

0.11.3

Table of contents

Mission Modeler Guide

- Foundations of Simulation & Modeling
 - Simulation Mechanics
- Creating a Mission Simulation Overview
- Developing a Mission Model
- Configuring a Mission Model
- Activities
- Activity Mappers
- Value Schemas
- Declaring Parameters
- Models & Resources
- Creating Plans
- Constraints
- Precomputed Profiles
- Automated Scheduling
- Glossary

Command Expansion Guide

- Upload and Generate Command Typescript Library
- Generate Activity Typescript Library
- Setup IDE and Writing Expansion Logic
- Submit Expansion Logic
- Create an Expansion Set
- Expand the Plan
- Get Expanded Commands
- Create Sequence
- Link Simulated Activity to Sequence
- Retrieve SeqJson Serialization of Sequence

User Guide

- Merlin Activity Plans
- Aerie GraphQL API
- Planning UI
- Simulation Configuration
- UI Views

Product Guide

- Installation
- System Requirements
- Administration
- Support

Mission Modeler Guide

Introduction

This guide explains how to make use of Aerie-provided capabilities in the latest version of Aerie, and will be updated as Aerie evolves.

Aerie is a new software system being developed by the MPSA element of MGSS (Multi-mission Ground System and Services), a subsystem of AMMOS (Advanced Multi-mission Operations System). Aerie will support mission operations by providing capabilities for activity planning, sequencing, and spacecraft analysis. These capabilities include modeling & simulation, scheduling, and validation. Aerie will replace several legacy MGSS tools, including but not limited to APGEN, SEQGEN, MPS Editor and MPS Server.

Aerie currently provides the following elements.

- Merlin modeling framework for defining mission resources and activity types.
- Merlin web UI for activity planning and simulation analysis.
- Falcon sequence editor UI

Mission Modeling with the Merlin Framework

In Merlin, a mission model serves activity planning needs in two ways. First, it describes how various mission resources behave autonomously over time. Second, it defines how activities perturb these resources at discrete time points, causing them to change their behavior. This information enables Aerie to provide scheduling, constraint validation, and resource plotting capabilities on top of a mission model.

The Merlin Framework empowers adaptation engineers to serve the needs of mission planners maintain as well as keep their codebase maintainable and testable over the span of a mission. The Framework aims to make the experience of mission modeling similar to standard Java development, while still addressing the unique needs of the simulation domain.

In the Merlin Framework, a mission model breaks down into two types of entity: system models and activity types.

System models can range in complexity from a single aspect of an instrument to an entire mission. In fact, Merlin only requires one system model to exist: the top-level mission model. The mission model can delegate to other, more focused models, such as subsystem models, which may themselves delegate further. Ultimately, fine-grained models capture the system state they own in a `Cell`, which is a simulation-aware analogue of a Java mutable field. (**In Merlin, mutable fields on models must not be used.** All mutable state must be controlled by a `Cell`.) Models may provide regular Java methods for interacting with that state, and other models (including activity types) may invoke those methods.

Activity types are a specialized kind of model. Each activity type defines the parameters for activities of that type, which may be instantiated and configured by a mission planner. An activity type also defines a single method that acts as the entrypoint into the simulated system: when an activity of that type occurs, its method is invoked with the activity parameters and the top-level mission model. It may then interact freely with the rest of the system.

Just as activity types define the entrypoints into a simulation, the mission model also defines *resources*, which allow information to be extracted from the simulation. A resource is associated with a method that returns a "dynamics" -- a description of the current autonomous behavior of the resource. Merlin currently provides discrete dynamics (constants held over time) and linear dynamics (real values varying linearly with time), and is designed to support more in the future.

A simulation over a mission model iteratively runs activities and queries resources for their updated dynamics, producing a composite profile of dynamics for each resource over the entire simulation duration.

Simulation Mechanics

In progress...

Creating a Mission Simulation Overview

Running an Aerie simulation requires a mission model that describes effects of activities over modeled resources, and a plan file that declares a schedule of activity instances with specified parameters. A plan file must be created with respect to an mission model file, since activities in a plan and their parameters are validated against activity type definitions in the mission model.

Here's a summary workflow to getting a simulation result from Aerie.

1. Install Aerie services following instructions on [Product Guide](#)
2. Create a Merlin mission model following the instructions on [Developing a Mission Model](#) page.
3. Upload the mission model to Aerie through [Planning Web GUI](#) or [Aerie API GraphQL graphical interface](#) .
4. Create a plan associated with the mission model using again the [Planning Web GUI](#) or [Aerie API](#)
5. Trigger simulation via either interfaces listed above.

Developing a Mission Model

A mission model defines the behavior of any measurable mission resources, and a set of **activity types**, defining the ways in which a plan may influence mission resources.

Mission Modeling Libraries

Aerie provides seven libraries that can be imported by a project. They are:

- [contrib](#)
- [merlin-framework](#)
- [merlin-framework-processor](#)
- [merlin-framework-junit](#)
- [merlin-sdk](#)
- [merlin-driver](#)
- [parsing-utilities](#)

Package-info File

A mission model must contain, at the very least, a `package-info.java` containing annotations that describe the highest-level features of the mission model. For example:

```
// examples/banananation/package-info.java
@MissionModel(model = Mission.class)
@WithActivityType(BiteBananaActivity.class)
@WithActivityType(PeelBananaActivity.class)
@WithActivityType(ParameterTestActivity.class)
@WithMappers(BasicValueMappers.class)
package gov.nasa.jpl.aerie.banananation;

import gov.nasa.jpl.aerie.banananation.activities.BiteBananaActivity;
import gov.nasa.jpl.aerie.banananation.activities.ParameterTestActivity;
import gov.nasa.jpl.aerie.banananation.activities.PeelBananaActivity;
import gov.nasa.jpl.aerie.contrib.serialization.rulesets.BasicValueMappers;
import gov.nasa.jpl.aerie.merlin.framework.annotations.MissionModel;
import gov.nasa.jpl.aerie.merlin.framework.annotations.MissionModel.WithActivityType;
import gov.nasa.jpl.aerie.merlin.framework.annotations.MissionModel.WithMappers;
```

This `package-info.java` identifies the top-level class representing the mission model, and registers activity types that may interact with the mission model. Merlin processes these annotations at compile-time, generating a set of boilerplate classes which take care of interacting with the Aerie platform.

The `@WithMappers` annotation informs the annotation processor of a set of serialization rules for activity parameters of various types; the `BasicValueMappers` ruleset covers most primitive Java types. Mission modelers may also create their own rulesets, specifying rules for mapping custom value types. If multiple mapper classes are included via the `@WithMappers` annotations, and multiple mappers declare a mapping rule to the same data type, the rule found in the earlier declared mapper will take precedence. For more information on allowing custom values, see [value mappers](#).

Mission Model Class

The top-level mission model is responsible for defining all of the mission resources and their behavior when affected by activities. Of course, the top-level model may delegate to smaller, more focused models based on the needs of the mission. The top-level model is received by activities, however, so it must make accessible any resources or methods to be used therein.

```
// examples/banananation/Mission.java
public class Mission {
    public final AdditiveRegister fruit = AdditiveRegister.create(4.0);
    public final AdditiveRegister peel = AdditiveRegister.create(4.0);
    public final Register<Flag> flag = Register.create(Flag.A);

    public Mission(final Registrar registrar) {
        registrar.discrete("/flag", this.flag, new EnumValueMapper<>(Flag.class));
        registrar.real("/peel", this.peel);
        registrar.real("/fruit", this.fruit);
    }
}
```

Mission resources are declared using `Registrar#discrete` or `Registrar#real`.

A model may also express autonomous behaviors, where a discrete change occurs in the system outside of an activity's effects. A **daemon task** can be used to model these behaviors. Daemons are spawned at the beginning of any simulation, and may perform the same effects as an activity. Daemons are prepared using the `spawn` method.

Activity types

An **activity type** defines a simulated behavior that may be invoked by a planner, separate from the autonomous behavior of the mission model itself. Activity types may define **parameters**, which are filled with **arguments** by a planner and provided to the activity upon execution. Activity types may also define **validations** for the purpose of informing a planner when the parameters they have provided may be problematic.

```
// examples/banananation/activities/PeelBananaActivity.java
@ActivityType("PeelBanana")
public final class PeelBananaActivity {
    private static final double MASHED_BANANA_AMOUNT = 1.0;

    @Parameter
    public String peelDirection = "fromStem";

    @Validation("peel direction must be fromStem or fromTip")
    public boolean validatePeelDirection() {
        return List.of("fromStem", "fromTip").contains(this.peelDirection);
    }

    @EffectModel
    public void run(final Mission mission) {
        if (peelDirection.equals("fromStem")) {
            mission.fruit.subtract(MASHED_BANANA_AMOUNT);
        }
        mission.peel.subtract(1.0);
    }
}
```

Merlin automatically generates parameter serialization boilerplate for every activity type defined in the mission model's `package-info.java`. Moreover, the generated `Model` base class provides helper methods for spawning each type of activity as children from other activities.

Uploading a Mission Model

In order to use a mission model to simulate a plan on the Aerie platform, it must be packaged as a JAR file with all of its non-Merlin dependencies bundled in. The [template mission model](#) provides this capability out of the box, so long as your dependencies are specified with Gradle's `implementation` dependency class. The built mission model JAR can be uploaded to Aerie through the Aerie web UI.

Configuring a Mission Model

A **mission model configuration** enables mission modelers to set initial mission model values when running a simulation. Configurations are tied to a plan, therefore each plan is able to define its own set of configuration parameters. The `examples/banananation` project contains a `Configuration` data class example to demonstrate how a simple configuration may be created.

Setup

Mission Model

The `examples/banananation` project makes use of the `@WithConfiguration` annotation within `package-info.java`:

```
@MissionModel(model = Mission.class)

@WithConfiguration(Configuration.class)
```

When the `@WithConfiguration` annotation is used, the model – defined within the `@MissionModel` annotation – must accept the configuration as a constructor argument. See `Mission.java`:

```
public Mission(final Registrar registrar, final Configuration config) {
    // ...
}
```

Configuration

A configuration class should be defined with the same parameter annotations as activities. See [Declaring Parameters](#) for a thorough explanation of all possible styles of `@Export` parameter declaration and validation.

Similarly to activities, the Merlin annotation processor will take care of all serialization/deserialization of the configuration object. The Merlin annotation processor will generate a configuration mapper for the configuration defined within `@WithConfiguration()`.

Examples

`Configuration` can be a simple data class. For example:

```
package gov.nasa.jpl.aerie.banananation;

import java.nio.file.Path;

import static gov.nasa.jpl.aerie.merlin.framework.annotations.Export.Template;

public record Configuration(int initialPlantCount, String initialProducer, Path initialDataPath) {

    public static final int DEFAULT_PLANT_COUNT = 200;
    public static final String DEFAULT_PRODUCER = "Chiquita";
    public static final Path DEFAULT_DATA_PATH = Path.of("/etc/os-release");

    public static @Template Configuration defaultConfiguration() {
        return new Configuration(DEFAULT_PLANT_COUNT, DEFAULT_PRODUCER, DEFAULT_DATA_PATH);
    }
}
```

See `examples/` for a demonstration of each possible style of configuration definitions:

- `examples/foo-missionmodel`: uses standard `@Parameter` configuration annotations.
- `examples/banananation`: (shown above) uses the `@Template` annotation to define a default `Configuration` object.
- `examples/config-with-defaults`: uses `@WithDefaults` to define a default for each parameter.
- `examples/config-without-defaults`: defined with no default arguments, requires all arguments to be supplied by the planner.

Use

The mission model may use a configuration to set initial values, for example:

```
this.sink = new Accumulator(0.0, config.sinkRate);
```


Activities

- [Activity Annotation](#)
- [Activity Metadata](#)
- [Activity Parameters](#)
- [Validations](#)
- [Activity Effect Model](#)
 - [A Note about Decomposition](#)
- [Computed Attributes](#)
- [Duration Types](#)

An activity is a modeled unit of mission system behavior to be utilized for simulations during planning. Activities emit events which can have various effects on modeled resources, and these effects can be modulated through input parameters. Activities can represent any part of the mission system behavior such as the availability of ground assets, or the execution of commands by the flight software or the instruments. In other words, activities in Merlin are entities whose role is to emit stimuli (events) to which the mission model reacts. Activities can therefore describe the relation: "when this activity occurs, this kind of thing should happen".

An activity type is a prototype for activity instances to be executed in a simulation. Activity types are defined by java classes that provide an `EffectModel` method to Merlin, along with a set of parameters. Each activity type exists in its own .java file, though activity types can be organized into hierarchical packages, for example as `gov.nasa.jpl.europa.clipper.gnc.TCMActivity`

Activity types consist of:

- metadata
- parameters that describe the range in execution and effects of the activity
- effect model that describes how the system will be perturbed when the activity is executed.

Activity Annotation

In order for Merlin to detect an activity type, its class must be annotated with the `@ActivityType` tag. An activity type is declared with its name using the following annotation:

```
@ActivityType("TurnInstrumentOff")
```

By doing so, the Merlin annotation processor can discover all activity types declared in the mission model, and validate that activity type names are unique.

Activity Metadata

Metadata of activities are structured such that the Merlin annotation processor can extract this metadata given particular keywords. Currently, the Merlin annotation processor recognizes the following tags: `contact`, `subsystem`, `brief_description`, and `verbose_description`.

These metadata tags are placed in a JavaDocs style comment block above the Activity Type to which they refer. For example:

```
/**
 * @subsystem Data
 * @contact mkumar
 * @brief_description A data management activity that deletes old files
 */
```

These tags are processed, at compile time, by the annotation processor to create documentation for the Activity types that are described in the mission model.

Activity Parameters

Activity parameters provide the ability to modulate the behavior of an activity instance's effect model. Aside from determining the effects of the

activity, these parameters can be used to determine its duration, decomposition into children activities and expansion into commands.

The Merlin annotation processor is used to extract and generate serialization code for parameters of activity types. The annotation processor also allows authors of a mission model to create mission-specific parameter types, ensuring that they will be recognized by the Merlin framework. See [Declaring Parameters](#) for a thorough explanation of all possible styles of parameter declarations. For more information on mission-specific parameter types, see [Value Mappers](#).

Validations

See [Declaring Parameters](#) for a thorough explanation of parameter validation.

Activity Effect Model

Every activity type has an associated "effect model" that describes how that activity impacts mission resources. An effect model is a method on the activity type class annotated with `@EffectModel`; there can only be one such method, and by convention it is named `run` that accepts the top level `Mission` model as a parameter. This method is invoked when the activity begins, paused when the activity waits a period of time, and resumes when that period of time passes. The activity's computed duration will be measured from the instant this method is entered, and extends either until the method returns or until all spawned children of the activity have completed -- whichever is longer.

Exceptions: If an uncaught exception is thrown from an activity's effect model, the simulation will halt. No later activities will be performed, and simulation time will not proceed beyond the instant at which the exception occurred. As such, uncaught exceptions are essentially treated as fatal errors. We advise mission models to employ uncaught exceptions sparingly, as it deprives planners of information about the behavior of the spacecraft after the fault. (At least as of 2021-06-11, an uncaught exception will cause the simulation to abort without producing *any* results -- not even up to the point of failure.) On the other hand, uncaught exceptions may be useful to identify bugs in the mission model or situations that the mission model is not intended to simulate.

Actions: An activity's effect model may wait for time to pass, spawn other activities, and affect spacecraft state. Spacecraft state is affected via the `Mission` model parameter, which depends on the details of the modeled mission systems. (See [Developing a Mission Model](#) for more on mission modeling.)

Actions related to the passage of simulation time are provided as static methods on the `merlin.framework.ModelActions` class:

- `delay(duration)` : Delay the currently-running activity for the given duration. On resumption, it will observe effects caused by other activities over the intervening timespan.
- `waitFor(activityId)` : Delay the currently-running activity until the activity with specified ID has completed. On resumption, it will observe effects caused by other activities over the intervening timespan.
- `waitUntil(condition)` : Delay the currently-running activity until the provided `Condition` becomes true. On resumption, it will observe effects caused by other activities over the intervening timespan.

Actions related to spawning other activities are provided by the generated `ActivityActions` class, usually found under the `generated` package within your codebase.

- `spawn(activity)` : Spawn a new activity as a child of the currently-running activity at the current point in time. The child will initially see any effects caused by its parent up to this point. The parent will continue execution uninterrupted, and will not initially see any effects caused by its child.
- `call(activity)` : Spawn a new activity as a child of the currently-running activity at the current point in time. The child will initially see any effects caused by its parent up to this point. The parent will halt execution until the child activity has completed. This is equivalent to calling `waitFor(spawn(activity))`.

For example, consider a simple activity for running the on-board heaters:

```

@ActivityType("RunHeater")
public final class RunHeater {
    private static final int energyConsumptionRate = 1000;

    @Parameter
    public long durationInSeconds;

    @Validation("duration must be positive")
    public boolean validateDuration() {
        return durationInSeconds > 0;
    }

    @EffectModel
    public void run(final Mission mission) {
        spawn(new PowerOnHeater());

        final double totalEnergyUsed = durationInSeconds * energyConsumptionRate;
        mission.batteryCapacity.use(totalEnergyUsed);

        delay(durationInSeconds, Duration.SECONDS);

        call(new PowerOffHeater());
    }
}

```

This activity first spawns a `PowerOnHeater` activity, which then continues concurrently with the current `RunHeater` activity. Next, the total energy to be used by the heater is subtracted from the remaining battery capacity (see [Models and Resources](#)). The energy used depends on the duration parameter of the activity, allowing the activity's effect to be tuned by the planner. Next, the activity waits for the desired heater runtime to elapse, then spawns and waits for a `PowerOffHeater` activity. The activity completes after both children have completed.

A Note about Decomposition

In Merlin mission models, decomposition of an activity is not an independent method, rather it is defined within the effect model by means of invoking child activities. These activities can be invoked using the `call()` method, where the rest of the effect model waits for the child activity to complete; or using the `spawn()` method, where the effect model continues to execute without waiting for the child activity to complete. This method allows any arbitrary serial and parallel arrangement of child activities. This approach replaces duration estimate based wait calls with event based waits. Hence, this allows for not keeping track of estimated durations of activities, while also improving the readability of the activity procedure as a linear sequence of events.

Computed Attributes

Upon the termination of an Activity, the effect model can optionally log some information that will be included with the simulation results. This information has no effect on the current simulation, but can be used by downstream tools that process the simulation results.

To specify the information that will be logged, the mission modeler must change the return type of the method marked with the `@EffectModel` annotation.

Let's consider the `RunHeater` example from above:

```

@ActivityType("RunHeater")
public final class RunHeater {
    private static final int energyConsumptionRate = 1000;

    @Parameter
    public long durationInSeconds;

    @Validation("duration must be positive")
    public boolean validateDuration() {
        return durationInSeconds > 0;
    }

    @EffectModel
    public void run(final Mission mission) {
        spawn(new PowerOnHeater());

        final double totalEnergyUsed = durationInSeconds * energyConsumptionRate;
        mission.batteryCapacity.use(totalEnergyUsed);

        delay(durationInSeconds, Duration.SECONDS);

        call(new PowerOffHeater());
    }
}

```

The effect model return type is "void", which means no useful information will be logged. Let's change that to `String` and add a log message:

```
@EffectModel
public String run(final Mission mission) {
    spawn(new PowerOnHeater());

    final double totalEnergyUsed = durationInSeconds * energyConsumptionRate;
    mission.batteryCapacity.use(totalEnergyUsed);

    delay(durationInSeconds, Duration.SECONDS);

    call(new PowerOffHeater());

    return "This is a log message!"
}
}
```

Computed attributes are not limited to strings - they can be any type that merlin knows how to serialize. See [ValueMappers](#) to learn about what types are supported out of the box, and how to define custom serializers for your own types.

Let's show how we can define computed attributes as a Map of Strings to Longs:

```
@EffectModel
public Map<String, Long> run(final Mission mission) {
    spawn(new PowerOnHeater());

    final double totalEnergyUsed = durationInSeconds * energyConsumptionRate;
    mission.batteryCapacity.use(totalEnergyUsed);

    delay(durationInSeconds, Duration.SECONDS);

    call(new PowerOffHeater());

    return Map.of("duration_in_seconds", durationInSeconds,
                  "energy_consumption_rate", (Long) energyConsumptionRate);
}
}
```

Duration Types

The scheduler (see [Scheduling Guide](#)) places activities in a plan to try achieving scheduling goals.

As the effect model of an activity is not accessible to the scheduler, its duration can only be computed by simulation. Satisfying temporal constraints associated with the scheduling of this activity ("activity A must end before activity B ") may lead to multiple simulations and thus more computation time.

However, it is possible to provide information about how the duration of an activity is determined to help the scheduler.

The duration of an activity is said to be:

- **controllable** if it is *only* determined by one of the activity parameter.
- **uncontrollable** otherwise.

By default, the duration of an activity is uncontrollable. To specify that the duration of an activity is controllable, the `@ControllableDuration` annotation can be added to the effect model method such as in the example below:

```

@ActivityType("RunHeater")
public final class RunHeater {
    private static final int energyConsumptionRate = 1000;

    @Parameter
    public long durationInSeconds;

    @Validation("duration must be positive")
    public boolean validateDuration() {
        return durationInSeconds > 0;
    }

    @EffectModel
    @ControllableDuration(parameterName = "durationInSeconds")
    public void run(final Mission mission) {
        spawn(new PowerOnHeater());

        final double totalEnergyUsed = durationInSeconds * energyConsumptionRate;
        mission.batteryCapacity.use(totalEnergyUsed);

        delay(durationInSeconds, Duration.SECONDS);
    }
}

```

Note that compared to the `RunHeater` activity present in the previous paragraphs, the `call(new PowerOffHeater())` at the end of the effect model has been removed because (1) it would have made the duration of the activity depend on the duration of the `PowerOffHeater` activity, (2) we assume `PowerOffHeater` has an uncontrollable duration. Spawning another activity does not affect the duration of this activity as they run in parallel.

However, let's say `PowerOffHeater` also has a controllable duration that is passed through a parameter, we could write the following activity:

```

@ActivityType("RunHeater")
public final class RunHeater {
    private static final int energyConsumptionRate = 1000;

    @Parameter
    public long durationInSeconds;

    @Validation("duration must be positive")
    public boolean validateDuration() {
        return durationInSeconds > 0;
    }

    @EffectModel
    @ControllableDuration(parameterName = "durationInSeconds")
    public void run(final Mission mission) {
        spawn(new PowerOnHeater());

        final double totalEnergyUsed = durationInSeconds * energyConsumptionRate;
        mission.batteryCapacity.use(totalEnergyUsed);

        durationPowerOff = 100L;
        durationPowerOn = durationInSeconds - durationPowerOff;

        delay(durationPowerOn, Duration.SECONDS);
        call(new PowerOffHeater(durationPowerOff))
    }
}

```

Even though the duration of `RunHeater` depends on the (fixed) duration of `PowerOffHeater`, we know its duration will be equal to `durationInSeconds`, making the activity effectively controllable.

The annotation `@ControllableDuration(parameterName = "durationInSeconds")` has no other effect than to tell the scheduler that the duration of this activity can be controlled via the `durationInSeconds` parameter. It acts like a contract between the mission model and the scheduler, ensuring that the duration of the activity will be equal to the duration specified by `durationInSeconds`.

After being given this information, the scheduler considers it can control the duration of the activity which will thus require less simulations, significantly improving scheduling performance for this activity type.

Activity Mappers

What is an Activity Mapper

An Activity Mapper is a Java class that implements the `ActivityMapper` interface for the `ActivityType` being mapped. It is required that each Activity Type in an mission model have an associated Activity Mapper, to provide several capabilities surrounding serialization/deserialization of activity instances.

The Merlin annotation processor can automatically [generate activity mappers](#) for every activity type, even for those with custom-typed parameters, but if it is desirable to create a custom activity mapper the interface is described below.

ActivityMapper Interface

The `ActivityMapper` interface is shown below:

```
public interface ActivityMapper<Instance> {
    String getName();
    Map<String, ValueSchema> getParameters();
    Map<String, SerializedValue> getArguments(Instance activity);

    Instance instantiateDefault();
    Instance instantiate(Map<String, SerializedValue> arguments) throws TaskSpecType.UnconstructableTaskSpecException;

    List<String> getValidationFailures(Instance activity);
}
```

The first thing to notice is that the interface takes a type parameter (here called `Instance`). When implementing the `ActivityMapper` interface, an activity mapper must supply the `ActivityType` being mapped. With that in mind, each of the methods shown must be implemented as such:

- `getName()` returns the name of the activity type being mapped
- `getParameters()` provides the named parameter fields of the activity along with their corresponding `ValueSchema`, that describes their structure
- `getArguments(Instance activity)` provides the actual values for each parameter from a provided activity instance
- `instantiateDefault()` creates a default instance of the activity type without any values provided externally
- `instantiate(Map<String, SerializedValue> arguments)` constructs an instance of the activity type from a the provided arguments, if possible
- `getValidationFailures(Instance activity)` provides a list of reasons a constructed activity is invalid, if any. Note that validation failures are different from instantiation errors. Validation failures occur when a constructed activity instance's parameters are outside acceptable range.

The `getParameters()` method returns a `Map<String, ValueSchema>`. In this map should be a key for every parameter, with a `ValueSchema` describing the structure of that parameter. See our [Value Schema documentation](#) for more information on creating value schemas.

Generated Activity Mappers

In most cases, you will likely want to let Merlin generate activity mappers for you. Thankfully, this is done automatically when running the Merlin Annotation Processor. When compiling your code with the Merlin annotation processor, the processor will produce an activity mapper for each activity type. This is made possible by the use of the `@WithMappers()` annotations in your `package-info.java`. Each java-file specified by these annotations is parsed to determine what types of values can be mapped. As long as there is a mapper for each activity parameter type used in the model, the annotation processor should have no issues creating activity mappers.

Value Mappers

Regardless of whether you create custom activity mappers or let Merlin generate them for you, you will likely find the need to work with a `ValueMapper` at some point. In fact, generating activity mappers is made quite simple by considering the fact that an activity instance is wholly defined by its parameter values.

You may find yourself asking "Just what `_is_` a value mapper?" A value mapper is a small, focused class whose sole responsibility is to tell Merlin how to handle a specific type of value. Value mappers allow all sorts of capabilities from custom-typed activity parameters to custom-typed resources.

One of the most convenient things about using value mappers is the fact that Merlin comes with them already defined for all basic types. Furthermore, value mappers for combinations of types can easily be created by passing one `ValueMapper` into another during instantiation.

Although we provide value mappers for basic types, it is entirely acceptable to create custom value mappers for other types, such as those imported from external libraries. This can be done by writing a Java class which implements the `ValueMapper` interface. Below is a value mapper for an apache `Vector3D` type as an example:

```
public class Vector3DValueMapper implements ValueMapper<Vector3D> {

    @Override
    public ValueSchema getValueSchema() {
        return ValueSchema.ofSequence(ValueSchema.REAL);
    }

    @Override
    public Result<Vector3D, String> deserializeValue(final SerializedValue serializedValue) {
        return serializedValue
            .asList()
            .map(Result::<List<SerializedValue>, String>success)
            .orElseGet(() -> Result.failure("Expected list, got " + serializedValue.toString()))
            .match(
                serializedElements -> {
                    if (serializedElements.size() != 3) return Result.failure("Expected 3 components, got " + serializedElements.size());
                    final var components = new double[3];
                    final var mapper = new DoubleValueMapper();
                    for (int i=0; i<3; i++) {
                        final var result = mapper.deserializeValue(serializedElements.get(i));
                        if (result.getKind() == Result.Kind.Failure) return result.mapSuccess(_left -> null);

                        // SAFETY: `result` must be a Success variant.
                        components[i] = result.getSuccessOrThrow();
                    }
                    return Result.success(new Vector3D(components));
                },
                Result::failure
            );
    }

    @Override
    public SerializedValue serializeValue(final Vector3D value) {
        return SerializedValue.of(
            List.of(
                SerializedValue.of(value.getX()),
                SerializedValue.of(value.getY()),
                SerializedValue.of(value.getZ())
            )
        );
    }
}
```

Notice there are just 3 methods to implement for a `ValueMapper`. The first is `getValueSchema()`, which should return a `ValueSchema` describing the structure of the value being mapped (see [here](#) for more info)

The next two methods are inverses of each other: `deserializeValue()` and `serializeValue()`. It is the job of `deserializeValue()` to take a `SerializedValue` and map it, if possible, into the mapper's supported value. Meanwhile, `serializeValue()` takes an instance of the mapper's supported value and turns it into a `SerializedValue`.

There are plenty of examples of value mappers over in the [contrib module](#).

Registering Value Mappers

As mentioned above, the `@WithMappers()` annotation is used to register value mappers for a mission model. Value mappers are expected to be defined with static constructor methods within classes listed in `@WithMappers()` annotations. For example, if `package-info.java` contains:

```
@WithMappers(BananaValueMappers.class)
```

Then the value mapper may define a custom `Configuration` value mapper with:

```
public final class BananaValueMappers {
    public static ValueMapper<Configuration> configuration() {
        return new ConfigurationValueMapper();
    }
}
```

Value mappers may be created for types that use parameterized types, but the parameterized types themselves must be either unbounded bounded or `Enum<>`. For example:

```
@Parameter
public List<? extends Foo> test;
```

or

```
@Parameter
public List<? extends Map<? super Foo, ? extends Bar>> test;
```

are not trivially resolved to a single value mapper due to the type constraints at play here.

What is a `SerializedValue`

When working with a `ValueMapper` it is inevitable that you will come across the `SerializedValue` type. This is the type we use for serializing all values that need serialization, such as activity parameters and resource values. In crafting a value mapper, you will have to both create a `SerializedValue` and parse one.

Constructing a `SerializedValue` tends to be more straightforward, because there are no questions about the structure of the value you are starting with. For basic types, you need only call `SerializedValue.of(value)` and the `SerializedValue` class will handle the rest. This can be done for values of the following types: `long`, `double`, `String`, `boolean`. Note that integers and floats can be represented by `long` and `double` respectively. For more complex types, you can also provide a `List<SerializedValue>` or `Map<String, SerializedValue>` to `SerializedValue.of()`. It is clear that these can be used to serialize lists and maps themselves, but arbitrarily complex structures can be serialized in this way. Consider the following examples:

```
int exInt = 5;
SerializedValue serializedInt = SerializedValue.of(exInt);

List<String> exList = List.of("a", "b", "c")
SerializedValue serializedList = SerializedValue.of(
    List.of(
        SerializedValue.of(exList.get(0)),
        SerializedValue.of(exList.get(1)),
        SerializedValue.of(exList.get(2))
    )
);

Map<String, Boolean> exMap = Map.of(
    "key1", true,
    "key2", false,
    "key3", true
);
SerializedValue serializedMap = SerializedValue.of(
    Map.of(
        "key1", SerializedValue.of(exMap.get("key1")),
        "key2", SerializedValue.of(exMap.get("key2")),
        "key3", SerializedValue.of(exMap.get("key3"))
    )
);

Vector3D exampleVec = new Vector3D(0,0,0);

SerializedValue serializedVec1 = SerializedValue.of(
    List.of(
        SerializedValue.of(exampleVec.getX()),
        SerializedValue.of(exampleVec.getY()),
        SerializedValue.of(exampleVec.getZ())
    )
);

SerializedValue serializedVec2 = SerializedValue.of(
    Map.of(
        "x", SerializedValue.of(exampleVec.getX()),
        "y", SerializedValue.of(exampleVec.getY()),
        "z", SerializedValue.of(exampleVec.getZ())
    )
);
```

The first 3 examples here are straightforward mappings from their java type to their serialized form, however the vector example is more interesting. To highlight this, two forms of `SerializedValue` have been given for it. In the first case, we serialize the `Vector3D` as a list of three values. This will work fine as long as whoever deserializes it knows that the list contains each component in order of x, y and z. In the second example, however, the vector is serialized as a map. Either of these representations may fit better in different scenarios. Generally, the structure of a `SerializedValue` constructed by a `ValueMapper` should match the `ValueSchema` the `ValueMapper` provides.

Example Activity Mapper

Below is an example of an Activity Type and its Activity mapper for reference:

Activity Type

```
@ActivityType("foo")
public final class FooActivity {
    @Parameter
    public int x = 0;

    @Parameter
    public String y = "test";

    @Parameter
    public List<Vector3D> vecs = List.of(new Vector3D(0.0, 0.0, 0.0));

    @Validation("x cannot be exactly 99")
    public boolean validateX() {
        return (x != 99);
    }

    @Validation("y cannot be 'bad'")
    public boolean validateY() {
        return !y.equals("bad");
    }

    @EffectModel
    public void run(final Mission mission) {
        // ...
    }
}
```

Generated Activity Mapper Example

```
package gov.nasa.jpl.aerie.foomissionmodel.generated.activities;

import gov.nasa.jpl.aerie.contrib.serialization.mappers.NullableValueMapper;
import gov.nasa.jpl.aerie.contrib.serialization.rulesets.BasicValueMappers;
import gov.nasa.jpl.aerie.foomissionmodel.Mission;
import gov.nasa.jpl.aerie.foomissionmodel.activities.FooActivity;
import gov.nasa.jpl.aerie.foomissionmodel.mappers.FooValueMappers;
import gov.nasa.jpl.aerie.merlin.framework.ModelActions;
import gov.nasa.jpl.aerie.merlin.framework.RootModel;
import gov.nasa.jpl.aerie.merlin.framework.ValueMapper;
import gov.nasa.jpl.aerie.merlin.protocol.model.Task;
import gov.nasa.jpl.aerie.merlin.protocol.model.TaskSpecType;
import gov.nasa.jpl.aerie.merlin.protocol.types.MissingArgumentException;
import gov.nasa.jpl.aerie.merlin.protocol.types.Parameter;
import gov.nasa.jpl.aerie.merlin.protocol.types.SerializedValue;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Optional;
import javax.annotation.processing.Generated;
import org.apache.commons.math3.geometry.euclidean.threed.Vector3D;

@Generated("gov.nasa.jpl.aerie.merlin.processor.MissionModelProcessor")
public final class FooActivityMapper implements TaskSpecType<RootModel<Mission>, FooActivity> {
    private final ValueMapper<Integer> mapper_x;

    private final ValueMapper<String> mapper_y;

    private final ValueMapper<List<Vector3D>> mapper_vecs;

    @SuppressWarnings("unchecked")
    public FooActivityMapper() {
        this.mapper_x =
            BasicValueMappers.$int();
        this.mapper_y =
            new NullableValueMapper<>() {
                BasicValueMappers.string();
            };
        this.mapper_vecs =
            new NullableValueMapper<>() {
                BasicValueMappers.list(
                    FooValueMappers.vector3d(
                        BasicValueMappers.$double()));
            };
    }

    @Override
    public String getName() {
        return "foo";
    }
}
```

```

@Override
public List<String> getRequiredParameters() {
    return List.of();
}

@Override
public ArrayList<Parameter> getParameters() {
    final var parameters = new ArrayList<Parameter>();
    parameters.add(new Parameter("x", this.mapper_x.getValueSchema()));
    parameters.add(new Parameter("y", this.mapper_y.getValueSchema()));
    parameters.add(new Parameter("vecs", this.mapper_vecs.getValueSchema()));
    return parameters;
}

@Override
public Map<String, SerializedValue> getArguments(final FooActivity activity) {
    final var arguments = new HashMap<String, SerializedValue>();
    arguments.put("x", this.mapper_x.serializeValue(activity.x));
    arguments.put("y", this.mapper_y.serializeValue(activity.y));
    arguments.put("vecs", this.mapper_vecs.serializeValue(activity.vecs));
    return arguments;
}

@Override
public FooActivity instantiate(final Map<String, SerializedValue> arguments) throws
    TaskSpecType.UnconstructableTaskSpecException {
    final var template = new FooActivity();
    Optional<Integer> x = Optional.ofNullable(template.x);
    Optional<String> y = Optional.ofNullable(template.y);
    Optional<List<Vector3D>> vecs = Optional.ofNullable(template.vecs);

    for (final var entry : arguments.entrySet()) {
        switch (entry.getKey()) {
            case "x":
                template.x = this.mapper_x.deserializeValue(entry.getValue()).getSuccessOrThrow($ -> new TaskSpecType.UnconstructableTaskSpecException());
                break;
            case "y":
                template.y = this.mapper_y.deserializeValue(entry.getValue()).getSuccessOrThrow($ -> new TaskSpecType.UnconstructableTaskSpecException());
                break;
            case "vecs":
                template.vecs = this.mapper_vecs.deserializeValue(entry.getValue()).getSuccessOrThrow($ -> new TaskSpecType.UnconstructableTaskSpecException());
                break;
            default:
                throw new TaskSpecType.UnconstructableTaskSpecException();
        }
    }

    if (!x.isPresent()) throw new MissingArgumentException("activity", "foo", "x", this.mapper_x.getValueSchema());
    if (!y.isPresent()) throw new MissingArgumentException("activity", "foo", "y", this.mapper_y.getValueSchema());
    if (!vecs.isPresent()) throw new MissingArgumentException("activity", "foo", "vecs", this.mapper_vecs.getValueSchema());

    return template;
}

@Override
public List<String> getValidationFailures(final FooActivity activity) {
    final var failures = new ArrayList<String>();
    if (!activity.validateX()) failures.add("x cannot be exactly 99");
    if (!activity.validateY()) failures.add("y cannot be 'bad'");
    return failures;
}

@Override
public Task createTask(final RootModel<Mission> model, final FooActivity activity) {
    return ModelActions.threaded(() -> activity.run(model.model())).create(model.executor());
}
}

```

Value Schemas

A Value What?

A value *schema*! A value schema is a description of the structure of some value. Using value schemas, users can tell our system how to work with arbitrarily complex types of values, so long as they can be described using the value schema constructs provided by Merlin. Let's take a look at how it's done.

Starting With The Basics

At a fundamental level, a value schema is no more than a combination of elementary value schemas. Merlin defines the elementary value schemas, so let's take a look at them:

- `REAL` : A real number
- `INT` : An integer
- `BOOLEAN` : A boolean value
- `STRING` : A string of characters
- `DURATION` : A duration value
- `PATH` : A file path
- `VARIANT` : A string value constrained to a set of acceptable values.

If you are trying to write a value schema for an integer value, all you have to do is use the `INT` value schema, but of course values can quickly take on more complex structures, and for that we must examine the remaining value schema constructs.

A Note About Variants

The `Variant` value schema is a little unique among the elementary value schemas in that it requires input, the set of acceptable values. The way to provide this set of values depends on the context in which you are creating a value schema and will be addressed in the corresponding section below.

Building Things Up

In order to combine elementary value schemas, we provide two main constructs:

- `SERIES` : Denotes a list of values of a single type
- `STRUCT` : Denotes a structure of independent values of varying types

The `SERIES` node allows a straightforward declaration of a list of values that fall under the same schema, while the `STRUCT` node opens things up, allowing you to create any combination of different values, each labeled by some string name.

Now that you've seen the basics, let's talk about the two different ways to create value schemas -- in code and in JSON/GraphQL (serialized value schemas).

Value Schemas From Code

Creating a value schema from code is straightforward thanks to the `ValueSchema` class. Each of the elementary value schemas is accessible as via the `ValueSchema` class. For example, a `REAL` is given by `ValueSchema.REAL`. The one exception is that to create a `VARIANT` type value schema you'll need to call `ValueSchema.ofVariant(Class<? extends Enum> enum)`, providing an enum with to specify the acceptable variants.

Like the `VARIANT` element, the `SERIES` and `STRUCT` constructs are created by calling their corresponding methods `ValueSchema.ofSeries(ValueSchema value)` and `ValueSchema.ofStruct(Map<String, ValueSchema> map)`.

Examples Using `ValueSchema`

Below are a few examples of how to create a `ValueSchema`. In each, a Java type and its corresponding `ValueSchema` are compared.

- `Integer` is described by `ValueSchema.INT`

- `List<Double>` is described by `ValueSchema.ofSeries(ValueSchema.REAL)`
- `Float[]` is described by `ValueSchema.ofSeries(ValueSchema.REAL)`

Note that the second and third examples are entirely different Java types, but are represented by the same `ValueSchema`. It is also important to take a look at a `Map` type, as it can be confusing at first how to represent its structure:

`Map<String, Integer>` is described by

```
ValueSchema.ofStruct(
    Map.of(
        "keys": ValueSchema.ofSeries(ValueSchema.STRING),
        "values": ValueSchema.ofSeries(ValueSchema.INT)
    )
)
```

Here we are taking note of the fact that a `Map` is really just a list of keys and a list of values. As a final example, consider the custom type below:

```
public class CustomType {
    public int foo;
    public boolean bar;
    public List<String> baz
}
```

A variable of type `CustomType` has structure described by:

```
ValueSchema.ofStruct(
    Map.of(
        "foo": ValueSchema.INT,
        "bar": ValueSchema.BOOLEAN,
        "baz": ValueSchema.ofSeries(ValueSchema.STRING)
    )
)
```

Serialized Value Schemas

Creating value schemas from JSON/GraphQL is a little less straightforward, since your IDE won't be able to help you, but fear not, you've come to the right place. A value schema is created by declaring an object with a `type` field that tells which type of schema is being created. The values allowed in this field are given below:

- `"real"` corresponds to `REAL`
- `"int"` corresponds to `INT`
- `"boolean"` corresponds to `BOOLEAN`
- `"string"` corresponds to `STRING`
- `"duration"` corresponds to `DURATION`
- `"path"` corresponds to `PATH`
- `"variant"` corresponds to `VARIANT`
- `"series"` corresponds to `SERIES`
- `"struct"` corresponds to `STRUCT`

Variant

For the `"variant"` type, you'll need to include a second field called `variants` whose value is a list of objects specifying the string-valued `key` and `label` fields of each variant like this:

```
{
  "type": "variant",
  "variants": [
    {
      "key": "ON",
      "label": "ON"
    },
    {
      "key": "OFF",
      "label": "OFF"
    }
  ]
}
```

Series

For the `"series"` type, a second field called `"items"` must be included as well that provides the value schema for the items in the series. See the below example, a value schema for a list of integers.

```
{
  "type": "series",
  "items": {
    "type": "int"
  }
}
```

Struct

Lastly, for the `"struct"` type, a second field called `"items"` must be included that provides the actual structure of the struct, mapping string keys to their corresponding value schema. See the below example, a value schema for a struct with a string-valued `label` field, real-valued `position` field, and boolean-valued `on` field:

```
{
  "type": "struct",
  "items": {
    "label": { "type": "string" },
    "position": { "type": "real" },
    "on": { "type": "boolean" }
  }
}
```

Examples Creating Serialized Value Schemas

Below are more examples of creating serialized value schemas using JSON:

1. A value schema for an integer:

```
{
  "type": "int"
}
```

2. A value schema for a list of paths:

```
{
  "type": "series",
  "items": {
    "type": "path"
  }
}
```

3. A value schema for a list of lists of booleans:

```
{
  "type": "series",
  "items": {
    "type": "series",
    "items": {
      "type": "boolean"
    }
  }
}
```

4. A value schema for a structure containing a list of integers labeled `lints` , and a boolean labeled `active` :

```
{
  "type": "struct",
  "items": {
    "lints": {
      "type": "series",
      "items": { "type": "int" }
    },
    "active": {
      "type": "boolean"
    }
  }
}
```

Declaring Parameters

The Merlin interface offers a variety of ways to define **parameters** for mission model configurations and activities. Parameters offer a concise way to export information across the mission-agnostic Merlin interface – namely a parameter's type to support serialization and a parameter's "required" status to ensure that parameters without mission-model-defined defaults have an argument supplied by the planner.

In this guide **parent class** refers to the Java class that encapsulates parameters. This class may take the form of either a mission model configuration or activity.

Both configurations and activities make use of the same Java annotations for declaring parameters within a parent class. The `@Export` annotation interface serves as the common qualifier for exporting information across the mission-agnostic Merlin interface. The following parameter annotations serve to assist with parameter declaration and validation:

- `@Export.Parameter``
- `@Export.Template``
- `@Export.WithDefaults``
- `@Export.Validation``

The following sections delve into each of these annotations along with examples and suggested use cases for each.

Declaration

Without Export Annotations

The first – and perhaps less obvious option – is to not use any parameter annotations. If a parent class contains no `@Export.Parameter``, `@Export.Template``, or `@Export.WithDefaults`` annotation it is assumed that every class member variable is a parameter to export to Merlin.

Defining a parent class becomes as simple as `public record Configuration(Integer a, Integer b, Integer c) { }`. However, it is not possible to declare a member variable that is not an exported parameter with this approach.

Example

```
@ActivityType("Simple")
public record SimpleActivity(Integer a, Integer b, Integer c) {

    @EffectModel
    public void run(final Mission mission) {
        mission.count.set(a);
        delay(1, SECOND);
        mission.count.set(b);
        delay(1, SECOND);
        mission.count.set(c);
    }
}
```

In the above example `a`, `b`, and `c` are all parameters that require planner-provided arguments at planning time.

See Also

Aerie's `examples/config-without-defaults` makes use of this succinct style for declaring mission model configuration parameters.

For more information on records in Java 16+, see [Java Record Classes](#).

Recommendation

Avoid `@Export.Parameter``, `@Export.Template``, or `@Export.WithDefaults`` when every member variable for a parent class should be an exported parameter without a default value.

This approach is great for defining a simple `record` type parent class without defaults. If defaults are desired then `@Template` or `@WithDefaults` can be used without changing a `record` type class definition.

`@Export.Parameter``

The `@Parameter` annotation is the most explicit way to define a parameter and its defaults. Explicitly declaring each parameter within a parent

class with or without a default value gives the mission modeler the freedom to decide which member variables are parameters and which parameters are required by the planner.

Example

```
public final class Configuration {

    public Integer a;

    @Export.Parameter
    public Integer b;

    @Export.Parameter
    public Integer c = 3;

    public Configuration(final Integer a, final Integer b, final Integer c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }
}
```

In the above example the parent class is a mission model configuration. Here's a close look at each member variable:

- `a`: a traditional member variable. Not explicitly declared as a parameter and therefore is not considered to be a parameter.
- `b`: explicitly declared as a parameter without a default value. A value will be required by the planner.
- `c`: explicitly declared as a parameter with a default value. A value will not be required by the planner.

See Also

Aerie's `examples/foo-missionmodel` uses this style when declaring mission model configuration parameters.

Recommendation

Declare each parameter with a `@Parameter` when a non-`record` type parent class is desired.

Some mission modelers may prefer the explicitness provided by individual `@Parameter` annotations. However, this opens the door to subtle mistakes such as an unintentionally absent `@Parameter` annotation or an unintentionally absent default assignment. Those who prefer a more data-oriented approach may also find this style to not be as ergonomic as using a simple `record` type.

`@Export.Template`

The `@Template` annotation decouples parameter definitions and default values, allowing `record` types to be used as parent classes. When the `@Template` annotation is used every parent class member variable is interpreted as a parameter to export to Merlin. This annotation must be attached to a `public static` constructor method.

Example

```
@ActivityType("ThrowBanana")
public record ThrowBananaActivity(double speed) {

    @Template
    public static ThrowBananaActivity defaults() {
        return new ThrowBananaActivity(1.0);
    }

    @Validation("Speed must be positive")
    public boolean validateBiteSize() {
        return this.speed() > 0;
    }

    @EffectModel
    public void run(final Mission mission) {
        mission.plant.add(-1);
    }
}
```

In the above example `ThrowBananaActivity` is a `record` type with one constructor parameter, `speed`.

See Also

Aerie's `examples/banananation` uses this style within `GrowBananaActivity` and `ThrowBananaActivity`.

Recommendation

Use `@Template` when every member variable for a parent class should be an exported parameter with a default value.

@WithDefaults

Similarly to `@Template`, `@WithDefaults` annotation also decouples parameter definitions and default values, allowing `record` types to be used as parent classes. When the `@WithDefaults` annotation is used every parent class member variable is interpreted as a parameter to export to Merlin. Unlike `@Template`, a sparse set of default values may be supplied.

This annotation must be attached to a nested `public static class` within the parent class. Each member variable of this nested class must have the same name as a parent class's member variable. Not every parent class member variable is required to have an associated member variable within the nested class. This allows the mission modeler to selectively choose which parameters must be supplied by the planner.

Example

```
public record Configuration(Integer a, Double b, String c) {  
  
    @WithDefaults  
    public static final class Defaults {  
        public static Integer a = 42;  
        public static String c = "JPL";  
    }  
}
```

In the above example the parent class is a mission model configuration. Here's a close look at each member variable:

- `a`: a parameter with an associated default value.
- `b`: a parameter without a default value. A value will be required by the planner.
- `c`: a parameter with an associated default value.

See Also

Aerie's `examples/config-with-defaults` uses this style within its mission model configuration. The `examples/banananation` mission model also uses this style within `BakeBananaBreadActivity`.

Recommendation

Use `@WithDefaults` when every member variable for a parent class should be an exported parameter with an optionally provided default value.

Validation

A mission model configuration or activity instance can be validated by providing one or more methods annotated by `@Validation`. The annotation message specifies the message to present to a planner when the validation fails. For example:

```
@Validation("instrument power must be between 0.0 and 1000.0")  
public boolean validateInstrumentPower() {  
    return (instrumentPower_W >= 0.0) && (instrumentPower_W <= 1000.0);  
}
```

The Merlin annotation processor identifies these methods and arranges for them to be invoked whenever the planner instantiates an instance of the class. A message will be provided to the planner for each failing validation, so the order of validation methods does not matter.

Models & Resources

Mission Resources

In Merlin, a resource is any measurable quantity whose behavior is to be tracked over the course of a simulation. Resources are general-purpose, and can model quantities such as finite resources, geometric attributes, ground and flight events, and more. Merlin provides basic models for three types of quantity: a discrete quantity that can be set (`Register`), a continuous quantity that can be added to (`Counter`), and a continuous quantity that grows autonomously over time (`Accumulator`).

A common example of a `Register` would be a spacecraft or instrument mode, while common `Accumulator` s might be battery capacity or data volume.

Defining a resource is as simple as constructing a model of the appropriate type. The model will automatically register its resources for use from the Aerie UI. Alternatively, a resource may be **derived** or **sampled** from an existing resource.

Derived Resources

A derived resource is constructed from an existing resource given a mapping transformation.

For example, the `Imager` sample model defines an "imaging in progress" resource with:

```
this.imagingInProgress = this.imagerMode.map($ -> $ != ImagerMode.OFF);
```

In this example, `imagingInProgress` is a full-fledged discrete resource and will depend only on the imager's on/off state.

A derived resource may also be constructed from a real resource. For example, given `Accumulator` s `instrumentA` and `instrumentB` , a resource that maintains the current sum of both volumes may be constructed with:

```
var sumResource = instrumentA.volume.resource.plus(instrumentB.volume.resource);
```

Sampled Resources

A sampled resource allows for a new resource to be constructed from arbitrarily many existing resources/values and to be sampled once per second. This differs from a derived resource which provides a continuous mapping transformation from a single existing resource.

For example, the `Mission` sample model defines a "battery state of charge" resource with:

```
this.batterySoC = new SampledResource<>(() -> this.source.volume.get() - this.sink.volume.get());
```

In this example, `batterySoC` will be updated once per second to with the current difference between the "source" volume and "sink" volume.

Custom models

Often, the semantics of the pre-existing models are not exactly what you need in your adaptation. Perhaps you'd like to prevent activities from changing the rate of an `Accumulator` , or you'd like to have some helper methods for interrogating one or more resources. In these cases, a custom model may be a good solution.

A custom model is a regular Java class, extending the `Model` class generated for your adaptation by Merlin (or the base class provided by the framework, if it's mission-agnostic). It may implement any helper methods you'd like, and may contain any sub-models that contribute to its purpose. The only restriction is that it **must not** contain any mutable state of its own -- all mutable state must be held by one of the basic models, or one of the internal state-management entities they use, known as "cells".

The `contrib` package is a rich source of example models. See [the repository](#) for more details.

Creating Plans

Plans can be created via [Planning Web GUI](#) or interfacing with the [Aerie GraphQL API](#).

Constraints

- [Defining Constraints](#)
- [Creating A Constraint](#)
- [Constraint Examples](#)
- [Constraint Violation Examples](#)
- [Constraint Definition Nodes](#)

Overview

When analyzing a simulation's results, it may be useful to detect windows where certain conditions are met. Constraints are the Aerie tool for fulfilling that role. A constraint is a condition on activities and resources that must hold through an entire simulation. If a constraint does not hold true at any point in a simulation, this is considered a violation. The results yielded by a simulation run will include a list of violation windows for every constraint that has been violated.

Defining Constraints

Constraints are defined by JSON via the Aerie GraphQL API or in the Aerie UI. To define a constraint, a JSON object must be constructed matching the format of the various nodes we have defined (see our [comprehensive list of constraint definition nodes](#)). Each node has a set of required fields, and a return type. For example, the `Transition` node requires a discrete resource, an "old" state and a "new" state, and returns a set of windows where that resource transitions from the "old" state to the "new" state.

Activity constraints are special in that the activity types involved must be declared at the start of the definition using the `ForEachActivity` node. This node requires three arguments: the activity type, an alias, and a constraint expression. The activity type should be the type of the activity being constrained, and the expression should be either another `ForEachActivity` or an expression that yields violation windows. The alias, however, is a string that is used to represent an instance of the activity type while defining the rest of the constraint expression. When evaluating constraints, this alias is replaced with each instance of the activity type and evaluated. For examples of this, see [below](#).

Creating a Constraint in Aerie

Constraints can be created via the Aerie GraphQL API. For an example of how to create a constraint for a mission model see the example, [Creating a constraints for a mission model](#) detailed in the [Aerie GraphQL API Software Interface Specification](#).

Constraint Examples

To define a constraint, you will need to build it up from nodes of our [constraint syntax tree](#). To help get you started, here are a few examples:

Constraint Example 1

Let's start off with a basic constraint that a resource, let's call it `BatteryTemperature`, doesn't exceed some threshold, say 340. We do so by using `RealResource` to get the `BatteryTemperature` resource, and `RealValue` to get a real number we can compare a real resource profile to.

```
{
  "type": "LessThanOrEqual",
  "left": {
    "type": "RealResource",
    "name": "BatteryTemperature"
  },
  "right": {
    "type": "RealValue",
    "value": 340
  }
}
```

Constraint Example 2

Now we examine a more complex constraint. Let's imagine a solar panel that rotates the panels to a certain angle. Suppose the panels can rotate as fast as 5 degrees per second, but are not allowed to go more than 3 degrees per second unless the spacecraft is operating in IDLE mode. For this we will use a real resource, `PanelAngle`, and a discrete resource, `OpMode`.

Note that this breaks down to two conditions, either of which must be true the entire simulation. This constraint should be satisfied as as either:

1. The `OpMode` is `"IDLE"`
2. The rate of the `PanelAngle` is no more than 3 degrees per second

```
{
  "type": "Or",
  "expressions": [
    {
      "type": "EqualTo",
      "left": {
        "type": "DiscreteResource",
        "name": "OpMode"
      },
      "right": {
        "type": "DiscreteValue",
        "value": "IDLE"
      }
    },
    {
      "type": "LessThan",
      "left": {
        "type": "Rate",
        "profile": {
          "type": "RealResource",
          "name": "PanelAngle"
        }
      },
      "right": {
        "type": "RealValue",
        "value": 3
      }
    }
  ]
}
```

Constraint Example 3

The first example of an activity constraint we present says that whenever an instance of `ActivityTypeA` occurs, the value of `ResourceX` must be less than 10.0. Notice the top level expression inside the `ForEachActivity` is an `Or` with the first expression meaning "not during an instance of activity A". This crucial step is what says that when the instance is not active, the reset of the constraint doesn't apply. If this part were left out, the constraint would say that `ResourceX` must be less than 10.0 throughout the entire simulation.

```
{
  "type": "ForEachActivity",
  "activityType": "ActivityTypeA",
  "alias": "actA",
  "expression": {
    "type": "Or",
    "expressions": [
      {
        "type": "Not",
        "expression": {
          "type": "During",
          "alias": "actA"
        }
      },
      {
        "type": "LessThan",
        "left": {
          "type": "RealResource",
          "name": "ResourceX"
        },
        "right": {
          "type": "RealValue",
          "value": 10.0
        }
      }
    ]
  }
}
```

Constraint Example 4

As a final example, we present a complex constraint containing two activity types, and several nested expressions. Most constraints should be much simpler than this, but this demonstrates just how capable the constraint syntax tree is.

The constraint below basically says this: For each pair of activities of type TypeA and TypeB, during the intersection of the two activities either parameter `b` of the TypeB instance must be false, or Resource ResC must be no greater than half of parameter `a` of the TypeA instance. Quite

a mouthful, but here it is:

```
{
  "type": "ForEachActivity",
  "activityType": "TypeA",
  "alias": "A",
  "expression": {
    "type": "ForEachActivity",
    "activityType": "TypeB",
    "alias": "B",
    "expression": {
      "type": "Or",
      "expressions": [
        {
          "type": "Or",
          "expressions": [
            {
              "type": "Not",
              "expression": {
                "type": "LessThan",
                "left": {
                  "type": "Times",
                  "profile": {
                    "type": "RealResource",
                    "name": "ResC"
                  },
                  "multiplier": 2.0
                },
                "right": {
                  "type": "AsReal",
                  "expression": {
                    "type": "Parameter",
                    "alias": "A",
                    "name": "a"
                  }
                }
              }
            },
            {
              "type": "Equal",
              "left": {
                "type": "DiscreteValue",
                "value": false
              },
              "right": {
                "type": "Parameter",
                "alias": "B",
                "name": "b"
              }
            }
          ]
        },
        {
          "type": "Not",
          "expression": {
            "type": "During",
            "alias": "A"
          }
        },
        {
          "type": "Not",
          "expression": {
            "type": "During",
            "alias": "B"
          }
        }
      ]
    }
  }
}
```

Violation Examples

Constraint violations contain two sets of information describing where constraints are violated. First, a list of associated activity instance IDs representing the activity instances in violation (this will be an empty list for constraints that don't involve activities). Second, the list of violation windows themselves tells when during the simulation violations occur.

Constraint violations are reported per activity instance, so it is entirely possible for multiple violations to be produced by a single constraint. This unambiguous representation clearly indicates activity instances that violate a constraint despite the constraint being defined at the type-level.

Below are several examples of constraint violations:

A single activity instance with ID "2" is in violation from time 5 to 7:

```
{
  "activityInstanceIds": [ "2" ],
  "windows": [ [5, 7] ]
}
```

A constraint is violated from 2 to 4 and also from 5 to 8. No activities are involved in this violation.

```
{
  "activityInstanceIds": [],
  "windows": [ [2, 4], [5, 8] ]
}
```

Constraint Definition Nodes

Below we list all constraint definition nodes currently implemented in Aerie along with their required fields and return type. When defining a constraint, use names exactly as they appear here:

Activity Constraint Related Nodes

Activity constraint related nodes make use of an `alias` field to represent individual activity instances. At constraint evaluation time, this alias is replaced with each instance of an activity type one at a time to determine which instances violate the provided constraint on an individual basis.

ForEachActivity

- **Return Type:** List of lists of violation windows
- **Return Value:** One list of violation windows for each activity instance in violation of a constraint
- **Required Fields:**
 - `activityType`: String representation of the activity type of interest
 - `alias`: String to represent instances of the provided activity type in the provided expression
 - `expression`: Either `ForEachActivity` node or any node that returns a set of windows

ForbiddenActivityOverlap

- **Return Type:** List of lists of violation windows (each inner list will have exactly one element, the violation window)
- **Return Value:** A list of violation windows where each forbidden overlap occurs
- **Required Fields:**
 - `activityType1`: String representation of the first activity type
 - `activityType2`: String representation of the second activity type

InstanceCardinality

- **Return Type:** List of violation windows
- **Return Value:** A list of violation windows where the number of instances of a given activity type is greater/less than the constrained value
- **Required Fields:**
 - `activityType`: String representation of the activity type of interest
 - `minimum`: Integer representation of the minimum instances that should occur throughout the duration of a plan
 - `maximum`: Integer representation of the maximum instances that should occur throughout the duration of a plan

During

- **Return Type:** Set of windows
- **Return Value:** The window from start to end of an activity instance

- **Required Fields:**

- `alias` : String representing an activity instance as defined in a `ForEachActivity` node

`StartOf`

- **Return Type:** Set of windows

- **Return Value:** The start time of an activity instance

- **Required Fields:**

- `alias` : String representing an activity instance as defined in a `ForEachActivity` node

`EndOf`

- **Return Type:** Set of windows

- **Return Value:** The end time of an activity instance

- **Required Fields:**

- `alias` : String representing an activity instance as defined in a `ForEachActivity` node

`DiscreteParameter`

- **Return Type:** `DiscreteProfile`

- **Return Value:** Discrete profile of the specified parameter's value over the simulation bounds

- **Required Fields:**

- `alias` : String representing an activity instance as defined in a `ForEachActivity` node
- `name` : String name of activity parameter to get the value of

`RealParameter`

- **Return Type:** `LinearProfile`

- **Return Value:** Linear profile of the specified parameter's value over the simulation bounds

- **Required Fields:**

- `alias` : String representing an activity instance as defined in a `ForEachActivity` node
- `name` : String name of a real-valued activity parameter to get the value of

Resource Profile Nodes

Resource constraint related nodes are generally used to define constraints on resources, though there are cases when they can be used in other ways i.e. constraining an activity parameter be less than some value.

`RealResource`

- **Return Type:** Linear profile

- **Return Value:** The profile of a real resource, or linear profile sourced from a real-valued discrete profile

- **Required Fields:**

- `name` : String name of the resource to get the profile of

`RealValue`

- **Return Type:** Linear profile

- **Return Value:** A profile across the simulation bounds with constant value as provided

- **Required Fields:**

- `value` : Real number to build a profile from

Plus

- **Return Type:** Linear profile
- **Return Value:** The sum of two linear profiles
- **Required Fields:**
 - `left` : An expression that yields a linear profile
 - `right` : An expression that yields a linear profile

Times

- **Return Type:** Linear profile
- **Return Value:** The profile achieved by multiplying a given profile by a real number
- **Required Fields:**
 - `profile` : An expression that yields a linear profile
 - `multiplier` : Real number to multiply the profile by

Rate

- **Return Type:** Linear profile
- **Return Value:** Linear profile representing the rate of change of another linear profile
- **Required Fields:**
 - `profile` : An expression that yields a linear profile

DiscreteResource

- **Return Type:** Discrete profile
- **Return Value:** The profile of a resource
- **Required Fields:**
 - `name` : String name of the resource to get the profile of

DiscreteValue

- **Return Type:** Discrete profile
- **Return Value:** A profile across the simulation bounds with constant value as provided
- **Required Fields:**
 - `value` : Any value that can be [serialized](#)

Window Supplier Nodes

Transition

- **Return Type:** Set of windows
- **Return Value:** Set of points where a discrete profile exhibits transition from one state to another
- **Required Fields:**
 - `profile` : Any expression that yields a discrete profile
 - `from` : A [serializable](#) value representing the state the profile must transition from
 - `to` : A [serializable](#) value representing the state the profile must transition from

Changed

- **Return Type:** Set of windows
- **Return Value:** All windows where a given profile is not constant
- **Required Fields:**
 - **expression**: Any expression yielding a profile

Equal

- **Return Type:** Set of windows
- **Return Value:** Windows where a one profile is equal to another
- **Required Fields:**
 - **left**: Expression yielding a profile to be equal to another
 - **right**: Expression yielding a profile to compare **left** against (must be same type of profile)

NotEqual

- **Return Type:** Set of windows
- **Return Value:** Windows where a one profile is not equal to another
- **Required Fields:**
 - **left**: Expression yielding a profile to be not equal to another
 - **right**: Expression yielding a profile to compare **left** against (must be same type of profile)

LessThan

- **Return Type:** Set of windows
- **Return Value:** Windows where a linear profile is less than another linear profile
- **Required Fields:**
 - **left**: Expression yielding a linear profile to be less than another
 - **right**: Expression yielding a Linear profile to compare **left** against

GreaterThan

- **Return Type:** Set of windows
- **Return Value:** Windows where a linear profile is greater than another linear profile
- **Required Fields:**
 - **left**: Expression yielding a linear profile to be greater than to another
 - **right**: Expression yielding a linear profile to compare **left** against

LessThanOrEqual

- **Return Type:** Set of windows
- **Return Value:** Windows where a linear profile is less than or equal to another linear profile
- **Required Fields:**
 - **left**: Expression yielding a linear profile to be less than or equal to another
 - **right**: Expression yielding a linear profile to compare **left** against

GreaterThanOrEqual

- **Return Type:** Set of windows
- **Return Value:** Windows where a linear profile is greater than or equal to another linear profile
- **Required Fields:**
 - `left` : Expression yielding a linear profile to be greater than or equal to another
 - `right` : Expression yielding a linear profile to compare `left` against

And

- **Return Type:** Set of windows
- **Return Value:** Intersection of windows from all provided expressions
- **Required Fields:**
 - `expressions` : List of expressions that yield sets of windows

Or

- **Return Type:** Set of windows
- **Return Value:** Union of windows from all provided expressions
- **Required Fields:**
 - `expressions` : List of expressions that yield sets of windows

Not

- **Return Type:** Set of windows
- **Return Value:** The subtraction of windows from an expression from simulation bounds
- **Required Fields:**
 - `expression` : Expression yielding sets of windows

IfThen

- **Return Type:** Set of windows
- **Return Value:** The `Not` of the condition `Or`'d with the expression
- **Required Fields:**
 - `condition` : Expression yielding a set of windows
 - `expression` : Expression yielding a set of windows

Precomputed Profiles

The What

Precomputed profiles are quite similar to the profiles produced by a Merlin simulation. They describe the behavior of some dynamic characteristic over time and can be viewed in the Aerie UI along with a plan. The big difference between precomputed profiles as those that come from simulation is that precomputed profiles are... well... precomputed. Precomputed profiles come from the user, rather than Merlin's simulation capabilities, and can be uploaded at any time, before or after simulation.

The Why

Ultimately, the purpose of precomputed profiles is up to the user. You might be looking to include some geometry information to assist in building a plan, or you may simply be adding power and thermal modeling results to be viewable with existing simulation results. Just be aware that at this time precomputed profiles are not accessible to the mission model during simulation, and are simply for viewing purposes in the UI.

The How

To upload a set of precomputed profiles, you will need to add what is called an external dataset to a plan. An external dataset is just a set of profiles with a start time from which the profiles are based. The dataset must be associated with a plan, and will persist as long as the plan exists, or until it is deleted by a user. An external dataset can be added via the GraphQL mutation `addExternalDataset`.

GraphQL Mutation: `addExternalDataset`

The `addExternalDataset` GraphQL mutation takes three parameters as specified below:

- `planId` : Type: String Description: The ID of the plan to associate the external dataset with.
- `datasetStart` : Type: String Description: The UTC timestamp the dataset is based from. UTC Format: `yyyy-dddThh:mm:ss`
- `profileSet` : Type: Object Description: The set of precomputed profiles that make up the external dataset.

The profile set to be uploaded should have one entry for each precomputed profile, indexed by a unique name mapping to an object specifying the details of the profile. Each profile should have a `type` field, which specifies whether the profile is real- or discrete-valued. A discrete profile must also contain a `schema` field, which specifies the schema of the values it takes on. These are not limited to basic types, but can take on any complex structure made up using our `ValueSchema` construct (for more information, see our [ValueSchema documentation](#)).

Finally, both profile types require a list of segments that describe the actual behavior of the profile. The `segments` field of each profile maps to a list of objects, each of which details a segment. Each segment should contain the following two fields:

- `duration` : Type: Integer Description: The duration (in microseconds) the segment's dynamics hold before the next segment begins.
- `dynamics` : Type: Dependent on profile type. Description: The behavior of the profile over the lifetime of this segment.

A discrete profile's dynamics should match the format specified by the `schema` field, while a real profile's dynamics should always contain an initial value and a rate of change. See our example external dataset mutations [below](#) to see both profile specification types.

Deleting External Datasets

There may be a time when you find an external dataset you've been using is no longer relevant, and must be removed. This is easily done by providing the ID of the dataset you wish to delete to the `delete_dataset_by_pk` mutation. For example, the following mutation will delete the dataset with id 5:

```
mutation deletedDataset {
  delete_dataset_by_pk(id: 5) {
    id
  }
}
```

Example External Dataset Mutations

Example 1

This example shows an external dataset being uploaded to the plan with ID `2` starting at `2018-331T04:00:00`. Two precomputed profiles are

included in the external dataset. First a real profile called `batteryEnergy` starts at a value of 50 and decreases at a rate of -0.5 units per second over 30 seconds. At that point, the value is 35 and the rate is changed to -0.1 units per second for 30 more seconds. The second profile, a discrete profile called `awake` contains a schema that tells us its values are boolean. The segments tell us that for the first 30 seconds the profile's dynamics are the value `true` and for the next 30 seconds the value `false`. At the end we see the ID of the external dataset that is created is queried as the result of this mutation.

```
mutation {
  addExternalDataset(
    planId: "2",
    datasetStart: "2018-331T04:00:00",
    profileSet: {
      batteryEnergy: {
        type: "real",
        segments:[
          {
            duration: 30000000,
            dynamics: {
              initial: 50,
              rate: -0.5
            }
          },
          {
            duration: 30000000,
            dynamics: {
              initial: 35,
              rate: -0.1
            }
          }
        ]
      },
      awake: {
        type: "discrete",
        schema: {
          type: "boolean"
        },
        segments: [
          {
            duration: 30000000,
            dynamics: true
          },
          {
            duration: 30000000,
            dynamics: false
          }
        ]
      }
    }
  ) {
    datasetId
  }
}
```

Example 2

This mutation adds to plan with id `7` an external dataset starting at `2038-192T14:00:00` with a single precomputed profile, `orientation`. This discrete profile's schema tells us that its values are structs with real-valued `x`, `y` and `z` fields. For the first hour, the profile takes a value of `x=0`, `y=0`, `z=1`. For the next hour thereafter, the profile is valued at `x=1`, `y=1`, `z=0`.

```

mutation {
  addExternalDataset(
    planId: "7",
    datasetStart: "2038-192T14:00:00",
    profileSet: {
      orientation: {
        type: "discrete",
        schema: {
          type: "struct",
          items: {
            x: { type: "real" },
            y: { type: "real" },
            z: { type: "real" }
          }
        }
      }
      segments:[
        {
          duration: 3600000000,
          dynamics: {
            x: 0,
            y: 0,
            z: 1
          }
        },
        {
          duration: 3600000000,
          dynamics: {
            x: 1,
            y: 1,
            z: 0
          }
        }
      ]
    }
  ) {
    datasetId
  }
}

```

Automated Scheduling

Note: Automated scheduling eDSL is under heavy development. This documentation will change regularly.

Introduction

This guide explains how to use the scheduling service with the latest version of Aerie.

The scheduling service allows you to add activities to a plan based on goals that you define. These goals are defined in [TypeScript](#), using a library provided by Aerie.

This guide assumes some very basic familiarity with programming (terminology like functions, arguments, return values, types, and objects will not be explained), but it does not assume the reader has seen TypeScript or JavaScript before.

Prerequisites

- You will need to have uploaded a Mission Model
- You will need to have created a plan from that mission model

Steps

1. [Authoring a new goal](#)
2. [Specifying the order of goals](#)
3. [Running the scheduler](#)

Authoring a new goal

In the Aerie UI, open the scheduling pane, and click **New Goal**. This will open a text editor, with the following default text:

```
export default (): Goal => {  
  // Your code here  
}
```

This is a TypeScript function that takes no arguments and returns a Goal.

To unpack all of the parts:

- **export default** signals to Aerie that this is the function that defines the Goal.
- **() => {}** in TypeScript is called an [arrow function](#).
 - The parenthesis **()** represent the parameters that the function takes. Scheduling goals cannot take any parameters, so these parenthesis must be empty.
 - The curly braces **{}** represent the definition of the goal. The return statement for the function must go inside the braces.
- The **: Goal** part signifies that this function returns a Goal. TypeScript will check that the function does indeed return a Goal - if it does not, it will underline your code in red.

The code provided when you click **New Goal** is incomplete - the function does not yet return a Goal, so you should see the word **Goal** underlined

Goal Definition

in red:

```
1  export default (): Goal => {  
2  
3  }
```

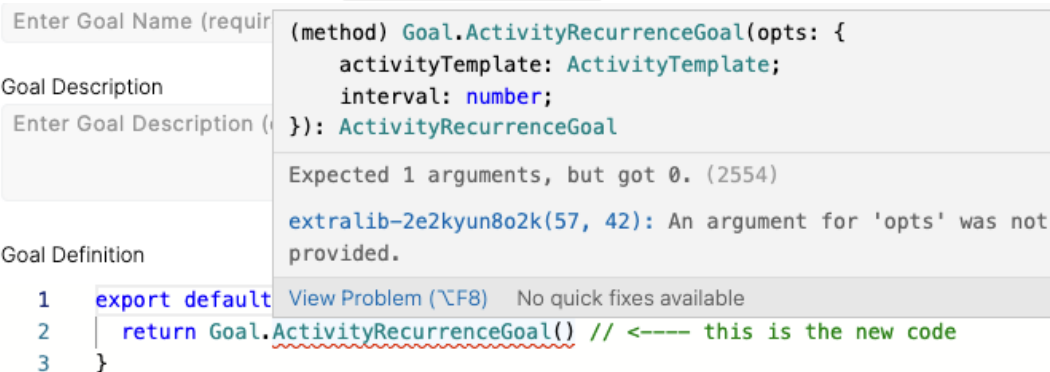
Mousing over the word Goal, you should see something akin to the following message:

A function whose declared type is neither 'void' nor 'any' must return a value.

This message means that the function has promised to return a value, but it currently lacks a return statement. Between the curly braces, add the following code: **return Goal.ActivityRecurrenceGoal();**

```
export default (): Goal => {
  return Goal.ActivityRecurrenceGoal() // <---- this is the new code
}
```

Now, the editor should tell you that `ActivityRecurrenceGoal()` takes one argument.



```
1 export default
2   | return Goal.ActivityRecurrenceGoal() // <---- this is the new code
3   }
```

The argument that we're missing is the "options" object. Objects in typescript are defined using curly braces `{}` with key-value pairs, like so:

```
{
  key: value
}
```

If we pass an empty object `{}` to `ActivityRecurrenceGoal`, we will get a new error message that will tell us what keys our object will need:

```
export default (): Goal => {
  return Goal.ActivityRecurrenceGoal({}) // <---- the empty object is written as {}
}
```

```
Argument of type '{}' is not assignable to parameter of type '{ activityTemplate: ActivityTemplate; interval: number; }'.
Type '{}' is missing the following properties from type '{ activityTemplate: ActivityTemplate; interval: number; }': activityTempl
ate, interval
```

This error message tells us that our object is missing two keys: `activityTemplate`, and `interval`. If we look up the definition of `ActivityRecurrenceGoal` in the scheduling documentation, we will see that it does indeed need an activity template and an interval. Let's add those:

```
export default (): Goal => {
  return Goal.ActivityRecurrenceGoal({
    activityTemplate: null,
    interval: 24 * 60 * 60 * 1000 * 1000 // 24 hours in microseconds
  })
}
```

Now, we just need to finish specifying the **activityTemplate**. Start by typing `ActivityTemplates.` (note the period), and select an activity type. Provide your activity an object with the arguments that that activity takes. Once the editor is no longer underlining your code, hit `Schedule & Analyze` and observe that your new activities were added to the plan.

Specifying the order of goals

The Aerie scheduler accepts a list of goals, and tries to satisfy them one by one by adding activities to your plan. We refer to this list of goals as a **scheduling specification**. Currently, Aerie creates one scheduling specification per plan. A goal's **priority** is simply a number reflecting that goal's position in the scheduling specification. The first goal will always have priority `0`, and the n-th goal will always have priority `n - 1`.

Scheduling DSL Documentation

Here you will find the full set of features in the scheduling DSL.

ActivityTemplate

An ActivityTemplate specifies the type of an activity, as well as the arguments it should be given. ActivityTemplates are generated for each mission model. You can get the full list of ActivityTemplates by typing `ActivityTemplates.` (note the period) into the scheduling goal editor, and viewing the auto-complete options.

Goal Types

Activity Recurrence Goal

The Activity Recurrence Goal (sometimes referred to as a "frequency goal") specifies that a certain activity should occur repeatedly throughout the plan, at some given interval.

Inputs

- **activityTemplate**: the description of the activity whose recurrence we're interested in.
- **interval**: a Duration of time specifying how often this activity must occur

Behavior

This interval is treated as an upper bound - so if the activity occurs more frequently, that is not considered a failure.

The scheduler will find places in the plan where the given activity has not occurred within the given interval, and it will place an instance of that activity there.

Note: The interval is measured between the *start times* of two activity instances. Neither the duration, nor the end time of the activity are examined by this goal.

Coexistence Goal (*coming soon*)

The Coexistence Goal specifies that a certain activity should occur once **for each** occurrence of some condition. Currently, that condition is limited to instances of a given activity type.

Inputs

- **forEach**: a string specifying the type of activity to look for in the plan
- **activityTemplate**: the description of the activity to insert after each activity identified by `forEach`

Behavior

The scheduler will find places in the plan where the `forEach` condition is true, check if there is already a matching activity there, and if not, it will insert a new instance using the given `activityTemplate`. The newly inserted instance's start time will be equal to the existing instance's end time.

Glossary

- [Aerie Domain](#)
- [Aerie Client User Interface](#)
- [Scheduling](#)
- [Simulation and Modeling](#)
- [Technologies](#)

Aerie Domain

Aerie deployment: The term 'adaptation' sometimes means an integrated system using some third-party elements. We refer to a particular configuration of the Aerie system as an 'Aerie Deployment', in the context of a broader ground data system (GDS) deployment.

Banananation: the tongue-in-cheek named toy mission model used by Aerie to provide extremely simple examples of mission modelling capabilities.

Planner: A person responsible for deciding how to concretely achieve a set of mission objectives over the course of a span of time, formalized as a plan, with the expectation that this plan will be executed by other mission elements, including (and especially) by a spacecraft's onboard system. The planner's concern is with achieving those objectives while balancing reward against risk. Planners iteratively perform scheduling and simulation in a feedback loop to refine their plans. Automated schedulers may themselves utilize simulation artifacts to inform their decisions.

Schedule:

Aerie Client User Interface

Guides-horz/vert:

Layers:

Rows:

Timelines:

View:

Scheduling

Rule/Goal:

Simulation and Modeling

Activity: is a modeled unit of mission system behavior to be utilized for simulations during planning. Activities emit events which can have various effects on modeled resources, and these effects can be modulated through input parameters.

Argument: a value given to a function or assigned to an Activity Parameter.

Call: is a function in the Merlin simulation modeling API. `Call()` executes a provided task and blocks until that task has completed. Typically `Call()` is used by a mission modeler who wants to execute an activity and wait for it's completion before continuing execution of their modeling code. `Call()` has a few function signatures so that it can be used to call an activity type and arguments or a Java lambda/Runnable.

Cell: allows a mission model to express **time-dependent state** in a way that can be tracked and managed by the host system.

Constraint: an expression built up with the [Aerie constraints eDSL](#), which evaluates to a set of windows during which the condition(s) defined by the expression is true or false.

Decomposition: A method for modeling the behavior of an activity (root activity) by composing a set of activities. Each composed activity describes some smaller aspect of behavior. The root activity orchestrates the execution of each child activity by interleaving function calls to `spawn()`, `delay()`, and `call()`. The goal of decomposition is to allow mission modelers to modularize their activity modeling code and to provide greater visibility into the simulated behavior of an activity. For example, consider an activity which models taking an observation with a particular spacecraft instrument. The process of taking an observation may include the distinct phases of instrument startup, take observation, and instrument shutdown. The mission modeler can decide to modularize their modeling and use decomposition to model each of the three

phases separately and then compose them with a root activity called "observation" which orchestrates the execution of each of the three activities.

Delay: is a function in the Merlin simulation modeling API. A modeler can delay (pause) an activity's execution during simulation effectively modeling some passage of simulated time, before resuming further modeling.

Dynamics:

Effect/Event:

Mission model: Aerie has ceased using the term 'adaptation'; Aerie uses the term 'mission model' to denote the modeling code written in Java, packaged as a JAR, and consulted by the Merlin simulator. It is more specific to the purpose of the code, not overloaded with other extant meanings, and better coheres with the modeling domain.

The term 'adaptation' means too many things in too many contexts. Take for example the term 'mission planning adaptation'. It is unclear whether the speaker refers solely to the APGen .aaf files, or also includes any modeling integrations, or includes further the integrated software deployment which produces resource profiles and associated analyses. The term is also very JPL-centric; different terms are used in the wider domains of planning, modeling, and simulation.

Parameter: a named variable (and data type) in a function description or Activity definition, utilized within the function/Activity definition.

Profile: the **time-dependent evolution** behavior of a resource.

Register: is a model of a resource which can have its value set and this value remains until it is set with a different value. The term register here is used to clearly indicate the semantics of this resource; the semantics of setting a memory register.

Resource: expresses the **time-dependent evolution** of quantities of interest to the mission.

Resources Types:

State: the value of a resource/variable at an instant in time. E.g. "The state of the radioMode resource is ON"

Simulation: Plan simulation is the act of predicting the usage and behavior of mission resources, over time, under the influence of planned activities. Put differently, simulation is an analysis of the effects of a plan upon mission resources.

Simulation Results: the output of a Merlin simulation run. The simulation results include, simulated activity spans, their arguments (root activities and those arising from decomposition), resource profiles, and events.

Task: allows a mission model to describe **time-dependent processes** that affect mission state.

Window:

Technologies

API: Application Programming Interface.

AWS: [Amazon Web Services](#). Provides on-demand cloud computing platforms and API.

CAM: Common Access Manager. A [NASA AMMOS](#) utility which provides application layer access control capabilities, including single sign-on (SSO), federation, authorization management, authorization checking & enforcement, identity data retrieval, and associated logging

Docker: [Docker](#) is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers.

GraphQL: is an open-source data query and manipulation language for APIs, and a runtime for fulfilling queries with existing data. [graphql.org](#)

Hasura: a GraphQL schema compiler and GraphQL API server. As a compiler, Hasura parses a PostgreSQL schema and generates a GraphQL schema defining a data graph of Queries, Mutation, and Subscriptions. Aerie utilizes a Hasura component to expose select database tables as the Aerie GraphQL API.

JAR: A [JAR](#) is a package file format typically used to aggregate many Java class files and associated metadata and resources into one file for distribution. JAR files are archive files that include a Java-specific manifest file. They are built on the ZIP format and typically have a .jar file extension.

JUnit: is a unit testing framework for the Java programming language. [junit.org](#)

JVM: A [Java virtual machine](#) is a virtual machine that enables a computer to run Java programs as well as programs written in other languages

that are also compiled to Java bytecode.

PostgreSQL: [PostgreSQL](#), also known as Postgres, is a free and open-source relational database management system emphasizing extensibility and SQL compliance.

SPICE: NASA's Navigation and Ancillary Information Facility (NAIF) offers NASA flight projects and NASA funded researchers the "SPICE" observation geometry information system to assist scientists in planning and interpreting scientific observations from space-based instruments aboard robotic planetary spacecraft. In the Merlin context SPICE is most commonly incorporated to a mission model as a Java library. The library is configured with data files called "kernels" and then a mission model will query the library for an assortment of mission specific geometric data.
[SPICE](#)

Worker: Aerie provides a multi-tenancy capability so that many users can run simulations concurrently. Simulation multi-tenancy is achieved by configuring Aerie to launch multiple simulation worker containers. Each simulation worker can execute a sand-boxed simulation run.

Command Expansion Guide

Introduction

This guide explains how to use the command expansion service with the latest version of Aerie.

The command expansion service allows you to generate commands from activity information within an Aerie Plan. With this service, you can create custom logic that can be assigned to an activity_type. This custom logic will be used by Aerie to generate the commands. Your custom logic will require some pre-generated libraries provided by the command expansion service. An IDE will be used to write your custom logic.

Prerequisites

- An AMPCS Command Dictionary
- Mission Model

Steps

1. Upload a Command Dictionary and retrieve the command_library.ts file
2. Select an activity_type to generate the activity_library.ts file
3. Setup an IDE to start writing custom logic
4. Upload the custom logic
5. Create an Expansion Set
6. Expand the plan with the Expansion Set
7. Retrieve the generated commands

Continue to [Upload and Generate Command Typescript Library](#)

Upload and Generate Command Typescript Library

We recommend using the latest command dictionary as a starting point. This will allow you to use the most up-to-date commands that have been tested and validated. When you upload a command dictionary, an autogenerated `command_library` typescript file will be available to download. This library will be used later.

Upload

Uploading a command dictionary is very simple using GraphQL. In order to upload a command dictionary, login to the Aerie UI and navigate to the GraphQL Playground. From there, you will use our pre-defined mutation to upload the command dictionary. Run this GraphQL mutation below to upload the command dictionary.

```
mutation Upload(  
  $commandDictionaryString: String!  
) {  
  uploadDictionary(dictionary: $commandDictionaryString) {  
    id  
  }  
}
```

Download Command Library

A successful upload will return an `id` used to retrieve the library. Running the GraphQL query below will retrieve the typescript library.

1. Run this GraphQL command below

```
query GetCommandTypescript(  
  $commandDictionaryId: String!  
) {  
  getCommandTypeScript(commandDictionaryId: $commandDictionaryId) {  
    typescript  
  }  
}
```

2. Save this to a file named `<command_library>.ts`

Continue to [Generate Activity Typescript Library](#)

Generate Activity Typescript Library

In order to write an author expansion logic, you will need to know which `activity_types` to use. These `activity_types` are defined in the mission model.

Upon the selection of an `activity_type`, the command expansion service will generate a typescript library file containing all parameters that can be used when developing your logic. Below is the GraphQL request used to retrieve the library.

1. Run the graphQL query below

```
query GetActivityTypescript(  
  $activityTypeName: String!  
  $missionModelId: Int!  
) {  
  getActivityTypeScript(  
    activityTypeName: $activityTypeName  
    missionModelId: $missionModelId  
  ) {  
    typescript  
  }  
}
```

2. Save to a file <activity_type_library>.ts ex. BakeBananaBread_library.ts

Continue to [Setup IDE and Writing Expansion Logic](#)

Setup IDE and Writing Expansion Logic

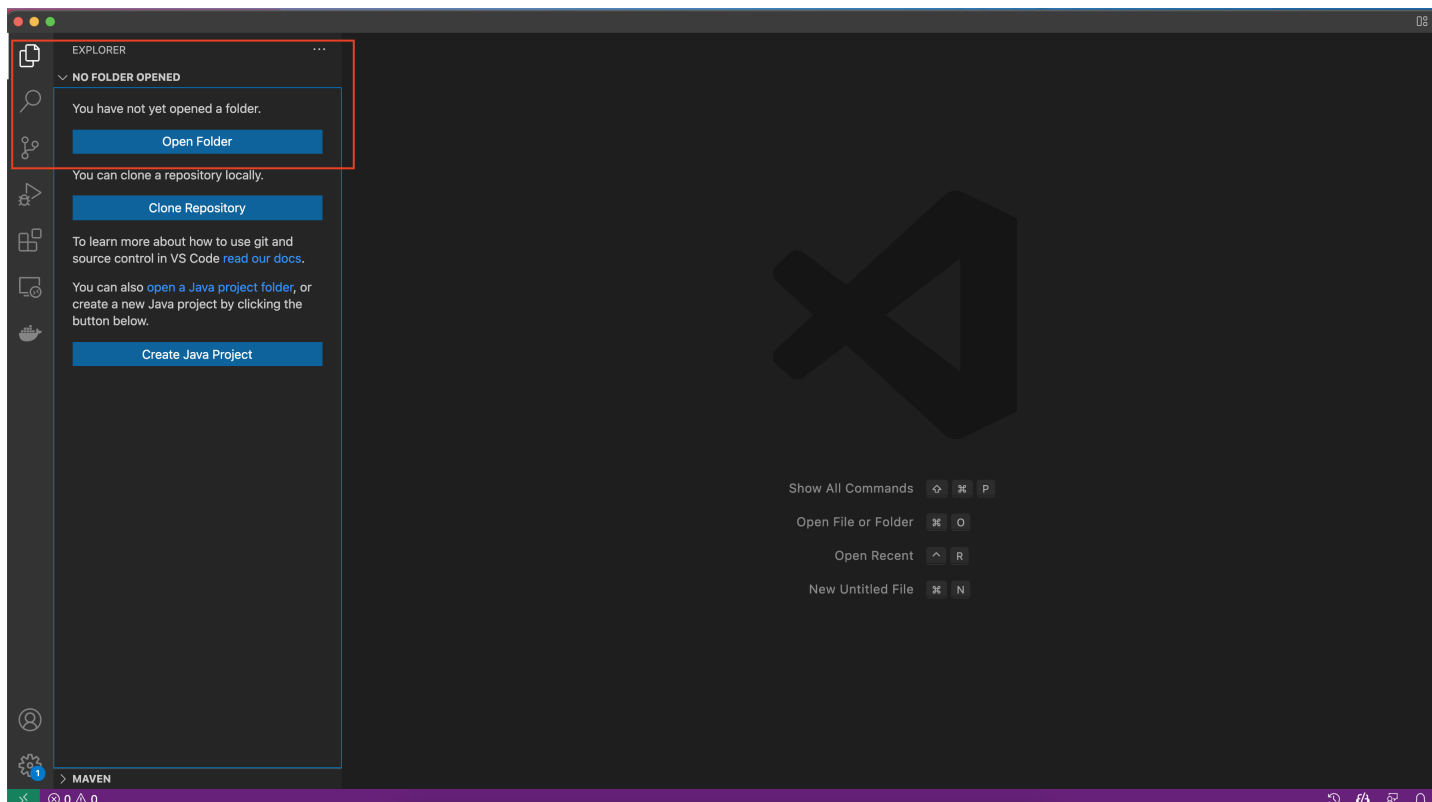
Now it is time to start writing the expansion logic for the activity type. The recommended IDE is Visual Studio Code as we are looking at embedding this IDE as a web editor as a future service. Another option is IntelliJ but you will need the paid Ultimate version to use the typescript plugin.

Recommended IDE:

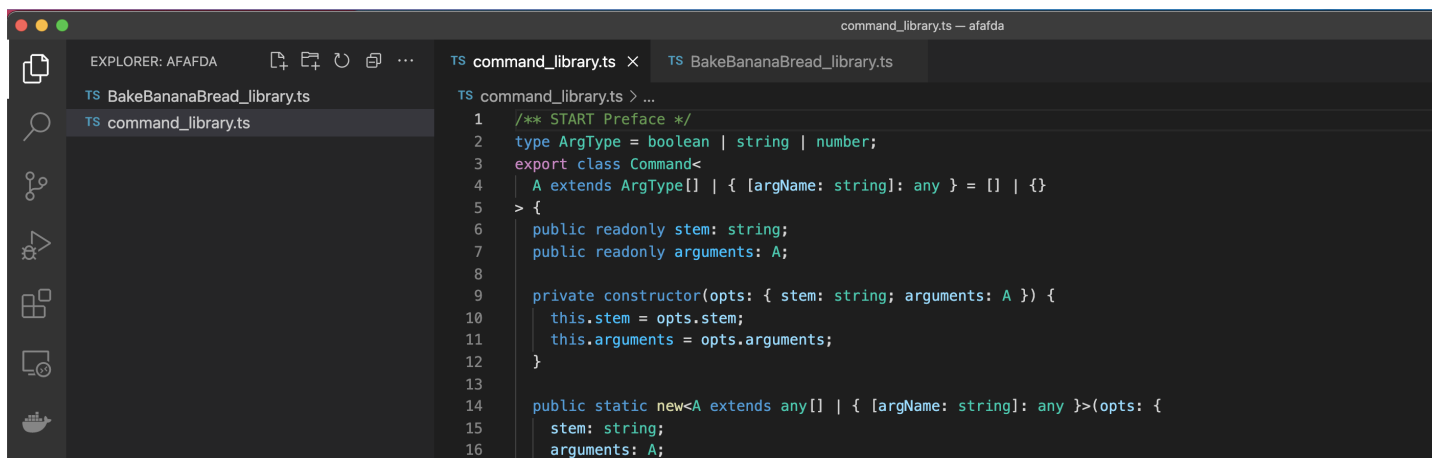
- Visual Studio Code: <https://code.visualstudio.com/>
- IntelliJ: <https://www.jetbrains.com/idea/>

Setup:

1. Open a new project folder. You can create a new folder as your workspace



2. Drag you <activity_library>.ts and <command_library>.ts into you project



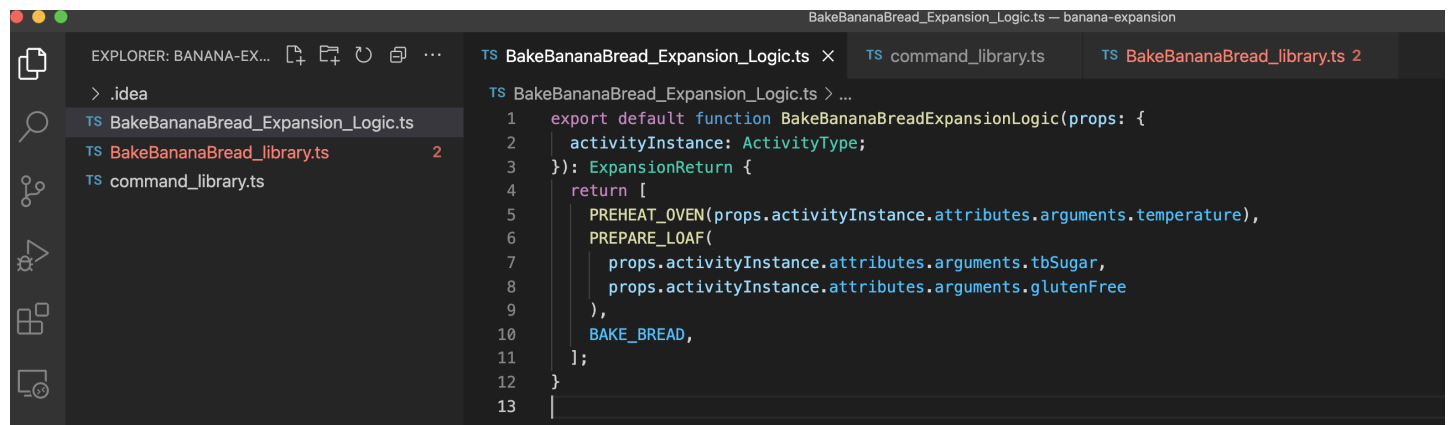
3. Create an expansion logic file. You can name it whatever you like. **ex. BakeBananaBreadExpansionLogic.ts**
4. Add the following below which is a boilerplate template


```
// This is the entry point for the expansion - name it whatever you want, it just has to be the default export
export default function CommandExpansion(
  // Everything related to this activity
  props: {
    activityInstance: ActivityType;
  }
): ExpansionReturn {
  // Commands are fully strongly typed, and intellisense in the authoring editor will
  // guide users to write correct expansions
  return [
    // COMMAND_A(arg1)
    Subroutine(),
    // Commands without any arguments omit the ()
    // COMMAND_C
  ];
}

// You can break it into smaller logical/reusable chunks to make comprehension easier
function Subroutine() {
  return [
    //COMMAND_B(arg1,arg2)
  ];
}
```

Writing

As you start typing in the editor, notice that each command is validated and type checked. This means as you write out your logic, the editor will ensure all commands are valid, and that each argument meets the requirements. When you are happy with your logic save your works.



Submit

Continue to [Submit Expansion Logic](#)

Submit Expansion Logic

Upload

Once you've defined your expansion logic you need to upload it back to the commanding server and link it to the `activity_type` you wrote it for. Below you can use the GraphQL to upload the expansion logic. Use the GraphQL to upload the expansion logic file for the activity_type

```
mutation UploadExpansionLogic(
  $activityTypeName: String!
  $expansionLogic: String!
) {
  addCommandExpansionTypeScript(
    activityTypeName: $activityTypeName
    expansionLogic: $expansionLogic
  ) {
    id
  }
}
```

Retrieve

If you would like to retrieve an expansion logic for modification use this GraphQL API below:

```
query GetExpansionLogic(
  $expansionRuleId: Int!
) {
  expansion_rule(where: {id: {_eq: $expansionRuleId}}) {
    expansion_logic
    activity_type
    id
  }
}
```

Retrieve a specific activity_type's various expansion logics:

```
query GetExpansionLogic(
  $activityType: String!
) {
  expansion_rule(where: {activity_type: {_eq: $activityType}}) {
    expansion_logic
    activity_type
    id
  }
}
```

Continue to [Create an Expansion Set](#)

Create an Expansion Set

Introduction

Think of an "expansion_set" as a way of grouping expansion logic to different versions of mission models or command dictionaries.

But why the need for an "expansion_set"?

For example, say you have a mission model and command dictionary version 1. You have written expansion logic that uses commands from the command dictionary. Later the command dictionary or mission model is updated to version 2. If your original expansion logic was created to only be compatible with the previous version of the command dictionary or mission model, you need to go through and rewrite/update your existing logic for all the activity_types.

An alternative is that you can use the "expansion_set". This will allow you to copy over a set of expansion logic into a new "expansion_set" without manually changing anything in your code. Some expansion logic might not be valid anymore but the user can decide which expansion logic to copy over to the new set containing the updated command dictionary or mission model.

A future feature will inform users of which expansion rules are compatible/incompatible for the selected mission model and command dictionary, making the migration of expansion_sets easier.

The below GraphQL creates an expansion set for a given command dictionary and mission model which includes the provided expansions.

```
mutation CreateExpansionSet(  
  $commandDictionaryId: Int!  
  $missionModelId: Int!  
  $expansionIds: [Int!]!  
) {  
  createExpansionSet(  
    commandDictionaryId: $commandDictionaryId  
    missionModelId: $missionModelId  
    expansionIds: $expansionIds  
  ) {  
    id  
  }  
}
```

Ex. 1

That was a lot to take in. Let's create our own expansion set and tie everything together. Below is an example GraphQL to create an "expansion_set". With the 4 expansion logic defined we want to group them with command dictionary and mission model version 1.

```
{  
  "commandDictionaryId": 1,  
  "missionModelId": 1,  
  "expansionIds": [1,2,4,9]  
}
```

Ex.2

Now let us say the mission model has changed and out of the 4 activity_types defined in the mission model we are dropping one of them. We will only copy over the expansion logic that is valid in this new set.

```
{  
  "commandDictionaryId": 1,  
  "missionModelId": 2,  
  "expansionIds": [1,2,9]  
}
```

List Expansion Sets

Below is a way to list the created expansion sets in GraphQL:

```

query GetAllExpansionSets {
  expansion_set {
    id
    command_dictionary {
      version
    }
    mission_model {
      id
    }
    expansion_rules {
      activity_type
      id
    }
  }
}

```

Get all `expansion_set`'s for a particular `mission_model` or `command dictionary` :

```

query GetSpecificExpansionSet(
  $commandDictionaryId: Int!
  $missionModelId: Int!
) {
  expansion_set(where: {command_dictionary: {id: {_eq: $commandDictionaryId}}, mission_model_id: {_eq: $missionModelId}}) {
    id
    expansion_rules {
      activity_type
      id
    }
  }
}

```

Continue to [Expand the Plan](#)

Expand the Plan

Introduction

Now for the exciting part! We can finally expand the plan into commands. To do that, we'll need to set up a couple of things with your plan.

1. Add an `activity_instances` to the plan which has expansion logic defined in the `expansion_set`
2. Run a simulation on the plan

Prerequisite

At this time you will have to use the Hasura Action API in order to expand the plan into commands. You will have to retrieve some information needed to expand via the API. The following two values are needed:

- `expansionSetID`
- `simulationDatasetId`

Below is the query to retrieve an `expansionSetID`:

```
query GetExpansionSet(
  $commandDictionaryId: Int!
  $missionModelId: Int!
) {
  expansion_set(where: { mission_model_id: { _eq: $missionModelId }, command_dictionary: { id: { _eq: $commandDictionaryId } } }) {
    id
  }
}
```

id : 1

Below is the query to retrieve the `simulationDatasetId`:

```
query GetSimulationDatasetId(
  $planId: Int!
) {
  simulation(where: { plan_id: { _eq: $planId } }, order_by: { dataset: { id: desc } }, limit: 1) {
    dataset {
      id
    }
  }
}
```

id : 5

Expanding

Below is an example of the Hasura Action you can use to expand a plan:

```
mutation ExpandPlan(
  $expansionSetId: Int!
  $simulationDatasetId: Int!
) {
  expandAllActivities(expansionSetId: $expansionSetId, simulationDatasetId: $simulationDatasetId) {
    id
    expandedActivityInstances {
      commands {
        id
      }
      errors {
        message
      }
    }
  }
}
```

Continue to [Get Expanded Commands](#)

Get Expanded Commands

When the plan has been expanded you will be able to retrieve the generated commands from the expansion run. Each `activity_instance` will have commands and any errors generated. If there are no commands and errors listed then an `activity_instance` didn't have any expansion logic defined. Below is an example of how to retrieve all the expanded commands from an expansion run or a specific command from `activity_instance`.

```
query GetAllExpandedCommandsForExpansionRun(
  $expansionRunId: Int!
) {
  activity_instance_commands(where: {expansion_run: {id: {_eq: $expansionRunId}}}) {
    activity_instance_id
    commands
    errors
  }
}
```

```
query GetCommandsForExpansionRunAndActivityInstance(
  $expansionRunId: Int!
  $activityInstanceId: Int!
) {
  activity_instance_commands(where: {activity_instance_id: {_eq: $activityInstanceId}, expansion_run: {id: {_eq: $expansionRunId}}})
  {
    commands
    errors
  }
}
```

Continue to [Create Sequence](#)

Create Sequence

Sequence creations starts with declaring a new sequence on a simulation dataset. Because of the complexities with activities in the plans regenerating on simulation, sequences are tied to a specific simulation run rather than a plan (future work will explore addressing this).

The API call to do this is a simple mutation

```
mutation CreateSequence(  
  $seqId: String!  
  $simulationDatasetId: Int!  
) {  
  insert_sequence_one(object: {  
    simulation_dataset_id: $simulationDatasetId,  
    seq_id: $seqId,  
    metadata: {},  
  }) {  
    seq_id  
  }  
}
```

Where the seqId is the id you want assigned to the sequence and simulationDatasetId is the id of the simulation dataset for the simulation you wish to create the sequence for.

Note: You can only have one of any given seqId for a given simulationDataset

Continue to [Link Simulated Activity to Sequence](#)

Link Simulated Activity to Sequence

Once a sequence is created, you can link simulated activities to the sequence. Linking simulated activities to a sequence specifies that that activity's commands should be included in the sequence. Note that a simulated activity is NOT the same as an activity inserted in the plan, but the result of simulation on that plan and the ids are NOT interchangeable. You can retrieve the simulated activities for a simulation dataset to determine which to associated with the following query

```
query GetSimulatedActivities(  
  $simulationDatasetId: Int!  
) {  
  simulated_activity(where: { simulation_dataset_id: {_eq: $simulationDatasetId } }) {  
    id  
    start_offset  
    duration  
    activity_type_name  
    attributes  
  }  
}
```

Linking an activity to a sequence is then a simple API call using the id from above

```
mutation LinkSimulatedActivityToSequence(  
  $seqId: String!  
  $simulationDatasetId: Int!  
  $simulatedActivityId: Int!  
) {  
  insert_sequence_to_simulated_activity_one(object: {  
    seq_id: $seqId  
    simulated_activity_id: $simulatedActivityId  
    simulation_dataset_id: $simulationDatasetId  
  }) {  
    seq_id  
  }  
}
```

Note: A simulated activity can only be associated with one sequence

Continue to [Retrieve SeqJson Serialization of Sequence](#)

Retrieve SeqJson Serialization of Sequence

Once a sequence is associated with activity instances, you can retrieve the SeqJson serialization of the sequence. This serialization is a simple concatenation of the commands associated with the activity instances in time-order. Any activities that do not have any associated commands (not yet expanded, no expansion associated, or some error during expansion) will not have any commands included in the resulting serialization. Additionally, any errors encountered during expansion will be included in the SeqJson as a command with the commands stem `$$ERROR$$`.

Retrieving the SeqJson serialization of a sequence is a simple API call

```
query GetSequenceSeqJson(
  $seqId: String!
  $simulationDatasetId: Int!
) {
  getSequenceSeqJson(
    seqId: $seqId,
    simulationDatasetId: $simulationDatasetId
  ) {
    id
    metadata
    steps {
      type
      stem
      args
      time {
        tag
        type
      }
      metadata
    }
  }
}
```

User Guide

Overview

This document is a guide to how to make use of current Aerie capabilities. Aerie is a new software system to support the activity planning, sequencing and spacecraft analysis needs of missions. Aerie is being developed by the MPSA element of Multi-mission Ground System and Services (MGSS), a subsystem of AMMOS (Advanced Multi-mission Operations System). This guide will be updated as new features are added.

Aerie is a collection of loosely coupled services that support activity planning and sequencing needs of missions with modelling, simulation, scheduling and rule validation capabilities. Aerie will replace legacy MGSS tools including but not limited to APGEN, SEQGEN, MPS Editor, MPS Server, Sinc II / CTS and ULGEN. Aerie currently provides the following capabilities:

1. Merlin adaptation framework offering a subset of APGEN capabilities,
2. Merlin web GUI for activity planning,
3. Merlin command line interface for activity planning, and
4. Falcon smart sequence editor GUI.

Prerequisites

An adaptation is software developed with the Merlin framework libraries that models spacecraft behavior while performing a set of activities over a plan duration. Merlin adaptations can simulate a variety of States that are perturbed by executed activities, and governed by system models. Merlin plans describe a scheduled collection of activity instances with specified parameters. This user guide describes how to upload an existing adaptation .JAR file, how to create plans with that adaptation, and how to edit and simulate those plans. For users to complete these steps, they should be able to develop and compile an adaptation and have access to an Aerie installation. For details of how to create adaptations with Aerie refer to the [Mission Modeler Guide](#). For information on how to install Aerie services refer to the [Product Guide](#).

Merlin Activity Plans

Aerie provides an API to manage a database of plans. The database of plans may be queried for a list of all plans, and new plans may be added to the repository. Existing plans may be retrieved in full, replaced in full or in part, or deleted in full. The list of activities in a plan may be appended to (by creating a new activity) and retrieved in full. Individual activities in a plan may be retrieved in full, replaced in full or in part, and deleted in full. How to execute queries and mutations against the Aerie API is found in the [GraphQL software interface specification](#).

Operations on plans are validated to ensure consistency with the mission model-specific activity model with which they are associated. Stored plans shall contain activities whose parameter names and types are defined by the associated activity type.

Aerie GraphQL API

Purpose

This document describes the Aerie GraphQL API, software interface provided by the Aerie 0.10.0 release.

Terminology and Notation

No special notation is used in this document. See Appendix A for complete GraphQL Scheme definition.

Interface Overview

GraphQL is not a programming language capable of arbitrary computation, but is instead a language used to query application servers that have capabilities defined by the GraphQL specification. Clients use the GraphQL query language to make requests to a GraphQL service.

The three key GraphQL terms are;

- **Schema:** a type system which defines the data graph. The schema is the data sub-space over which queries can be defined.
- **Query:** a JSON-like read-only operation to retrieve data from a GraphQL service.
- **Mutation:** An explicit operation to effect server side data mutation.

A REST API architecture defines a particular URL endpoint for each "resources". In contrast, GraphQL's conceptual model is an entity graph. As a result, entities in GraphQL are not identified by URL endpoints and GraphQL is not a REST architecture API. Instead, a GraphQL server operates on a single endpoint, and all GraphQL requests for a given service are directed at this endpoint. Queries are constructed using the query language and then submitted as part of a HTTP request (either GET or POST).

The schema (graph) defines nodes and how they connect/relate to one another. A client composes a query specifying the fields to retrieve (or mutation for fields to create/update). A client develops their query/mutation with reference to the exposed GraphQL schema. As a result, a client develops custom queries/mutations targeted to its own use cases to fetch only the needed data from the API. In many cases this may reduce latency and increase performance by limiting client side data manipulation/filtering. For example, with a REST API there may be significant client side overhead and request latency when querying for an entire plan and then filtering the plan for the specific information of concern (and the work of adding filter fields as parameters to the end point). In contrast the GraphQL API allows the client to request only the fields of the plan data structure needed to satisfy the client's use case.

The Aerie GraphQL API is versioned with Aerie releases. However, a GraphQL based API gives greater flexibility to clients and Aerie when evolving the API. Adding fields and data to the schema does not affect existing queries. A client must specify the fields that make up their query. The additional fields in the graph simply play no part in the client's composed query. As a result, additions to the API do not require updates on the client side. However, clients do need to deal with schema changes when fields are removed or type definitions are evolved. Furthermore, GraphQL makes possible per-field auditing of the frequency and combinations with which certain fields are referenced by a client's queries and mutations. This provides Aerie developers with evidence of usage frequency which informs decision processes regarding deprecating or updating fields in the schema.

GraphQL Query Fundamentals

A round trip usage of the API consists of the following three steps:

1. Composing a request (query or mutation)
2. Submitting the request via POST
3. Receiving the result as JSON

POST request

A standard GraphQL HTTP POST request should use the `application/json` content type, and include a JSON-encoded body of the following form:

```
{
  "query": "...",
  "operationName": "...",
  "variables": { "myVariable": "someValue", ... }
}
```

`operationName` and `variables` are optional fields. The `operationName` field is only required if multiple operations are present in the query.

Response

Regardless of the method by which the query and variables are sent, the response is returned in the body of the request in JSON format. A query's results may include some data and some errors, and those are returned in a JSON object of the form:

```
{
  "data": { ... },
  "errors": [ ... ]
}
```

If there were no errors returned, the `"errors"` field is not present in the response. If no data is returned, the `"data"` field is only included if the error occurred during execution.

GraphQL Clients

Since a GraphQL API has more underlying structure than a REST API, there are a range of methods by which a client application may choose to interact with the API. A simple usage could use the `curl` command line tool, whereas a full featured web application may integrate a more powerful client library like [Apollo Client](#) or [Relay](#) which automatically handle query building, batching and caching.

Command Line

One may build and send a query or mutation via any means that enable an HTTP POST request to be made against the API. For example, this can be easily done using the command line tool [Graphqlurl](#).

GraphQL Playground

The GraphQL API is described by a schema of the data graph. One can view the schema of the installed version of Aerie at `https://<your_domain>:27184`. The GraphQL playground allows one to compose and test queries. The playground also provides the functionality to export the composed query as a fully formed `curl` command string.

Playground Authentication

In order to use queries in the playground, first you need to authenticate against CAM to get an authorization token. In the `QUERY VARIABLES` section of the playground, first define your JPL username and password:

```
{
  "username": "YOUR_JPL_USERNAME",
  "password": "YOUR_JPL_PASSWORD"
}
```

Then you can use the Aerie utility HTTP API to obtain your `ssoCookieValue` from `https://<your_domain>:9000`.

Next, add the `authorization` header to the `HTTP HEADERS` section :

```
{
  "authorization": "SSO_COOKIE_VALUE_HERE"
}
```

You should now be able to make queries using the playground. For example try querying for all the plan ids:

```
query Plans {
  plan {
    id
  }
}
```

Note your CAM server configuration determines how long your authentication token is valid. If your session expires you will have to re-authenticate and place the new token in the `authorization` header.

Browser Developer Console

Requests can also be tested from the browser. Navigating to `https://<your_domain>:9000/playground`, open a developer console, and paste in:

```

fetch('', {
  method: 'POST',
  headers: {
    'Authorization': 'SSO_COOKIE_VALUE_HERE',
    'Content-Type': 'application/json',
    'Accept': 'application/json',
  },
  body: JSON.stringify({query: "{plan{id}}"})
})
.then(r => r.json())
.then(data => console.log('data returned:', data));

```

The data returned is logged in the console as:

```

{
  "data": {
    "plans": [
      {"id": "5f763f2b513fec1988930f03"},
      {"id": "5f7f5d0218c85a5533f1dc4b"}
    ]
  }
}

```

This JavaScript can then be used as a hard-coded query within a client tool/script. For more complex and dynamic interactions with the Aerie API it is recommended to use a GraphQL client library.

Client Libraries

When developing a full featured application that requires integration with the Aerie API it is advisable that the tool make use of one of the many powerful GraphQL client libraries like Apollo Client or Relay. These libraries provide an application functionality to manage both local and remote data, automatically handle batching, and caching.

In general, it will take more time to set up a GraphQL client. However, when building an Aerie integrated application, a client library offers significant time savings as the features of the application grow. One might choose to begin using HTTP requests as the client's API integration mechanism and later switch to a client library as the application becomes more complex.

GraphQL clients exist for the following programming languages;

- C# / .NET
- Clojurescript
- Elm
- Flutter
- Go
- Java / Android
- JavaScript
- Julia
- Kotlin
- Swift / Objective-C iOS
- Python
- R

A full description of these clients is found at <https://graphql.org/code/#graphql-clients>

Aerie GraphQL API

Schema

The schema is too large to include here and in Aerie's automatic documentation generation (it creates a 300 pg. doc). The schema for your Aerie installation can be viewed at https://<your_domain>:9000/playground or the current Aerie release can be seen at <https://aerie-staging.jpl.nasa.gov:9000/playground>

Usage

Aerie employs the Hasura GraphQL (<https://hasura.io/>) engine to generate the Aerie GraphQL API. It is important to understand the significance and power of a data graph based API. A small primer of the GraphQL syntax can be found at <https://graphql.org/learn/schema/>

Query / Subscription

- Controlling Queries: <https://hasura.io/docs/latest/graphql/core/databases/postgres/queries/index.html>
- Controlling Subscriptions: <https://hasura.io/docs/latest/graphql/core/databases/postgres/subscriptions/index.html>
- Query/Subscription Syntax: <https://hasura.io/docs/latest/graphql/core/api-reference/graphql-api/query.html>

Mutations

- Controlling Mutations: <https://hasura.io/docs/latest/graphql/core/databases/postgres/mutations/index.html>
- Mutation Syntax: <https://hasura.io/docs/latest/graphql/core/api-reference/graphql-api/mutation.html>

Examples

The following queries are examples of what Aerie refers to as "canonical queries" because they map to commonly understood use cases and data structures within mission subsystems. When writing a GraphQL query, refer to the schema for all valid fields that one can specify in a particular query.

Query all plans

```
query {  
  plan {  
    id  
    name  
    start_time  
    duration  
    model_id  
  }  
}
```

Query a single plan

```
query {  
  plan_by_pk(id: 1) {  
    id  
    name  
    start_time  
    duration  
    model_id  
  }  
}
```

Query all activity instances from a plan

You can either query "plan_by_pk" for all activity instances from a single plan or query "plan" for all activity instances from all the plans

Returns activity instances

```
query {  
  plan_by_pk(id: 1) {  
    activities {  
      id  
      type  
      arguments  
    }  
  }  
}
```

Query mission model from a plan

Returns mission model for a particular plan

```

query {
  plan_by_pk(id: 1) {
    mission_model {
      id
      mission
      name
      owner
      version
    }
  }
}

```

Query activity types within a mission model from a plan

Returns a list of activity types. For each activity type, the name, parameter schema, which parameters are required (must be defined).

```

query {
  plan_by_pk(id: 1) {
    mission_model {
      activity_types {
        name
        parameters
        required_parameters
      }
    }
  }
}

```

Run simulation and query the result

Returns a list of resource profiles, constraint failures, and the simulated activities.

```

query {
  simulate(planId: 1) {
    reason
    results
  }
}

```

To read more about the response to the `simulate` action, see [Simulation Results](#)

Query for all resource types in a mission model

```

query {
  resourceTypes(missionModelId: 1) {
    name
    schema
  }
}

```

Creating activity instances

```

mutation {
  insert_activity(objects: [
    {arguments: { name: "peelDirection", value: "fromTip" }, plan_id: 4, start_offset: "1749:01:35.575", type: "PeelBanana"},
    {arguments: { name: "peelDirection", value: "fromTip" }, plan_id: 4, start_offset: "1750:01:35.575", type: "PeelBanana"}]) {
    returning {
      id
      start_offset
    }
  }
}

```

Query for activity effective arguments

The `getActivityEffectiveArguments` query returns a set of effective arguments given a set of required (and overridden) arguments.


```

query TestGetActivityEffectiveArguments {
  getActivityEffectiveArguments(missionModelId: 1, activityTypeName: "BakeBananaBread", activityArguments: {tbSugar: 1, glutenFree:
false}) {
    success
    errors
    arguments
  }
}

```

Resulting in:

```

{
  "data": {
    "getActivityEffectiveArguments": {
      "success": true,
      "arguments": {
        "temperature": 350,
        "tbSugar": 1,
        "glutenFree": false
      }
    }
  }
}

```

When a required argument is not provided, the returned JSON will indicate which argument is missing. For example:

With `examples/banananation`'s `BakeBananaBread`, where only the `temperature` parameter has a default value:

```

query TestGetActivityEffectiveArguments {
  getActivityEffectiveArguments(missionModelId: 1, activityTypeName: "BakeBananaBread", activityArguments: {}) {
    success
    errors
    arguments
  }
}

```

Resulting in:

```

{
  "data": {
    "getActivityEffectiveArguments": {
      "success": false,
      "errors": {
        "tbSugar": {
          "schema": {
            "type": "int"
          },
          "message": "Required argument for activity \"BakeBananaBread\" not provided: \"tbSugar\" of type ValueSchema.INT"
        },
        "glutenFree": {
          "schema": {
            "type": "boolean"
          },
          "message": "Required argument for activity \"BakeBananaBread\" not provided: \"glutenFree\" of type ValueSchema.BOOLEAN"
        }
      },
      "arguments": {
        "temperature": 350
      }
    }
  }
}

```

Query for mission model configuration effective arguments

The `getModelEffectiveArguments` returns the same structure as `getActivityEffectiveArguments`; a set of effective arguments given a set of required (and overridden) arguments.

For example, `examples/config-without-defaults`'s all required arguments:

```
query TestGetModelEffectiveArguments {
  getModelEffectiveArguments(missionModelId: 1, modelArguments: {}) {
    success
    errors
    arguments
  }
}
```

Results in:

```
{
  "data": {
    "getModelEffectiveArguments": {
      "success": false,
      "errors": {
        "a": {
          "schema": {
            "type": "int"
          },
          "message": "Required argument for configuration \"Configuration\" not provided: \"a\" of type ValueSchema.INT"
        },
        "b": {
          "schema": {
            "type": "real"
          },
          "message": "Required argument for configuration \"Configuration\" not provided: \"b\" of type ValueSchema.REAL"
        },
        "c": {
          "schema": {
            "type": "string"
          },
          "message": "Required argument for configuration \"Configuration\" not provided: \"c\" of type ValueSchema.STRING"
        }
      },
      "arguments": {}
    }
  }
}
```

Planning UI

The Aerie planning web application provides a graphical user interface to create, view, update and delete mission model and plans.

Uploading Mission Models

Mission models can be uploaded to Aerie via the UI. To navigate to the [mission model page](#), click the **Mission Models** icon on the on the left navigation bar. Once an **mission model JAR** is prepared, it can be uploaded to the mission model service with a name, version, mission and owner. The name and version must match (in case and form) the name and version specified in the mission model.

For example, if the mission model is defined in code as `@MissionModel(name="Banananation", version="0.0.1")`, then the name field must be entered as **Banananation** and the version as **0.0.1**. Once the mission model is uploaded it will be listed in the table shown in Figure 1. Mission Models can be deleted from this table using the context menu by right clicking on the mission model.

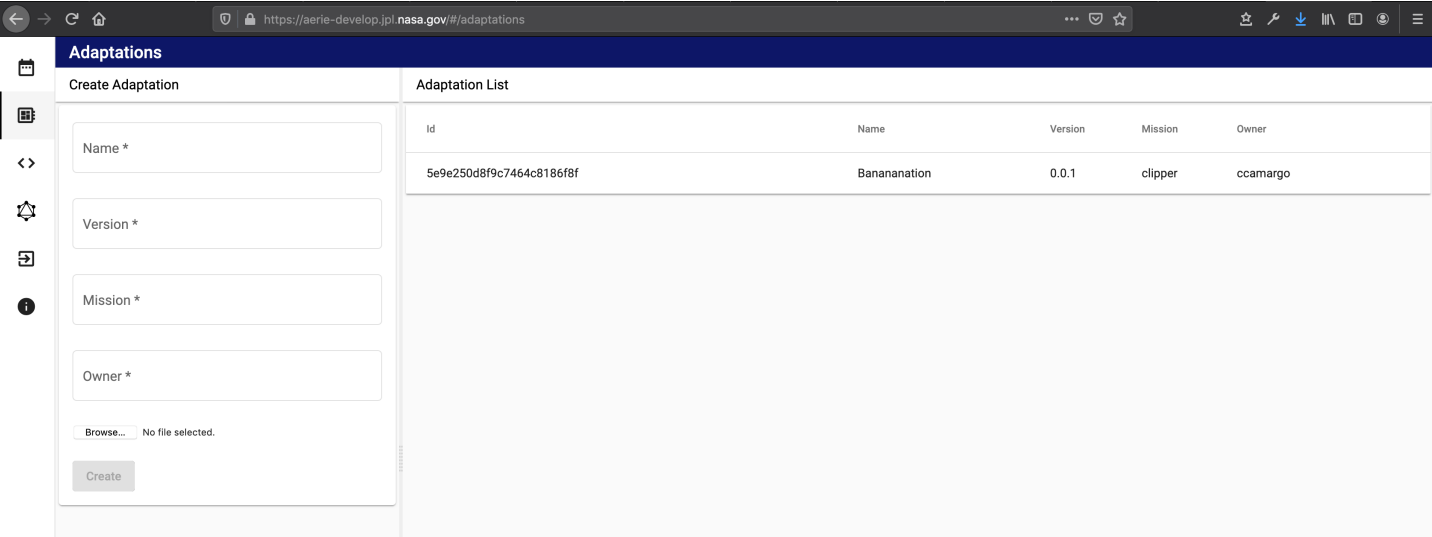


Figure 1: Upload mission model, and view existing mission model.

Creating Plans

To navigate to the [plans page](#), click the **Plans** icon on the left navigation bar. Users can use the left panel to create new plans associated with any mission model. A **start** and **end** date has to be specified to create a plan. Existing plans are listed in the table on the right. Use right click on the table to reveal a drop down menu to delete and view plans.

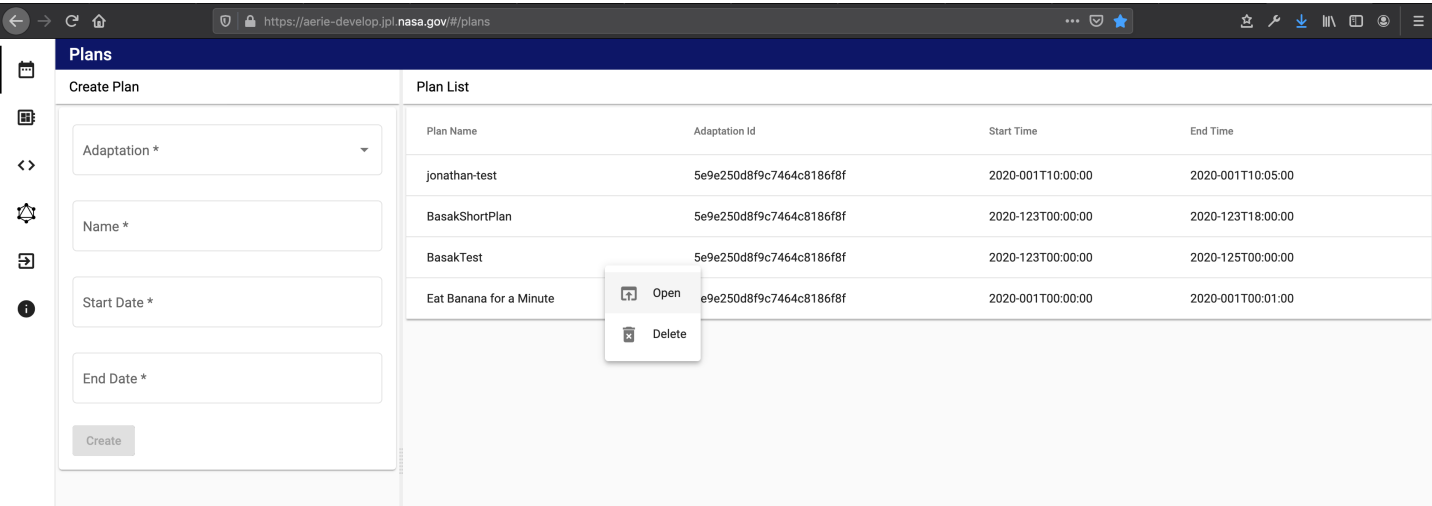


Figure 2: Create plans, and view existing plans.

Base simulation arguments may be supplied at plan creation time. For more information please refer to [Aerie Simulation Configuration UI](#) documentation.

View and Edit Plans

Once a user clicks on an existing plan, they can view contents, add/remove activity instances, and edit activity instance parameters. The plan view is split into the following default panels:

- Schedule Visualization
- Simulation Visualization
- Activity Instances Table
- Side drawer containing:
 - Activity Dictionary
 - Activity Instance Details

In the default side drawer the activity dictionary is displayed. Once a type or instance is selected, users can view details such as metadata and parameters by moving the arrow keys down. Activities can be dragged into the timeline from the activity dictionary. Once instances are added they will appear in the Activity Instances table panel.

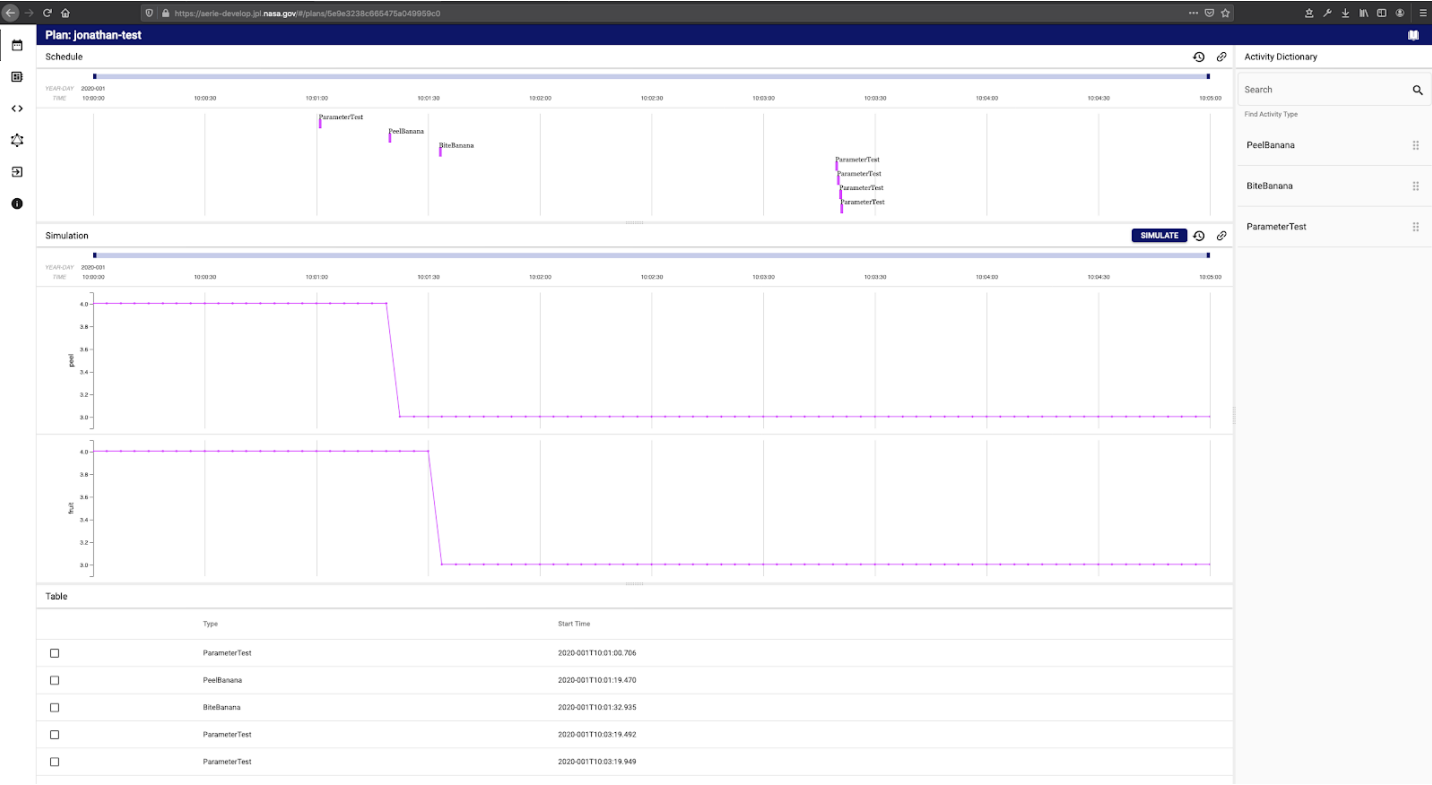


Figure 3: Default panels.

When a user clicks on an activity instance in the plan, the form to update activity parameters and start time will appear on the right drawer as shown in Figure 4. Users can use this form view to remove instances from the plan.

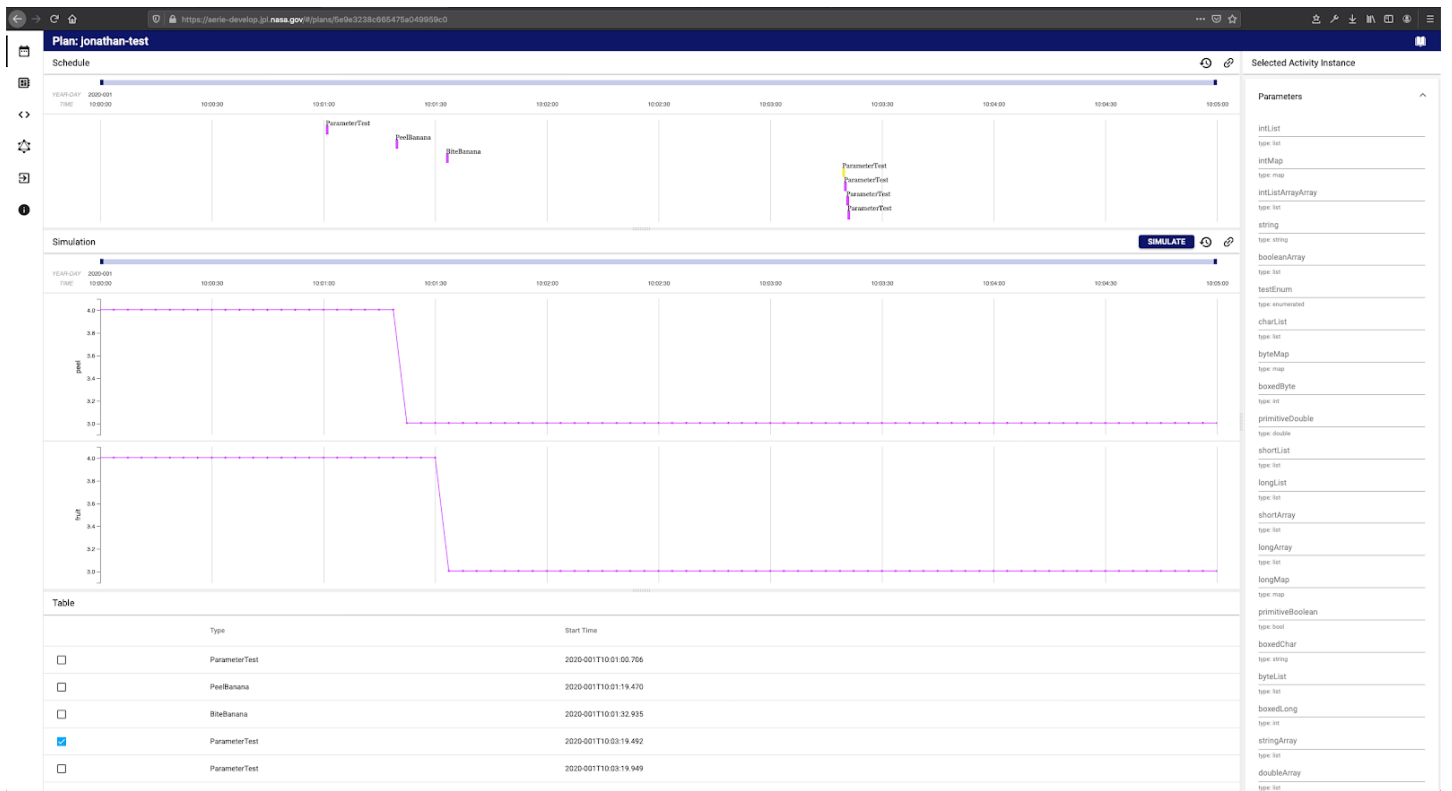


Figure 4: When an activity instance in plan is selected, its details will appear in the right drawer.

In the schedule and simulation elements, violations are shown as red regions in their respective bands as well as the corresponding time axis.

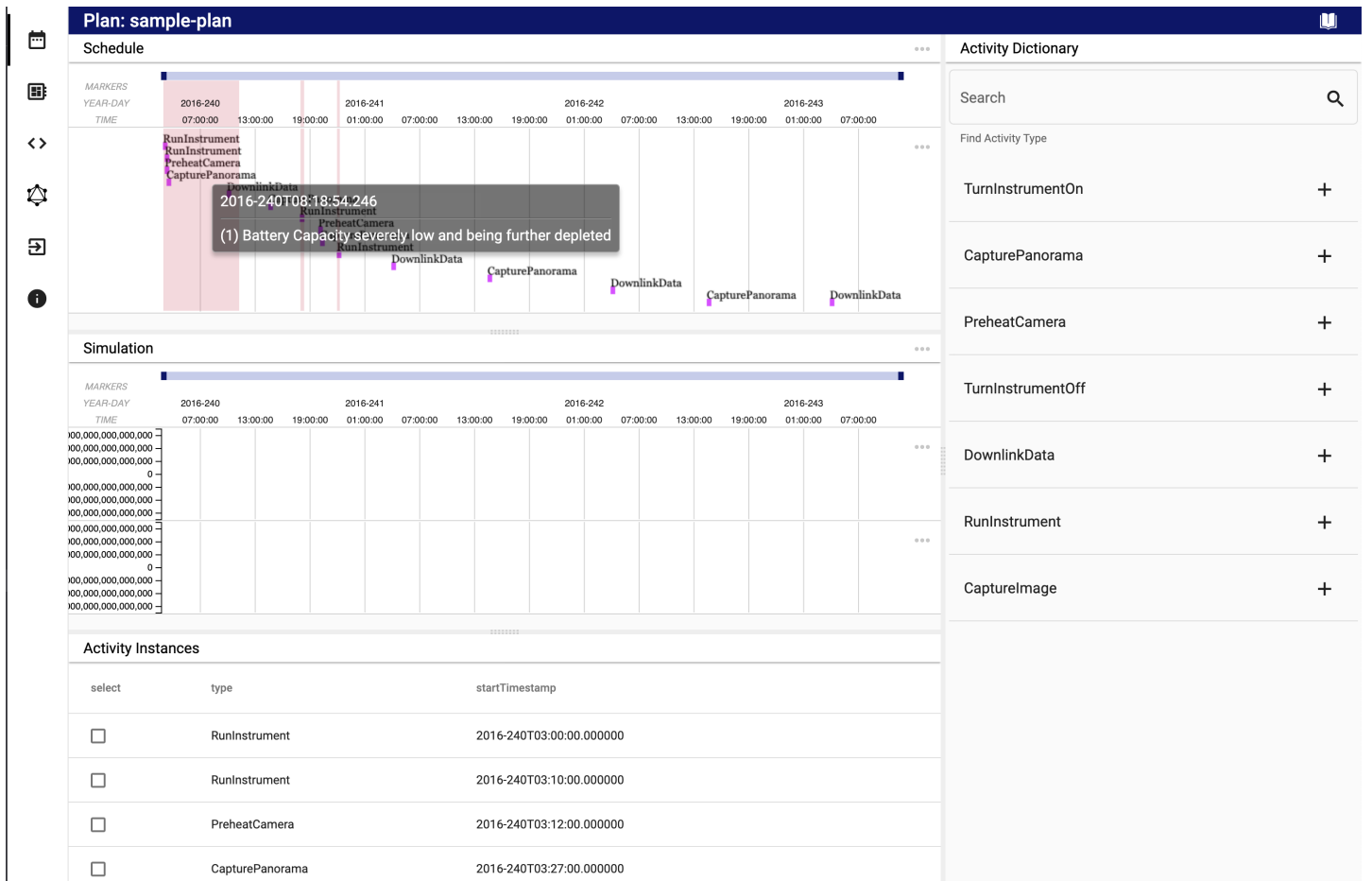
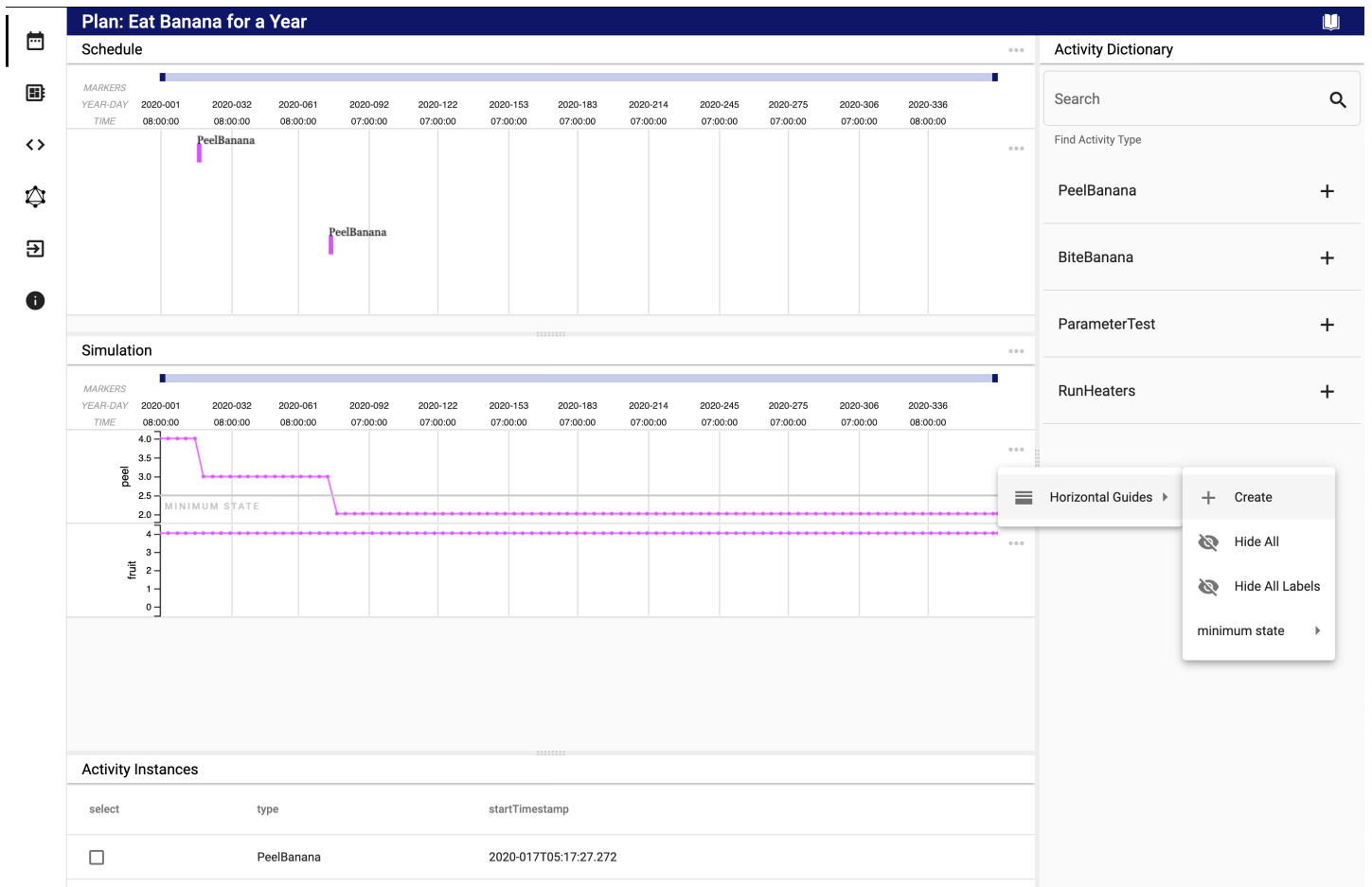


Figure 4.1: When violations occur, they will be represented as red areas on the timeline, with an accompanying hover state.

Horizontal guides may be added to any simulation or activity schedule band. The horizontal guides control UI may be accessed through each band's three dots more menu.



*Figure 4.2: Horizontal guides may be added to each activity schedule or simulation bands

Aerie UI provides a flexible arrangement where users can hide any of these panels by simply dragging dividers vertically. In Figure 5 this feature of the UI is illustrated.

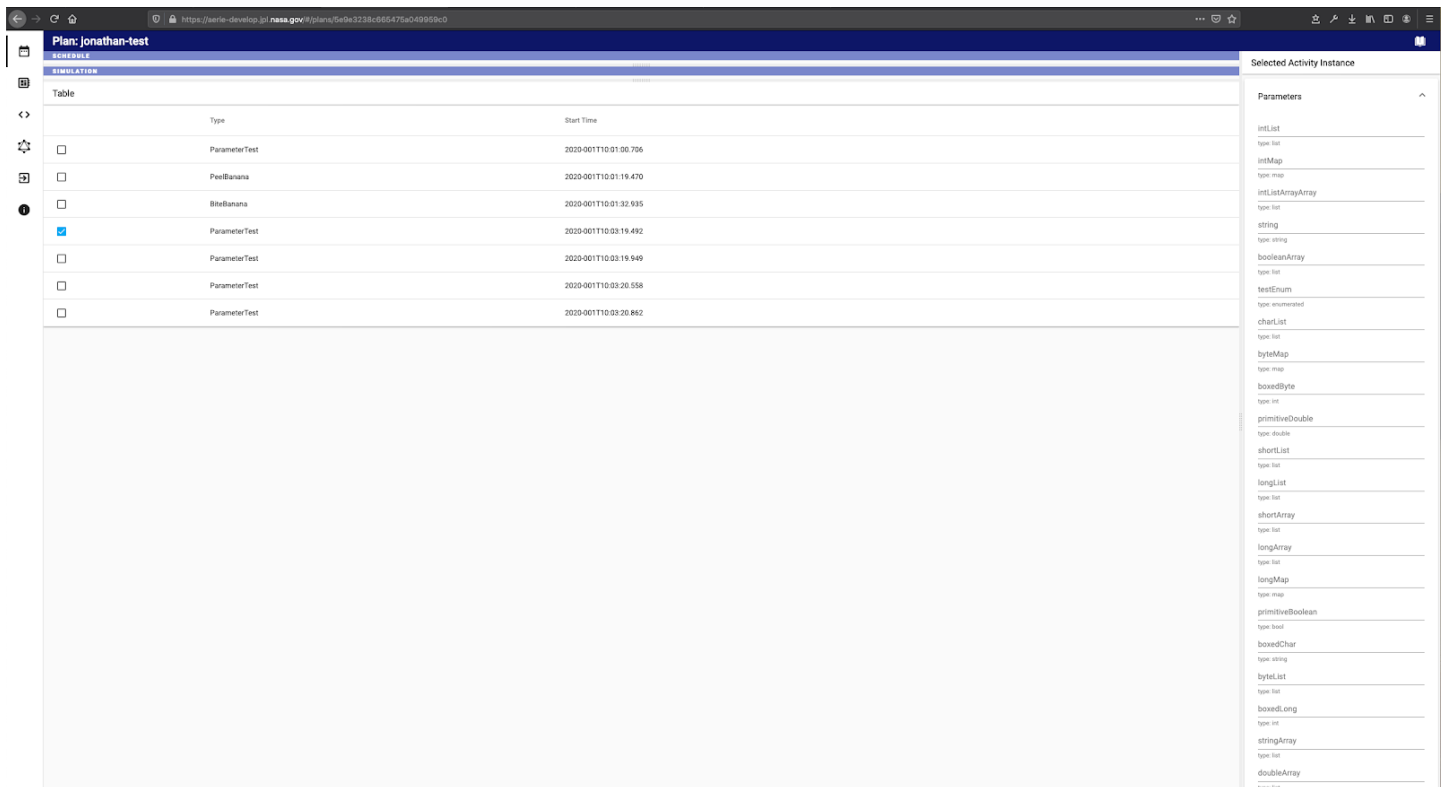


Figure 5: The bottom panels are dragged to the top edge completely leaving only one panels in view.

Note that all the default panels outlined here can be configured and changed based on the needs of a mission. You can read about how to do that in the [UI Configurability](#) documentation.

Simulation Configuration

The Aerie web application provides a graphical user interface to set and update simulation configuration arguments.

Using Web-App GUI

Navigate to the plans page. As seen in **Figure 1**, the plan creation form includes a "Simulation Configuration" section that accepts a user-provided file. Specifically, the user-provided file is expected to be a serialized (JSON) set of mission model arguments.

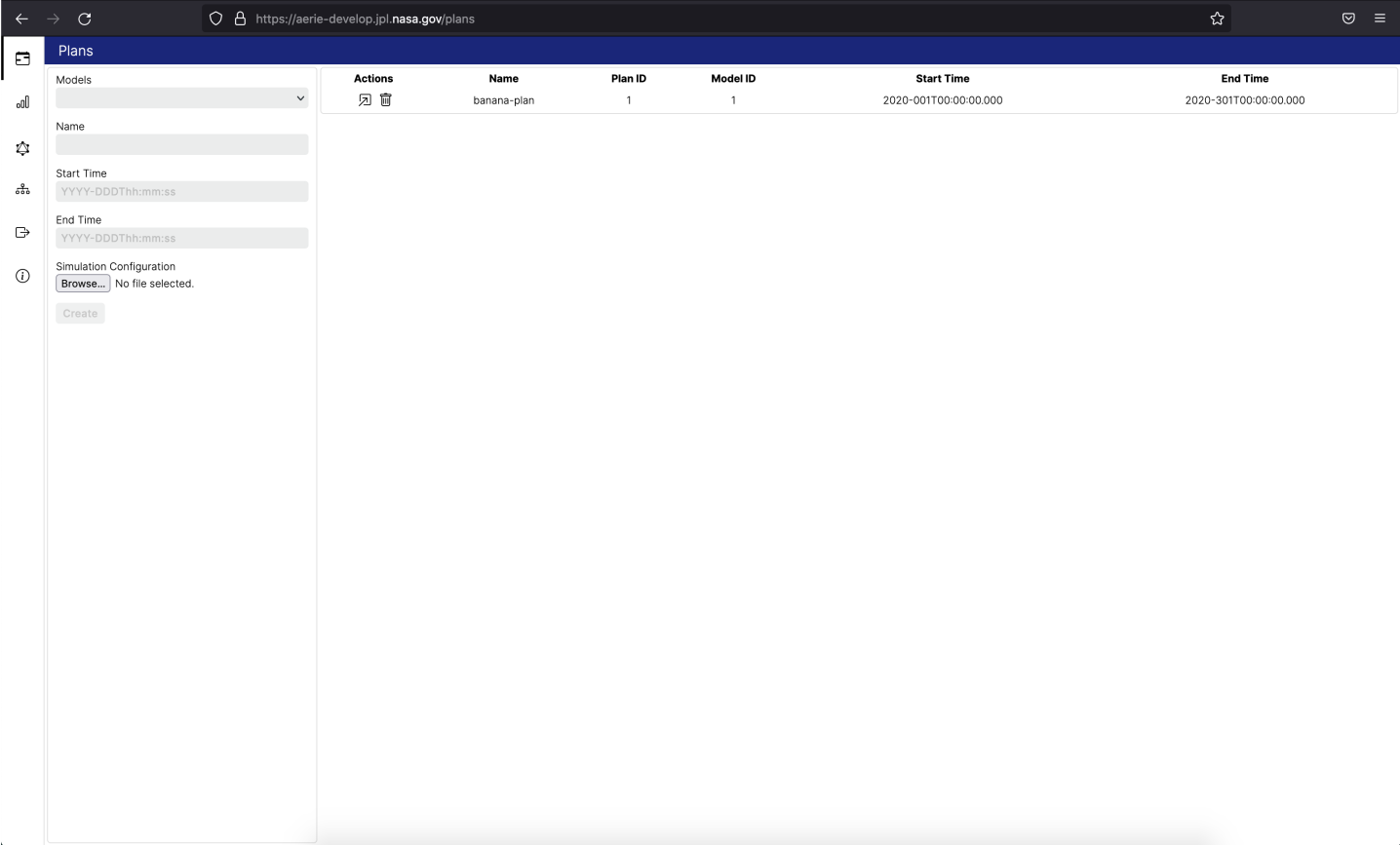


Figure 1: Plan creation and selection view.

Conceptually, these uploaded simulation configuration arguments are stored within a plan-specific simulation **template**. Within the GUI each plan may be associated with just one simulation template; individual simulations may set missing arguments or override existing arguments using the template's arguments as a base set of arguments. **Figure 2** shows the simulation configuration view where arguments for a specific simulation may be set/overridden.

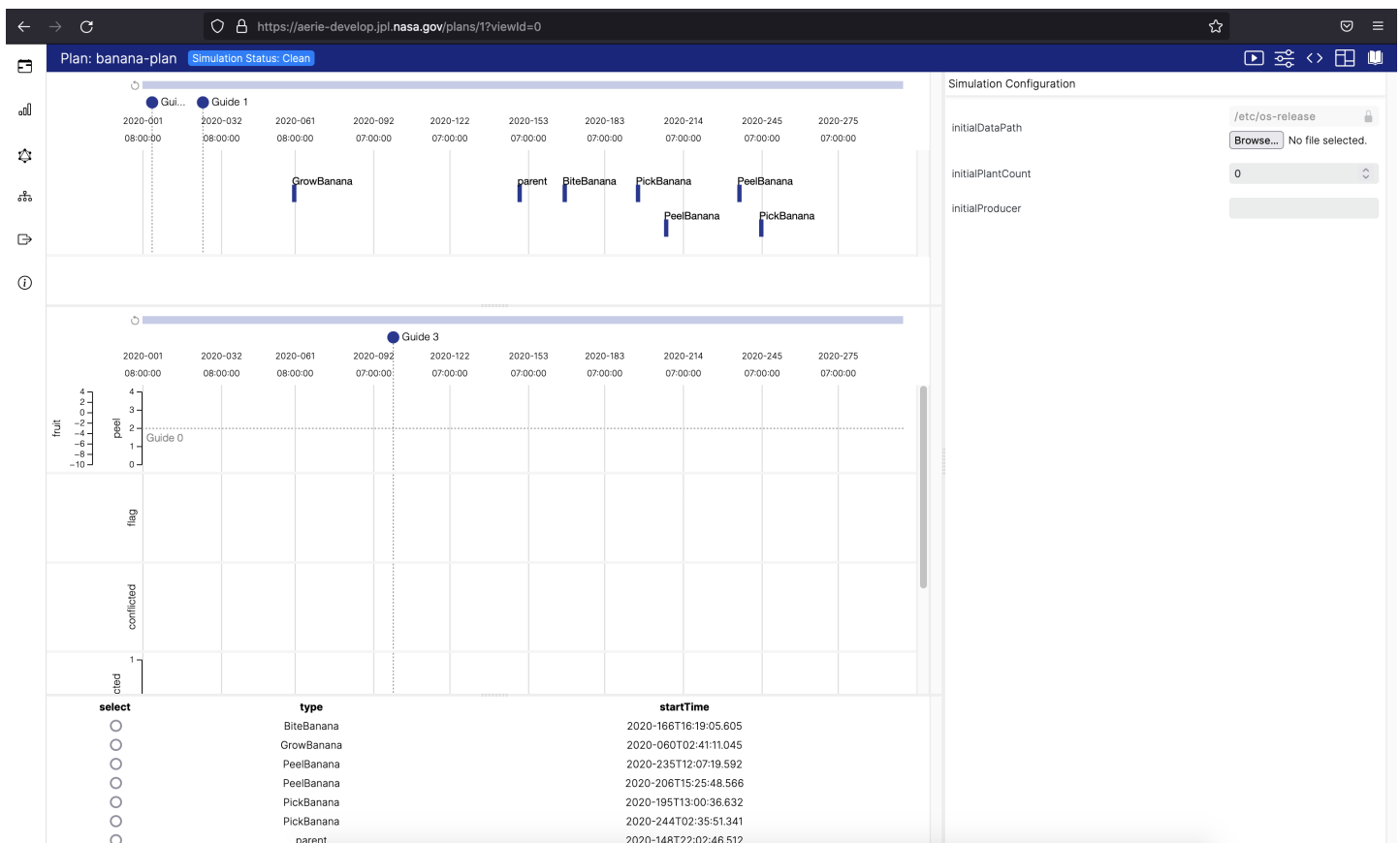


Figure 2: Simulation configuration view.

Formalized loosely, the sets of arguments at play here are:

- **T**, the template-defined arguments provided at plan creation time.
- **S**, the simulation-defined arguments provided prior to simulation invocation.
- **M**, the merged set of arguments derived from the union between **T** and **S**. Any duplicates will be resolved using arguments from **S**. This set is ultimately used in a simulation.

Using GraphQL API

Using the GraphQL API it is possible to define simulations from as many simulation templates as desired. Please refer to the [Aerie GraphQL API](#) [SIS](#) to get up-and-running with a GraphQL client capable of communicating with the Aerie API.

Create a Plan

Create a plan `test-plan`. This will trigger Merlin to create a new simulation that can be queried for from the GraphQL API:

```
query {
  simulation {
    id
  }
}
```

Which indicates that the one and only simulation `id` is:

```
{
  "data": {
    "simulation": [
      {
        "id": 1
      }
    ]
  }
}
```

This ID will be used to identify the newly created simulation.

Create a Simulation Template

A simulation template must be associated with a mission model. In this example the only existing mission model ID is `1`.

```
mutation {
  insert_simulation_template(objects: {
    model_id: 1,
    description: "first template",
    arguments: { initialPlantCount: 42, initialProducer: "template"}
  }) {
    returning {
      id
    }
  }
}
```

Resulting in:

```
{
  "data": {
    "insert_simulation_template": {
      "returning": [
        {
          "id": 1
        }
      ]
    }
  }
}
```

This ID will be used to identify the newly created simulation template.

Note that the arguments supplied here are not the full set of arguments required for simulation:

```
query {
  mission_model_parameters_by_pk(model_id: 1) {
    parameters
  }
}
```

Results in:

```
{
  "data": {
    "mission_model_parameters_by_pk": {
      "parameters": {
        "initialDataPath": {
          "order": 1,
          "schema": {
            "type": "path"
          }
        },
        "initialProducer": {
          "order": 2,
          "schema": {
            "type": "string"
          }
        },
        "initialPlantCount": {
          "order": 0,
          "schema": {
            "type": "int"
          }
        }
      }
    }
  }
}
```

As can be seen above, the template does not define a `initialDataPath` argument. In this case this must be provided by the simulation's argument set prior to simulation.

Associate Simulation with Simulation Template

Attach the simulation template to the current simulation and update the simulation's arguments.

```

mutation {
  update_simulation(
    _set: {
      simulation_template_id: 1,
      arguments: {
        initialPlantCount: 200,
        initialDataPath: "/etc/os-release"
      }
    },
    where: {id: {_eq: 1}}) {
    returning {
      id
    }
  }
}

```

The simulation with ID `1` is now associated with the template with ID `1`. In addition to being associated with a template, the simulation has defined a `initialPlantCount` argument override and a `initialDataPath` assignment.

Run the Simulation

```

query {
  simulate(planId: 1) {
    results
  }
}

```

The full results on this query are omitted for brevity but a sampling should look like:

```

"/plant": [
  {
    "x": 0,
    "y": 200
  },
  {
    "x": 31536000000000,
    "y": 200
  }
],
"/producer": [
  {
    "x": 0,
    "y": "template"
  },
  {
    "x": 31536000000000,
    "y": "template"
  }
]

```

Since an `initialProducer` argument was not provided in the simulation argument set the template's `initialProducer` value comes through here.

UI Views

Users can create custom planning views for different sub-systems (e.g. science, engineering, thermal, etc.), where only data (e.g. activities and resources) for those sub-systems are visualized. This is done through custom JSON files (or directly via the UI). The format of a UI View is the subject of this document.

See the [UI view JSON schema specification](#) for the complete set of view object properties and types. For more concrete examples see the [JSON view objects](#) included in the [default Aerie deployment](#).

View

This is the main type interface for the planning UI view:

```
type ViewActivityTable = {
  columnDefs: ColumnDef[];
  id: number;
};

type ViewIFrame = {
  id: number;
  src: string;
  title: string;
};

type ViewMeta = {
  owner: string;
  timeCreated: number;
  timeUpdated: number;
  version: string;
};

type ViewPlan = {
  activityTables: ViewActivityTable[];
  iFrames: ViewIFrame[];
  layout: Grid;
  timelines: Timeline[];
};

interface View {
  id: number;
  meta: ViewMeta;
  name: string;
  plan: ViewPlan;
}
```

For example, here is a JSON object that implements the `View` interface:

```
{
  "id": 0,
  "meta": {
    "owner": "system",
    "timeCreated": 1615235211527,
    "timeUpdated": 1615235211527,
    "version": "0.11.1"
  },
  "name": "Example View",
  "plan": {
    "activityTables": []
    "iFrames": [],
    "layout": {},
    "timelines": []
  }
}
```

View Layout (Grid)

A planning UI view consists of a layout which describes the different visible components and how they are arranged in resizable rows and columns. The layout follows the following `Grid` type definitions:

```

type GridComponentName =
  | 'ActivityForm'
  | 'ActivityTable'
  | 'ActivityTypes'
  | 'ConstraintEditor'
  | 'Constraints'
  | 'ConstraintViolations'
  | 'IFrame'
  | 'SchedulingEditor'
  | 'Scheduling'
  | 'Simulation'
  | 'Timeline'
  | 'TimelineForm'
  | 'ViewEditor'
  | 'Views';

type GridComponent = {
  activityTableId?: number;
  componentName: GridComponentName;
  gridName?: GridName;
  iFrameId?: number;
  id: number;
  timelineId?: number;
  type: 'component';
};

type GridColumns = {
  gridName?: GridName;
  id: number;
  columns: Grid[];
  columnSizes: string;
  type: 'columns';
};

type GridGutter = {
  gridName?: GridName;
  id: number;
  track: number;
  type: 'gutter';
};

type GridName = 'Activities' | 'Constraints' | 'Scheduling' | 'Simulation' | 'View';

type GridRows = {
  gridName?: GridName;
  id: number;
  rows: Grid[];
  rowSizes: string;
  type: 'rows';
};

type Grid = GridColumns | GridComponent | GridGutter | GridRows;

```

For example, here is a grid layout definition in JSON:

```
{
  "columnSizes": "1fr 3px 2fr 3px 1fr",
  "columns": [
    { "componentName": "ActivityForm", "id": 1, "type": "component" },
    { "id": 2, "track": 1, "type": "gutter" },
    {
      "id": 3,
      "rowSizes": "70% 3px 1fr",
      "rows": [
        {
          "componentName": "Timeline",
          "id": 4,
          "timelineId": 0,
          "type": "component"
        },
        { "id": 5, "track": 1, "type": "gutter" },
        {
          "activityTableId": 0,
          "componentName": "ActivityTable",
          "id": 6,
          "type": "component"
        }
      ],
      "type": "rows"
    },
    { "id": 7, "track": 3, "type": "gutter" },
    { "componentName": "ActivityTypes", "id": 8, "type": "component" }
  ],
  "gridName": "Activities",
  "id": 0,
  "type": "columns"
}
```

View Timeline

The `timelines` section allows you to specify a list of `timeline` visualizations which display time-ordered data (i.e. activities or resources). Here is the interface of a timeline:

```
interface Timeline {
  id: number;
  marginLeft: number;
  marginRight: number;
  rows: Row[];
  verticalGuides: VerticalGuide[];
}
```

To visualize data in a timeline you need to add row objects to the `rows` array. A row is a layered visualization of time-ordered data. Each layer of a row is specified as an object of the `layers` array. The interfaces for a `Row` and `Layer` are as follows:

```
interface Row {
  autoAdjustHeight: boolean;
  height: number;
  horizontalGuides: HorizontalGuide[];
  id: number;
  layers: Layer[];
  yAxes: Axis[];
}

interface Layer {
  chartType: 'activity' | 'line' | 'x-range';
  filter: {
    activity?: ActivityLayerFilter;
    resource?: ResourceLayerFilter;
  };
  id: number;
  yAxisId: number | null;
}
```

Here is a JSON object that creates a single row with one activity layer. Notice the `filter` property, which is a [JavaScript Regular Expression](#) that specifies we only want to see `activity` of `type` `.*`. This is a regex for giving all activity types.

```
{
  "autoAdjustHeight": true,
  "height": 200,
  "horizontalGuides": [],
  "id": 0,
  "layers": [
    {
      "activityColor": "#283593",
      "activityHeight": 20,
      "chartType": "activity",
      "filter": { "activity": { "type": ".*" } },
      "id": 0,
      "yAxisId": null
    }
  ],
  "yAxes": []
}
```

For data that has y-values (for example resource data), you can specify a y-axis and link a layer to it by ID. Here are the interfaces for `Axis` and `Label`:

```
interface Axis {
  color: string;
  id: number;
  label: Label;
  scaledDomain: (number | null)[];
  tickCount: number | null;
}

interface Label {
  align?: CanvasTextAlign;
  baseline?: CanvasTextBaseline;
  color?: string;
  fontFace?: string;
  fontSize?: number;
  hidden?: boolean;
  text: string;
}
```

Y-axes are specified in the row separately from layers so we can specify multi-way relationships between axes and layers. For example you could have many layers corresponding to a single row axis.

Here is the JSON for creating a row with two overlaid `resource` layers. The first layer shows only resources with the name `peel`, and uses the y-axis with ID `1`. The second layer shows only resources with the name `fruit`, and uses the y-axis with the ID `2`.

```

{
  "autoAdjustHeight": false,
  "height": 100,
  "horizontalGuides": [],
  "id": 1,
  "layers": [
    {
      "chartType": "line",
      "filter": { "resource": { "name": "peel" } },
      "id": 1,
      "lineColor": "#283593",
      "lineWidth": 1,
      "pointRadius": 2,
      "yAxisId": 1
    },
    {
      "chartType": "line",
      "filter": { "resource": { "name": "fruit" } },
      "id": 2,
      "lineColor": "#ffcd69",
      "lineWidth": 1,
      "pointRadius": 2,
      "yAxisId": 2
    }
  ],
  "yAxes": [
    {
      "color": "#000000",
      "id": 1,
      "label": { "text": "peel" },
      "scaleDomain": [0, 4],
      "tickCount": 5
    },
    {
      "color": "#000000",
      "id": 2,
      "label": { "text": "fruit" },
      "scaleDomain": [-10, 4],
      "tickCount": 5
    }
  ]
}

```

Product Guide

- [Product Installation](#)
- [System Requirements](#)
- [Administration](#)
- [Product Support](#)

Product Installation

Installation Instructions

Installation instructions are located in the Aerie repository [deployment documentation](#). There you can also find an example [docker-compose.yml](#) file.

Docker Images

External

Aerie Docker images are stored in [GitHub packages](#). The currently available images are:

```
ghcr.io/nasa-amos/aerie-gateway:latest
ghcr.io/nasa-amos/aerie-merlin:latest
ghcr.io/nasa-amos/aerie-scheduler:latest
ghcr.io/nasa-amos/aerie-commanding:latest
ghcr.io/nasa-amos/aerie-ui:latest
```

Known Issues

1. When using the IntelliJ IDE, upon a source file change, only the affected source files will be recompiled. This causes conflicts with the annotations processing being used for Activity Mapping. For now manually rebuilding every time is the solution.
2. Hasura requires Postgres 14+ A known issue with [older versions of Postgres < 14](#), interacts with Hasura to create the following. For fields absent in a query, Hasura passes the DEFAULT token into the generated SQL. For these older Postgres versions, when a database field that is defined as GENERATE ALWAYS, an SQL query cannot pass any token for the generated field. Yet Hasura provides one, trying to be helpful, and because of the Postgres bug linked above the database doesn't want the value. Hence we get an error message of the variety

```
"message": "cannot insert into column \"id\",
```

```
`"description": "Column \"id\" is an identity column defined as GENERATED ALWAYS."`
```

This is a [bug being tracked by Hasura](#). All this being said Aerie will require Postgres 14 and up so we absolve ourselves of having to track the issue.

System Requirements

Software Requirements

Name	Version
Docker	19.X
PostgreSQL	14.X

Supported Browsers

Name	Version
Chrome	Latest
Firefox	Latest

Hardware Requirements

Hardware	Details
CPU	2 Gigahertz (GHZ) or above
RAM	8 GB at minimum
Storage	15 GB (the system should have access to additional storage as system databases grow according to mission operations)
Display resolution	2560-BY-1600, recommended
Internet connection	High-Speed connection, at least 10MBPS

AWS EC2 Instance Type

The workload that can be submitted to Aerie is highly dependent on the computational complexity of the mission model being simulated. An [m4.large](#) or greater EC2 instance will satisfy generic usage of Aerie with simple mission model. For missions that develop more complex mission models, operations such as performing simulation will benefit from the increased CPU of a [c4.xlarge](#) or greater instance.

TCP Port Requirements

Service	Default Port	Public
Aerie Commanding	27184	No
Aerie Gateway	9000	Yes
Aerie Merlin	27183	No
Aerie Merlin Worker	5005	No
Aerie Scheduler	27185	No
Aerie UI	80	Yes
Hasura	8080	Yes
Postgres	5432	No

Administration

This product is using Docker containers to run the application. The Docker containers are internally bridged (connected) to run the application. Containers can be restarted in case of any issues using Docker CLI. See the above [TCP Port Requirements](#) for which containers should be exposed publicly/outside the Docker network.

Configuration

The `merlin` Docker image can be configured by mounting a JSON configuration file and providing the path to that file as a command-line argument to the container. For instance, if using Docker Compose and assuming the file is mounted at `/srv/config.json`, the line `command: /srv/config.json` can be added to the `merlin` container definition.

Environment Variables

The `merlin` Docker image can be provided additional JVM arguments, such as to configure the JVM allocated heap size. Add any desired JVM flags to the `JAVA_OPTS` environment variable for this container.

Network Communications

The Aerie deployment configures the port numbers for each container via docker-compose. The port numbers must match those declared within the services' config.json. In a large majority of Aerie deployments no change to these port numbers will be needed, nor should one be made. The only port number that might be desired to change is the Aerie-UI port (80). In this case the number to change is the first port number of the pair [XXXX:XXXX]. The second number represents the port number within the container itself.

Administration Procedures

Aerie is orchestrated as a set of Docker containers. Each of the software components are packaged and run in an isolated docker container independently from one another. There exists seven docker containers:

- Aerie UI: Hosts the web application and communicates with Aerie via the GraphQL Apollo Server.
- Aerie Gateway: Main API gateway for Aerie
- Merlin Server: Handles all the logic and functionality for activity planning.
- Postgres: Holds the data for the Merlin server container.
- Hasura: Serves the Aerie GraphQL API.

Aerie database containers are isolated to connect only with service containers internal to the application (Merlin server, UI server, Aerie gateway server). Aerie databases are not accessible directly from outside the Aerie application.

Network File System Deployment

You can install and use a Network File System with Aerie. Please see the [deployment documentation](#) for complete instructions.

Software Requirements

Name	Version
*Node.js	16.X LTS
*Open JDK	17.X

*For build purposes only. Not needed for installing the application.

Product Support

Defect Reporting Procedure

All defect reports should go to aerie_support@jpl.nasa.gov.

Points of Contact

- Deployment and Integration: Kenneally, Patrick W, Development Lead
- Administration: Kenneally, Patrick W, Development Lead
- General Help: Alper Ramaswamy, Emine Basak, Product Lead