Refined Program States

Checklist

- [x] User Stories Documented
- [ ] User Stories Reviewed
- [ ] Design Reviewed
- [ ] APIs reviewed
- [ ] Release priorities assigned
- [ ] Test cases reviewed
- [ ] Blog post

Introduction

Programs on CDAP have broad state definitions that don't clearly indicate what the program is doing during execution. We want to have clearer definitions for our program states to accurately reflect its true behavior during a program run.

Goals

- Redefine program statuses to accurately represent the current status of any program at any time.

User Stories

1. As a user with access to a small cluster, I want to see when my workflow is in a state of acquiring resources so that I know why my program is taking so long to begin running.
2. As a user, if I stop a flow, I want to see the flow marked as 'stopped', and not killed or failed
3. As a user, when I start a service, I want to be able to send requests to it as soon as the the service is running.
4. As a user, if I kill a program through the YARN resource manager, I want to see the same state reflected on CDAP.
5. As a user, when I start a Spark job, I should see the Spark job run only after it has finished requesting resources so that I know when my Spark job is actually running.

Requirements

1. Add more granular state definitions of programs to aid in debugging and clearer program state distinction.
2. Add transitions between these granular state definitions at the relevant sections in the code so that program states accurately represent the current status of the program.
3. Use TMS to broadcast any user-facing program status transitions so that other services can subscribe to these notifications.

See the parent and related JIRAs related to these issues here: https://issues.cask.co/browse/CDAP-4157

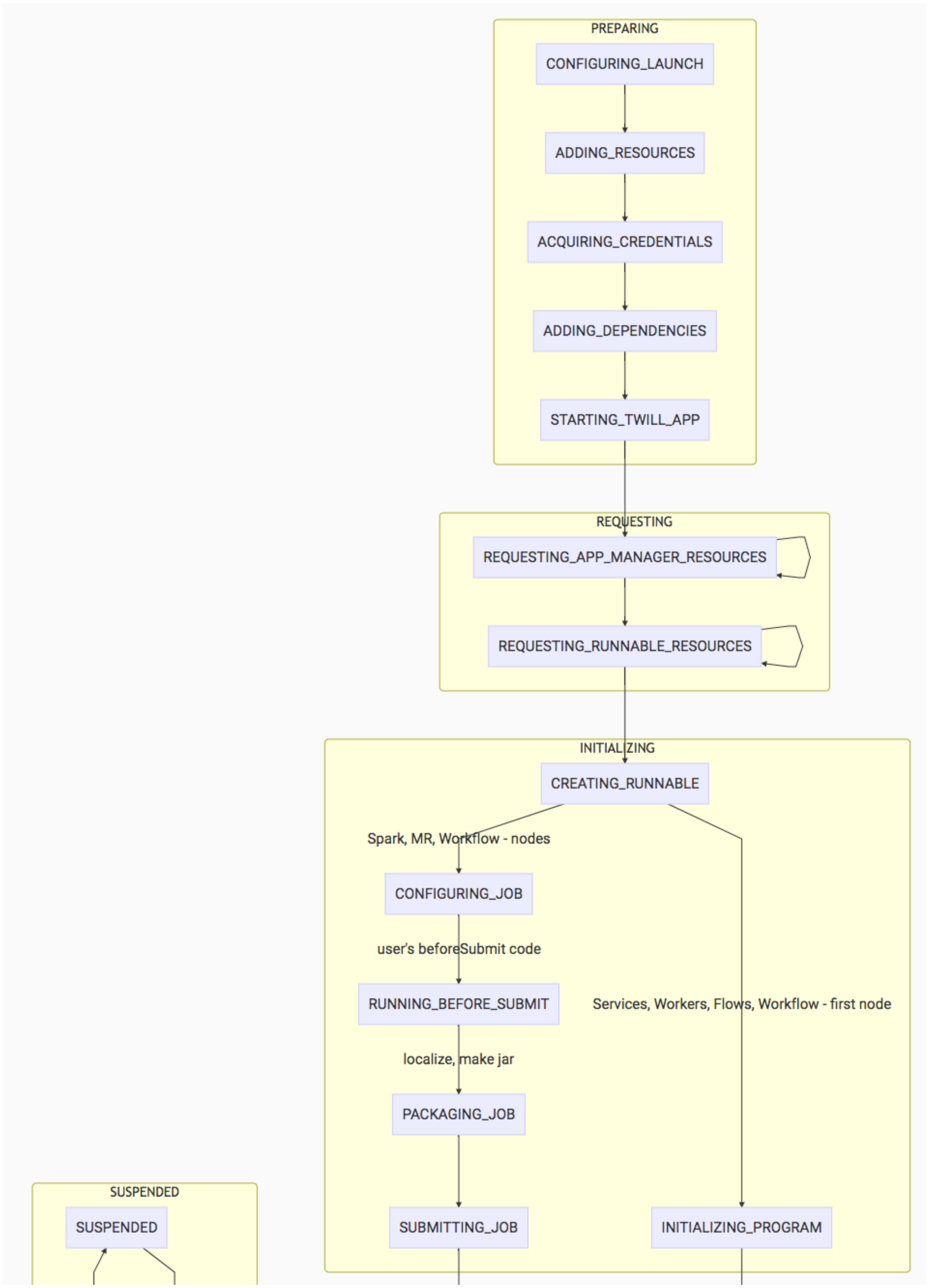
Definition

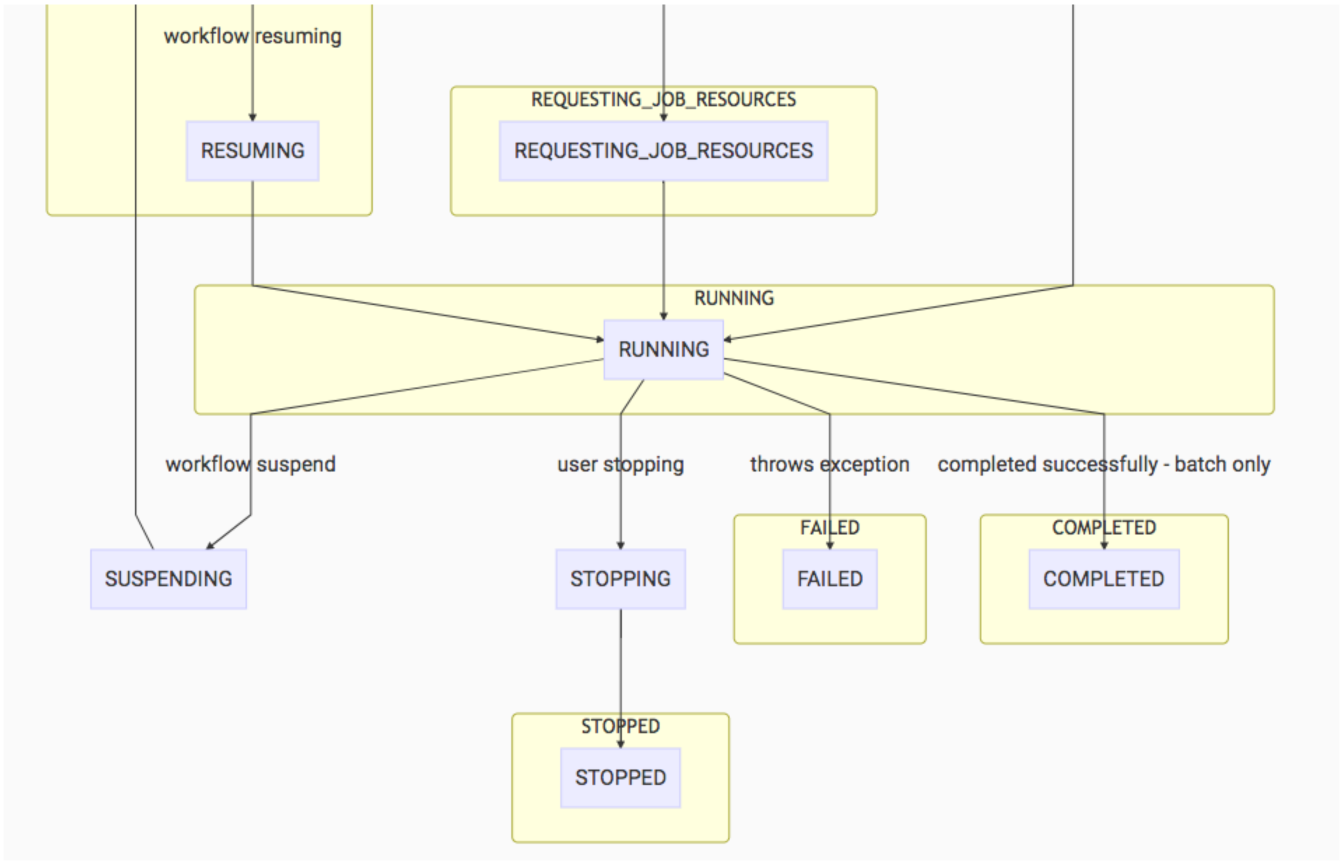
- Before we attempt to refactor how program states, we should understand how they are defined in CDAP today. These are the different ways that program states are defined today in CDAP:
    - ProgramController.State: This is the internal representation of a program's state, used only by program runners and program controllers. This is not exposed to the user.
    - ProgramStatus: This is the user-facing representation of a program's state.
    - ProgramRunStatus: This is used for querying program runs.

Design

- Below are two diagrams that propose more granular program states.

Program State Diagram


PREPARING
CONFIGURING_LAUNCH
ADDING_RESOURCES
ACQUIRING_CREDENTIALS
ADDING_DEPENDENCIES
STARTING_TWILL_APP
REQUESTING
REQUESTING_APP_MANAGER_RESOURCES
REQUESTING_RUNNABLE_RESOURCES
INITIALIZING
CREATING_RUNNABLE
Spark, MR, Workflow - nodes
CONFIGURING_JOB
user's beforeSubmit code
RUNNING_BEFORE_SUBMIT
Services, Workers, Flows, Workflow - first node
localize, make jar
PACKAGING_JOB
SUBMITTING_JOB
INITIALIZING_PROGRAM
SUSPENDED
SUSPENDED



Legend:

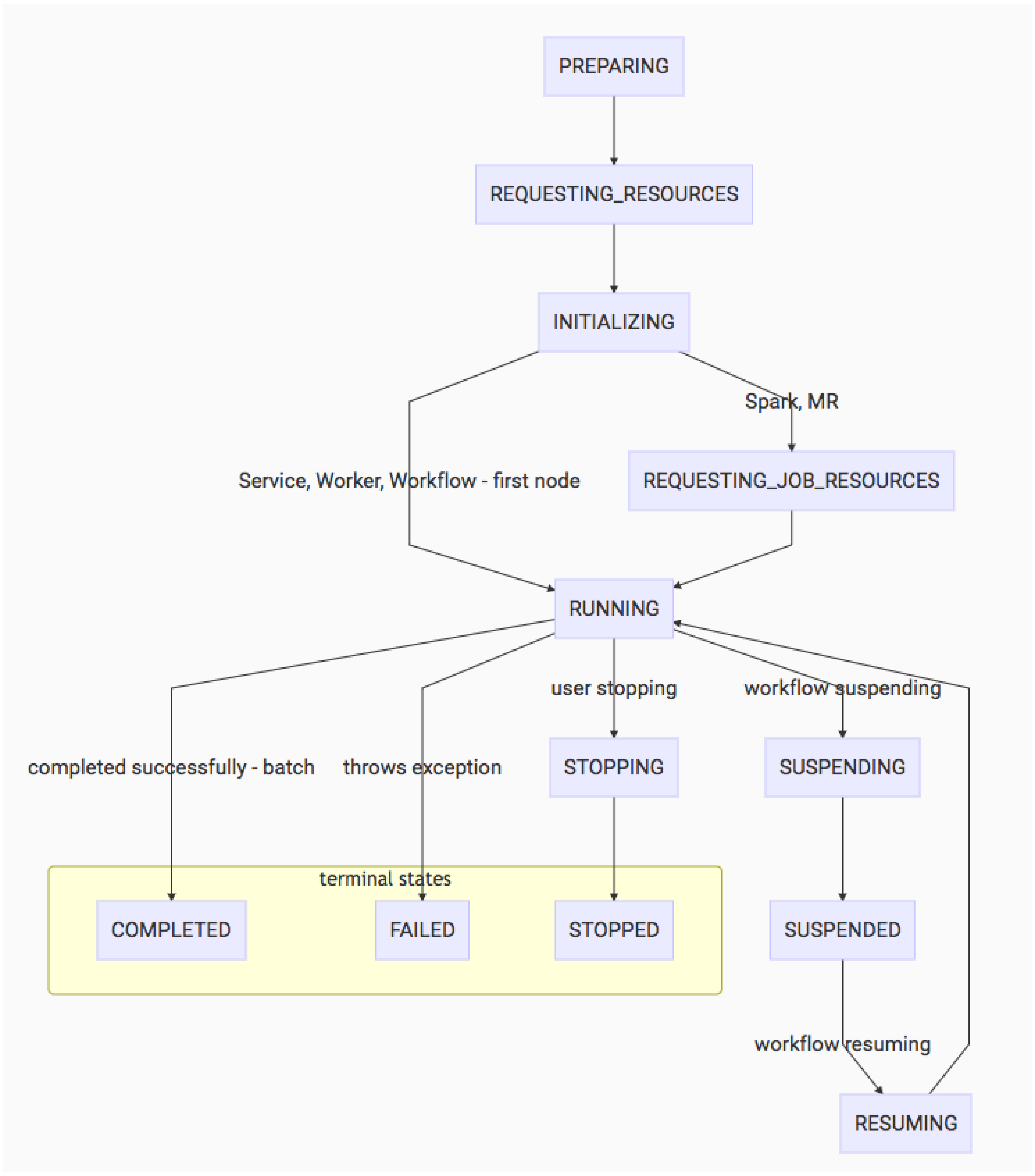
- The yellow boxes represent the states that would be exposed to the user if the program status was requested at any given point.
- The blue boxes represent the internal states of the program.

*Workflows have a different behavior that is not depicted fully in the diagram. A workflow is defined to be running when executing the first node. Since the first node of a workflow can be a Spark job, a MR job, or a custom action, internally, that node itself is executing a program, so internally, that node's state will be looping back to the CONFIGURING_JOB state for a Spark or MR job. However, the workflow *itself* woul d remain in a running state. This is for two reasons:

1. It is difficult to propagate the internal program's states back to the workflow.
    1. There can be parallel executions of nodes.
2. There isn't a use case currently for knowing the substate (node's state) of a workflow.

If a random node fails during a workflow's run, the failure will be propagated to the workflow state.

User-Facing Program States Diagram



Note that a workflow will go directly from INITIALIZING to RUNNING since the workflow itself will not be requesting resources for a job, but its nodes will. The nodes themselves may be Spark or MR jobs, which individually request job resources to begin their jobs. However, the workflow stays in a RUNNING status. See the note above this diagram for more information.

These are the states that should be persisted to the store and notifications should be sent through TMS for every one of these transitions.

Failure States

- The above two flow charts represents the programs state transitions. Failure states that may occur along every step are not explicitly shown. For instance, if we fail to request containers, then the program status will transition to KILLED.

Approach

- Whenever a program state has been updated, we should determine if the state transition is valid and update the state.
- A message should be sent through a TMS topic (program.status.event.topic) for every user-facing program status transition (the second diagram).
- We no longer make HTTP requests to the store with an updated status of a program run. Instead, we can subscribe to the TMS topic and persist the appropriate state changes into the store when necessary.

Publishing Program Status Changes

- Before defining more granular program status, we need to first refactor the store by publishing existing program statuses to TMS so that the store can subscribe to the TMS topics.
- We will move all store persistence (Flows, Services, Workers) to the program type's respective containers to unify where the state changes occur for each program. (**CDAP-2013** - Getting issue details... STATUS)
- Then, we will replace the store persistence calls with program status notification calls that will validate the program state change before publishing the notification.
    - The RemoteRuntimeStore can be replaced with a store that subscribes to the program status notifications.
- If the container dies, or if there is any error with publishing program statuses, then the record fixer needs to handle the program state inconsistency and publish a notification to correct the incomplete program state.

What does it mean for a program to be running?

- For every program type, the program should be marked running when it is actually running:
    - A `Service` is marked running after the HTTP server is up and ready to receive requests.
    - A `Spark` Job is running after it has been submitted by the SparkSubmitter (and in AbstractTwillProgramController, we get an event from the `onRunning` listener)
    - A `MapReduce` job is running after the job has been submitted (and in AbstractTwillProgramController, we get an event from the `onRunning` listener)
    - A `Workflow` is running when the first node is running (actions, MapReduce job, Spark job).
    - A `Flow` is running when it is ready to process events.
    - A `Worker` is running when `worker.run()` has been called.
- What about when 80% of workers are running and 20% are not running? What will the state be classified as then?
    - In this case, not all workers are running, but since there are workers that *are* running, the Worker should be marked as running

Terminal States

- We wish to have 3 terminal states:
    - COMPLETED: When a batch program successfully completes
        - Ex: any successful batch program completes.
    - FAILED: When a program terminates with an error.
        - Ex: A MapReduce job fails with an error, or a container dies.
        - This can occur for both batch and realtime programs
    - STOPPED: When a user specifically tries to stop a program (both realtime and batch can be stopped). Currently, we map a stopped process to killed. This would no longer happen under this new proposal. If there was an error with trying to stop a program, then this can transition to FAILED accordingly.
        - A program that is stopped can be started again (although under a different run), so it doesn't make sense for a stopped program to be marked killed as they represent different things.
        - Ex: When a user stops a flow or a worker.
- What happens if Yarn / Twill returns killed? Do we map it to stopped now?
    - Yes.

Program Statuses (External)

- We wish to add a few more user-facing program statuses so that this can be exposed to the user and make it clearer what state a program is involved in, at a higher level:
    - `PREPARING`: Represents building and starting an application environment for a program
    - `REQUESTING_RESOURCES`: Represents whenever the program is requesting resources to start application master and the basic containers to run a Spark job or a MR job. Useful since requesting containers can take a long time.

- `REQUESTING_JOB_RESOURCES`: Represents whenever the job is requesting containers to start its packaged program. This is different from `REQUESTING_RESOURCES` because this state is active after the application master has already started. This state would only be reached by Spark and MR jobs.
- `INITIALIZING`: Represents building of program jar and localizing program resources and configuration before running a program

API changes

New Programmatic APIs

An internal State enum must be composed:

State

```
enum State {
  CONFIGURING_LAUNCH(ProgramStatus.PREPARING, true),
  ADDING_RESOURCES(ProgramStatus.PREPARING, false),
  ACQUIRING_CREDENTIALS(ProgramStatus.PREPARING, false),
  STARTING_TWILL_APP(ProgramStatus.PREPARING, false),

  REQUESTING_APP_MANAGER_RESOURCES(ProgramStatus.REQUESTING_RESOURCES,
true),
  REQUESTING_RUNNABLE_RESOURCES(ProgramStatus.REQUESTING_RESOURCES,
false),

  CREATING_RUNNABLE(ProgramStatus.INITIALIZING, true),
  INITIALIZING_PROGRAM(ProgramStatus.INITIALIZING, false),
  CONFIGURING_JOB(ProgramStatus.INITIALIZING, false),
  PREPARING_SUBMIT(ProgramStatus.INITIALIZING, false),
  SUBMITTING_JOB(ProgramStatus.INITIALIZING, false),

  REQUESTING_JOB_RESOURCES(ProgramStatus.REQUESTING_JOB_RESOURCES,
true),

  RUNNING(ProgramStatus.RUNNING, false),
  STOPPING(ProgramStatus.RUNNING, false),
  SUSPENDING(ProgramStatus.RUNNING, false),

  SUSPENDED(ProgramStatus.SUSPENDED, true),
  RESUMING(ProgramStatus.SUSPENDED, false),

  COMPLETED(ProgramStatus.COMPLETED, true),
  FAILED(ProgramStatus.FAILED, true),
  STOPPED(ProgramStatus.STOPPED, true);

  private final ProgramStatus programStatus;
  private final boolean shouldSendNotification;

  State(ProgramStatus programStatus, boolean shouldSendNotification) {
    this.programStatus = programStatus;
    this.shouldSendNotification = shouldSendNotification;
  }

  public ProgramStatus getRunStatus() {
    return programStatus;
  }

  public boolean shouldSendNotification() {
    return shouldSendNotification;
  }

  public boolean isDone() {
    return this == COMPLETED || this == FAILED || this == STOPPED;
  }
}
```

The user-facing ProgramStatus enum will look like:

ProgramStatus

```
public enum ProgramStatus {
  PREPARING,
  REQUESTING_RESOURCES,
  INITIALIZING,
  REQUESTING_JOB_RESOURCES,
  RUNNING,
  SUSPENDED,
  COMPLETED,
  FAILED,
  STOPPED
}
```

Deprecated Programming APIs

- We wish to deprecate ProgramRunStatus in favor of using ProgramStatus. To maintain the existing functionality of querying program runs regardless of the status (i.e using ProgramRunStatus.ALL), users no longer specify any status, and all statuses will be assumed instead.

New Status Endpoint

Path	Method	Description	Request Body	Response Code	Response

`/apps/{app-id}/versions/{version-id}/{program-type}/{program-id}/status`	GET	To get the status of a Program	None	200 - On success
				
				404 - When application is not available
				
				500 - Any internal server errors
					Response
					`{`
					
					`"status":`

					`"<One of ProgramStatus>"`
					`}`

Deprecated REST API

- We still need to add backwards compatibility for the API, to support how new external program statuses will be defined now as well as the current definitions.

Resolved Questions

1. Can we distinguish when a Twill program is initializing and when a Twill program is actually running? Doesn't seem to be the case, since Yarn controller and Yarn Application State doesn't have an `initializing` state.
    1. `twillController` only subscribes to `onRunning` and `onTerminated`
    2. Answer: No, it would be nice if we could.
2. When a workflow is suspended, we seem to mark it as STOPPED, but shouldn't it be marked instead as SUSPENDED? ProgramStatus currently doesn't have SUSPENDED as a status (but it should).
    1. Answer: We should mark workflows as suspended when they are suspended.
3. How long is the abort time when requesting containers?
    1. Answer: It is configurable.
4. Does it seem redundant now to have ProgramStatus and ProgramRunStatus? I don't believe so, as the ProgramStatus enum will be used for other things, like program status based scheduling. It wouldn't be appropriate to use ProgramRunStatus as that should be associated with querying records only.
    1. Answer: Yes it is. Let's look to remove ProgramRunStatus.

Open Questions

1. The internal program state enum needs to be in a different place other than ProgramController, as the lifetime of the ProgramController no longer matches the lifetime of the states. Ideas on where to define this?
2. Should we deprecate proto's ProgramStatus, an enum that just shows running or stopped? It is used as part of the API we are deprecating.

Security Impact

- `ProgramController.State` must be internal, and its internal state will not be exposed. Currently, it doesn't seem to be exposed either, so there is no change here.
- Instead, `ProgramRunStatus` can represent the status of a specific program run for the user. This will also continue to be exposed to the user, but the difference is that the user-exposed states are a bit more granular.

Impact on Infrastructure Outages

- If TMS is down, sending program states would also go down. No additional outages should occur.

Test Scenarios

Test ID	Test Description	Expected Results

Releases

Release X.Y.Z

Release X.Y.Z

Related Work

- Work #1
- Work #2
- Work #3

Future work