize this task's behavior.

For concept documentation on tasks, refer to *task concepts*. For reference documentation on tasks, refer to Tasks

Bindings Bindings define how a certain input of a task should receive its *literal* value. Bindings can be static (assigned at compile/registration time to a literal) or can be references to outputs of other nodes.

A Binding of an input of type collection of strings can either be bound to an output of the same type of a different node or else individual items can be bound to outputs of type string of other tasks.

e.g. .. code-block:

```
let taskA return output S of type String
let taskB return output SList of type list of string
let taskC takes input i of type list of string

// Bind the entire input to the entire output.
taskC.i <- taskB.SList

// Bind individual items
taskC.i[0] <- taskA.S
taskC.i[1] <- taskA.S
```

For reference documentation on bindings, refer to Types

Identifiers An identifier is a globally unique structure that identifies an entity in the system. Varrious entities have different identifiers. Tasks, Workflows and Launchplans are all globally identified with an identifier.

For reference documentation on identifiers, refer to Indentifier

Conditional Expressions Flyte Spec Language supports conditional expressions of two types; logical and comparison expressions. Expressions are represented as a tree of boolean expressions.

Logical Expressions A logical expression is expressed as a logical operator (only AND and OR operators are supported) and two conditional expressions

Comparison Expressions A comparison expression is expressed as a comparison operator (Equal, Not Equal, Greater Than, Greater Than or Equal, Less Than or Less Than or Equal) and two operands.

An operand can either be a *primitive* or a variable name that exists as an input to the node where the expression is evaluated.

For reference documentation on conditions, refer to Indentifier

Nodes Nodes are encapsulations around any execution unit (task, workflow, launchplan… etc.) that abstracts away details about the execution unit and allows interaction with other nodes with minimal information.

Nodes define *how to bind* the inputs of the underlying execution unit (e.g. task). They can also alter how outputs are exposed by providing output aliases to some or all of the outputs variables.

Multiple nodes can reference the same execution unit with different input bindings.

Dependencies between nodes is driven by bindings; e.g. NodeC depends on data from NodeA, therefore the system will wait to execute NodeC until NodeA has successfully finished executing. Nodes can optionally define execution dependencies that are not expressed in bindings.

For more information about the different types of nodes, please refer to *node concepts*. For reference documentation on nodes, refer to Nodes

Workflows Workflows represent an execution scope in the system. A workflow is a directed-acyclic-graph that describes what steps need to be executed, in what order as well as which steps need to consume data from other steps.

For concept documentation on workflows, refer to *node concepts*. For reference documentation on workflows, refer to Workflows

4.2.2 Compiler

Flyte system requires compiled Workflows to execute. A CLI is provided for convenience to compile locally. Flyte automatically compiles all registered tasks and workflows. The compilation process validates that all nodes and tasks interfaces match up. It then pre-computes the actual execution plan and serializes the compiled workflow.

4.2.3 Properties of types and values

Type Identity Two types are equal if and only if their Protocol Buffers representation is identical.

Assignability Flyte types are invariant. Two variables can be assigned to each other if and only if their types are identical.

4.2.4 Extensibility

New types Flyte types can be extended in two ways:

A customization of an existing type e.g. Creating a URL type can be represented as a Literal Type String (that contains the final URL). As far as the task interface is concerned, its output is of type String. A subsequent task can then have an input of type string to bind to that output.

Because this approach localizes the visibility of the new type to the one task that produced it, it's hard to enforce any type checks (or validation for URL format… etc.) at the system layer.

This approach won't always satsify more complex types' needs.

Using generic types Flyte offers a literal type *STRUCT* that allows the passing of any valid Struct as a value. This approach is very powerful to pass other Protocol Buffers messages or custom types (represented in JSON) that only *plugins* knows about.

Note: Flyte doesn't yet support subtyping, or custom strongly typed structs.

4.3 Extending Flyte

4.3.1 Custom Tasks

Writing logic for your own task types is the most common way to extend Flyte. In fact, Flyte ships with several extensions by default. These are tasks like the Qubole-run Hive queries or K8s-run Spark tasks, which were critical to Lyft's internal deployment of Flyte but aren't part of Flyte's core.

Extending the IDL

Writing your own task will likely start with adding your own IDL message, which will look something like this protos/flyteidl/plugins/sidecar.proto or protos/flyteidl/plugins/qubole.proto. Your custom task's proto message can reference other objects like in the Qubole case, but ultimately it has to be one message class.

An instance of this message class will be included alongside the rest of the `TaskTemplate` (in the `custom` field) for a given task. Your message should include all the additional information that the execution layer of Flyte will need to execute the task. That is, you don't need to worry about the container image, cpu/memory resources, input/output types, etc. since that is all covered in the normal task definition. You only need to worry about the custom information for your task. Technically, if your custom task doesn't need any additional information whatsoever, you can skip this step.

Extending the SDK

The next step is to write a task handler on the SDK side. That is, now that we have the definition of what your custom task will need, we need a way for users to write that task in Python, and then transform those tasks into task specifications containing that Protobuf message. Continuing with the above examples, we can look at how the SDK bits are built for the sidecar task and the Qubole Hive task.

Broadly, the steps are:

1. Define a task type. Concretely, this is just the string here protos/flyteidl/core/tasks.proto#L92. As mentioned above, technically you do not need an IDL if for some reason your custom task has no additional information whatsoever. But even in that case, you'll need a new task type string here. This is the key that the execution plane will reference to decide how to run your task.
2. Create a class for your custom task that wraps the base task class (`flytekit.common.tasks.sdk_runnable.SdkRunnableTask` or just `flytekit.common.tasks.task.SdkTask` if a decorator is not required).
3. Optionally, create a decorator to make it simple to define user code.

Qubole Hive Example

The Hive task is slightly more complicated in that it produces a futures file, but the basic steps are the same.

1. First write a class that subclasses either the `SdkRunnableTask` or `SdkTask` like so flytekit/common/tasks/hive_task.py#L27
2. Override the execute method to have the behavior that you want to see. In this case, we're running the user code, and then compiling the output of the user's code into our futures file. * Also, an instance of the custom Protobuf message defined in the IDL should be created (if you need it), and added here.
3. Create a decorator for your task like this flytekit/sdk/tasks.py#L623.

Ultimately, FlyteKit is a large wrapper around the Flyte IDL objects. That is, its primary function is to translate user Python code into the Flyte component protobufs, sometimes in multiple stages like in the Hive example, that the Flyte engine and control plane can then understand.

Extending Propeller

Flyte plugins extend the Flyte execution plane with custom behavior for special tasks. When Flyte Propeller begins to run a task, it will look at the task type that you've defined and invoke the appropriate plugin. Spark tasks and Qubole Hive tasks are examples of tasks that are run by plugins.

Structure of Plugins

At a high level, a Flyte Propeller plugin is something that satisfies the `Executor` interface specified in go/tasks/v1/types/task.go. The plugin's initialization code should register itself against the aforementioned task type string. When Propeller comes across a task with this type, the plugin will be invoked. Be cognizant that a plugin runs as a singleton in the engine.

One of the important objects to understand is the `TaskContext`. This interface will be an object created by Propeller, and supplied to your plugin code. Most importantly, the `GetCustomState()` function returns a custom struct that is integral to the cycle of your task's execution.

You supply the initial value of this custom struct as the output of your `StartTask` call. On each call of the check loop thereafter, you get the version of the custom state that you returned before. Since this is the only state that is stored in a durable store (etcd), it should be your source of truth.

Note: Keep in mind that Flyte Propeller can restart at any time, which means your plugin can restart at any time. This custom state is the only state that your plugin can rely on.

Note that this custom state is different than the custom IDL object that you previously defined. The IDL message should be thought of as data describing the task itself whereas this customstate should be thought of as a way to keep track of state during execution of your task.

Note: Note that while the `CustomState` object returned by the `GetCustomState()` function is a `map[string]interface{}`, those interface values are not directly convertible to your Golang custom state objects. That is, they need to be first marshaled from JSON into bytes, and then unmarshaled from bytes back into your object, like so: go/tasks/v1/qubole/qubole_work.go#L187.

Task Initialization

The `StartTask` function is only called once and will be called with the task template containing the custom IDL struct if you chose to create one. You are only given this task template on this one `StartTask` call, so be sure your plugin code retrieves all the information from it that's necessary to complete the task's execution. For the Qubole Hive plugin for example, the queries to be run are copied from the custom IDL object into the custom state object.

Task Updates

This is the function that will be called periodically by Flyte Propeller, and is responsible for deciding how your custom task is progressing. Note that while the task template is there in the function signature, it is not actually used and will always be nil. This was an unfortunate optimization that had to be made to save on S3 access times.

Please refer to the generated documentation for a brief discussion of the other methods of the interface.

Background Updates

Often you'll need something to monitor the progress of your execution. To this end, some plugins (like our Qubole Hive plugin and the waitable task plugin) make use of an AutoRefreshCache, to which you can specify a sync function. This sync function will be periodically run on all objects in your cache. See the cache's documentation for more information.

4.4 Contributing to Docs

Documentation for Flyte spans a few repos, and is hosted on GitHub Pages. The contents of the `docs/` folder on the `master` branch are hosted on GitHub Pages. See GH Pages documentation for more information.

Documentation that the steps below will compile come from:

- This repository, comprising written RST pages
- RST files generated from the Flyte IDL repository
- RST files generated from the Flytekit Python SDK

4.4.1 Building

In order to create this set of documentation run:

```
$ make update_ref_docs && make all_docs
```

What happens is: * `./generate_docs.sh` runs. All this does is create a temp directory and clone the two aforementioned repos. * The Sphinx builder container will run with files from all three repos (the two cloned one and this one) mounted.

- It will generate RST files for Flytekit from the Python source code
- Copy RST files from all three repos into a common location
- Build HTML files into the `docs/` folder

Please then commit the newly generated files before merging your PR. In the future we will invest in CI to help with this.

4.4.2 Notes

We aggregate the doc sources in a single `index.rst` file so that `flytekit` and the Flyte IDL HTML pages to be together in the same index/table of contents.

There will be no separate Flyte Admin, Propeller, or Plugins documentation generation. This is because we have a *Contributor Docs* guide and high level usage/architecture/runbook documentation should either there, or into the administrator's guide.