# 0.9.2

# Table of contents

# 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 assoociated 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.

# Creating A Mission Simulation Overview

Running an Aerie simulation requires a mission model that describes effects of activities over modeled states, 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.

Aerie takes the mission model and the plan file as inputs and executes a simulation. Simulation currently returns all states declared in the mission model at a parameterized sampling rate. The simulation results format will be improved in upcoming releases. In later releases of Aerie will also return constraint violation results.

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

1. Install Aerie services following instructions on Product Guide
2. Create an 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 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. We recommend forking the mission model template to get started.

## package-info.java

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:

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

import gov.nasa.jpl.aerie.bananation.activities.BiteBananaActivity;
import gov.nasa.jpl.aerie.bananation.activities.ParameterTestActivity;
import gov.nasa.jpl.aerie.bananation.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 `@WithMappers` annotations are specified, and multiple mappers apply to the same type, mappers found in earlier instances of the annotation will take precedence. For more information on allowing custom values, see value mappers.

## Mission.java

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.

```
// bananation/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.

```java
// bananation/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 example `bananaanation` contains a `Configuration` data class example to demonstrate how a simple configuration may be created.

## Setup

### Mission Model

The `bananaanation` example makes use of the `@WithConfiguration` annotation within `package-info.java`:

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

@WithConfiguration(Configuration.class)
```

Where `Configuration` is a simple data class:

```
public final 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 Configuration defaultConfiguration() {
    return new Configuration(DEFAULT_PLANT_COUNT, DEFAULT_PRODUCER, DEFAULT_DATA_PATH);
  }
}
```

As long as a value mapper has been provided for a `Configuration` object (for example: `public static ValueMapper<Configuration> configuration()`), then the object will be correctly serialized/deserialized. To support a default configuration the mission model must define a forgiving configuration value mapper. Using the `bananaanation` example, `ConfigurationValueMapper` interprets an empty configuration (`SerializedValue.of(Map.of())`) as a default configuration. This value mapper also allows for the `initialDataPath` configuration parameter to be omitted in place of a default. A relevant excerpt from `bananaanation/src/main/java/gov/nasa/jpl/aerie/bananaanation/ConfigurationValueMapper.java`:

```
  @Override
  public Result<Configuration, String> deserializeValue(final SerializedValue serializedValue) {
    final var map$ = serializedValue.asMap();
    if (map$.isEmpty()) return Result.failure("Expected map, got " + serializedValue);
    final var map = map$.orElseThrow();

    // Return a default configuration when deserializing a null serialized value
    if (map.isEmpty()) return Result.success(Configuration.defaultConfiguration());

    if (!map.containsKey("initialPlantCount")) return Result.failure("Expected field \"initialPlantCount\", but not found: " + serializedValue);
    final var plantCount$ = new IntegerValueMapper().deserializeValue(map.get("initialPlantCount"));
    if (plantCount$.getKind() == Result.Kind.Failure) return Result.failure(plantCount$.getFailureOrThrow());
    final var plantCount = plantCount$.getSuccessOrThrow();

    if (!map.containsKey("initialProducer")) return Result.failure("Expected field \"initialProducer\", but not found: " + serializedValue);
    final var producer$ = new StringValueMapper().deserializeValue(map.get("initialProducer"));
    if (producer$.getKind() == Result.Kind.Failure) return Result.failure(producer$.getFailureOrThrow());
    final var producer = producer$.getSuccessOrThrow();

    // Use a default data path parameter if this parameter is not present within the serialized value being deserialized
    final var dataPath$ = new PathValueMapper().deserializeValue(map.getOrDefault("initialDataPath", new PathValueMapper().serializeValue(Configuration.
DEFAULT_DATA_PATH)));
    if (dataPath$.getKind() == Result.Kind.Failure) return Result.failure(dataPath$.getFailureOrThrow());
    final var dataPath = dataPath$.getSuccessOrThrow();

    return Result.success(new Configuration(
        plantCount,
        producer,
        dataPath));
  }
```

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) {
  // ...
}
```

# Use

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

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

# Activities

The mission system's behavior is modeled as a series of activities that emit discrete events. These events are manifested by the real mission system as the schedule of tasks of ground based assets and a spacecraft's onboard sequences, commands, or flight software. 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` 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 tailor the behavior of an activity instance's effect model without changing the activity type. These parameters can be used to determine the effects of the activity, as well as its duration, decomposition and expansion into commands.

```
/**
 * The bus power consumed by the instrument while it is turned on measured in Watts
 */
@Parameter
public double instrumentPower_W = 100.0;
```

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. For more information on mission-specific parameter types, see Value Mappers.

For models written in Java 16 and above, activity type parameters may be alternatively defined by using Record classes. Record classes are a special type of class intended to provide implicit functionality for classes and reduce verbosity in class

definitions. For more information on records in Java 16+, see Java Record Classes.

To define parameters in Java record classes, simply declare their name and type in the header for the record class.

```
@ActivityType("RotateCamera")
public final record RotateCameraActivity(float degrees, boolean clockwise) { ... }
```

For any activity types not declared using records, each of the parameters will be inferred as having a default value. In this sense, these parameters are all optional, such that a planner instantiating a given activity is not required to provide non-default values for these parameters. This can be demonstrated in the prior example, where `instrumentPower_W` is given a default value of `100.0`.

In record-style definitions, activities can have parameters which are either all required to be given a non-default value by the planner/user, or all optional such that they have a default value.

To provide an activity definition with default values for**all** of its parameters, provide a factory method that returns a default instance of the activity type. While the method name for this default factory can take any name, the method should be annotated with `@Template`.

```
@ActivityType("RotateCamera")
public final record RotateCameraActivity(float degrees, boolean clockwise) {
  public static @Template RotateCameraActivity defaults() {
    return new RotateCameraActivity(30.0, true);
  }
}
```

To provide an activity definition with default values for**some** of its parameters, provide a nested class with fields representing the type, name, and default value of the desired default parameters. While the name for this nested class can take any name, the class should be annotated with `@WithDefaults`.

```
@ActivityType("CapturePhoto")
public final record CapturePhotoActivity(int pixelWidth, int pixelHeight, double dlRate) {
  public static @WithDefaults final class Defaults {
    public static double dlRate = 3200.0;
  }
}
```

To define an activity type as having all required parameters (with no defaults), simply do not provide any template or factory methods. An activity defintion without such a method will be inferred as only having non-default parameters that must be provided by the planner/user.

```
@ActivityType("TurnInstrumentOff")
public final record TurnInstrumentOffActivity(boolean resetState) {
  //No Default set by the modeler
  //The parameter resetState MUST be provided by the user
}
```

# Validations

An 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 activity. A message will be provided to the planner for each failing validation, so the order of validation methods does not matter.

# 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 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.

# 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 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 (see below), 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.

## `ValueSchema` Explained

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. The `ValueSchema` class provides a set of values to be used for this purpose:

`ValueSchema.BOOLEAN` represents a boolean value `ValueSchema.INT` represents an integer value `ValueSchema.REAL` represents a real number, typically with a decimal part `ValueSchema.STRING` represents a string of characters `ValueSchema.DURATION` represents a duration value

In addition to these types, `ValueSchema` also provides some more complex types. These require parameters to describe their structure in more detail, so they are provided as methods:

`ValueSchema.ofSeries(ValueSchema value)` represents a list of a single type, described by the provided `ValueSchema` `ValueSchema.ofStruct(Map<String, ValueSchema> map)` represents a structured value, containing named componenets, each with their own value, represented by the associated `ValueSchema` in `map` `ValueSchema.ofVariant(Class<? extends Enum> enumeration)` represents an enumerated value, and requires the represented enumeration be provided

Using the described fields and methods of `ValueSchema`, any value's structure should be describable.

### Examples of ValueSchema

Below are a few examples of how `ValueSchema` might be used to describe some values:

`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> biz
}
```

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

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

# Generated Activity Mappers

In most cases, you will likely want to let Merlin generate activity mappers for you. Thankfully, this is the 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:

```java
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` (see below).

There are plenty of examples of value mappers over in the contrib module.

## 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

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 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 high light 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 l ong 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 a s 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
```java
@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) {
    // ...
  }
}
```

## Activity Mapper

```java
public final class FooActivityMapper implements ActivityMapper<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 Map<String, ValueSchema> getParameters() {
    final var parameters = new HashMap<String, ValueSchema>();
    parameters.put("x", this.mapper_x.getValueSchema());
    parameters.put("y", this.mapper_y.getValueSchema());
    parameters.put("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 instantiateDefault() {
    return new FooActivity();
  }

  @Override
  public FooActivity instantiate(final Map<String, SerializedValue> arguments) throws
      TaskSpecType.UnconstructableTaskSpecException {
    final var activity = new FooActivity();
    for (final var entry : arguments.entrySet()) {
      switch (entry.getKey()) {
        case "x":
          activity.x = this.mapper_x
              .deserializeValue(entry.getValue())
              .getSuccessOrThrow($ -> new TaskSpecType.UnconstructableTaskSpecException());
          break;
        case "y":
          activity.y = this.mapper_y
              .deserializeValue(entry.getValue())
              .getSuccessOrThrow($ -> new TaskSpecType.UnconstructableTaskSpecException());
          break;
        case "vecs":
          activity.vecs = this.mapper_vecs
              .deserializeValue(entry.getValue())
              .getSuccessOrThrow($ -> new TaskSpecType.UnconstructableTaskSpecException());
          break;
        default:
          throw new TaskSpecType.UnconstructableTaskSpecException();
      }
    }
    return activity;
  }

  @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;
  }
}
```

# 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. Seethe repository for more details.

# Creating Plans

Plans can be created via Planning Web GUI, by uploading a JSON plan file through the Aerie CLI, and interfacing with the Aerie GraphQL API. Instructions on these interfaces can be found in their respective pages.

A sample JSON Plan file format can be found in the User Guide Appendix

# Constraints



## 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 syntnax 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 beserialized

## 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

# Glossary

## Aerie

**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 'GDS' deployment.

**Mission model**: Aerie is trying to move away from the term 'adaptation'; see third paragraph below. However, this term still exists in some of our messaging. 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.

## Simulation and Modeling

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

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

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

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

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

- Activity
- Effect Model
- Activity Type
- Activity Type Parameters
- Activity Arguments
- Parameter
- Argument
- Model Parameter
- Condition
- Constraint
- Flight Rule
- Schedule Rule/Goal/Profile
- Opportunity Window
- Effect/Event
- GraphQL
- CAM
- Plan vs Schedule
- Scheduler vs. Planner
- Event graph

- Register
- Decomposition
- Simulation Actions/Control - get a better term for this
- Spawn
- Call
- Delay
- - pull out domain model terminology
- UI Compoenents
- Timelines
- Rows
- Layers
- Guides-horz/vert
- UI View
- Simulation
- Simulation Results
- Simulation configuration (model parameters)
- JAR
- Mission model interface
- Docker
- Hasura
- Apollo/Gateway/Aerie API
- AWS
- JUnit
- JVM
- SPICE
- PostgreSQL
- Simulation worker
- Simulation isolation
- Banananation
- Linear/Real/Discrete resource
- Dynamics
- Behavior
- Records based activities
- Required/Optional Parameters

# 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, Slinc II / CTS and ULSGEN. 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

# Merlin Activity Plans

Aerie provides a "plan service" that manages a repository of plans. The repository 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 0.9.1 release.

## Terminology and Notation

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

# Aerie GraphQL API Environment

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.

## Interface Overview

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 Components

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.

# Methods of Querying The Aerie API

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 Graphqurl.

### 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_PASSOWRD"
}
```

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://<you_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.

## GraphQL 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

# API 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

## Basic Usage 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
  }
}
```

## 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
    }
  }
}
```

# The Aerie API GraphQL 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://:9000/playground or the current Aerie release can be seen at https://aerie-staging.jpl.nasa.gov:9000/playground

# Aerie Planning UI

The Aerie planning web application provides a graphical user interface to create, view, update and delete mission model and plans. This section will refer to the demo instance of the UI available at: https://aerie-staging.jpl.nasa.gov

## Uploading Mission Models

Mission models can be uploaded to Aerie via the UI. To navigate to themission 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="Bananananation", version="0.0.1")`, then the name field must be entered as `Bananananation` 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.

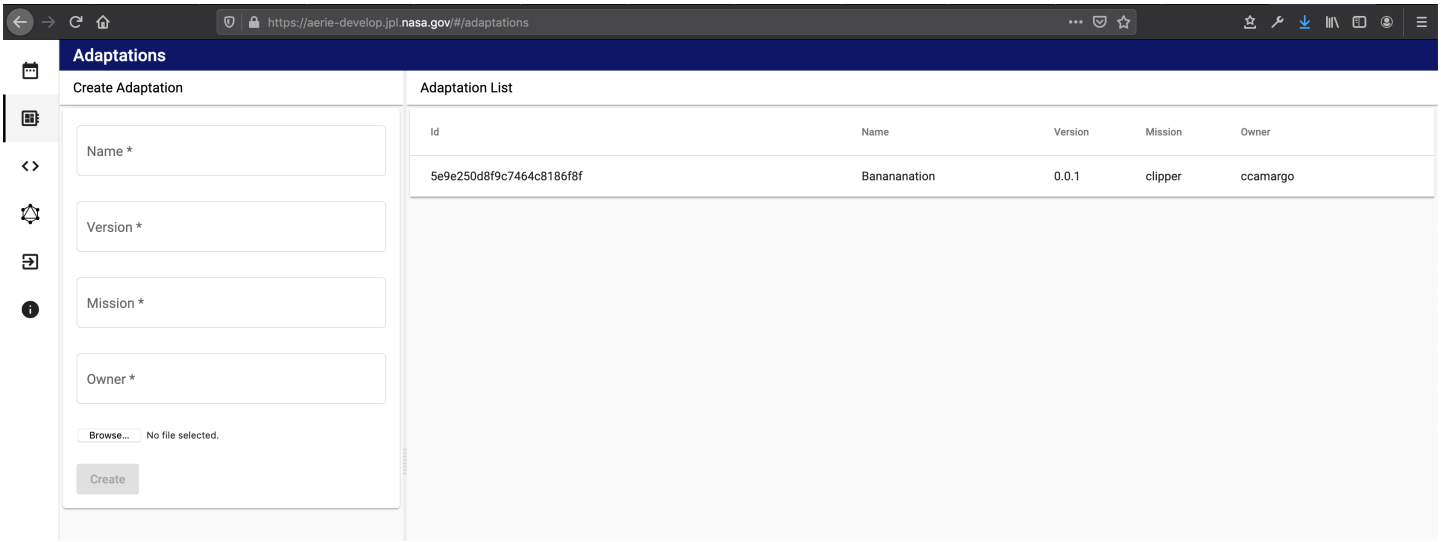


*Figure 1: Upload mission model, and view existing mission model.*

## 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.

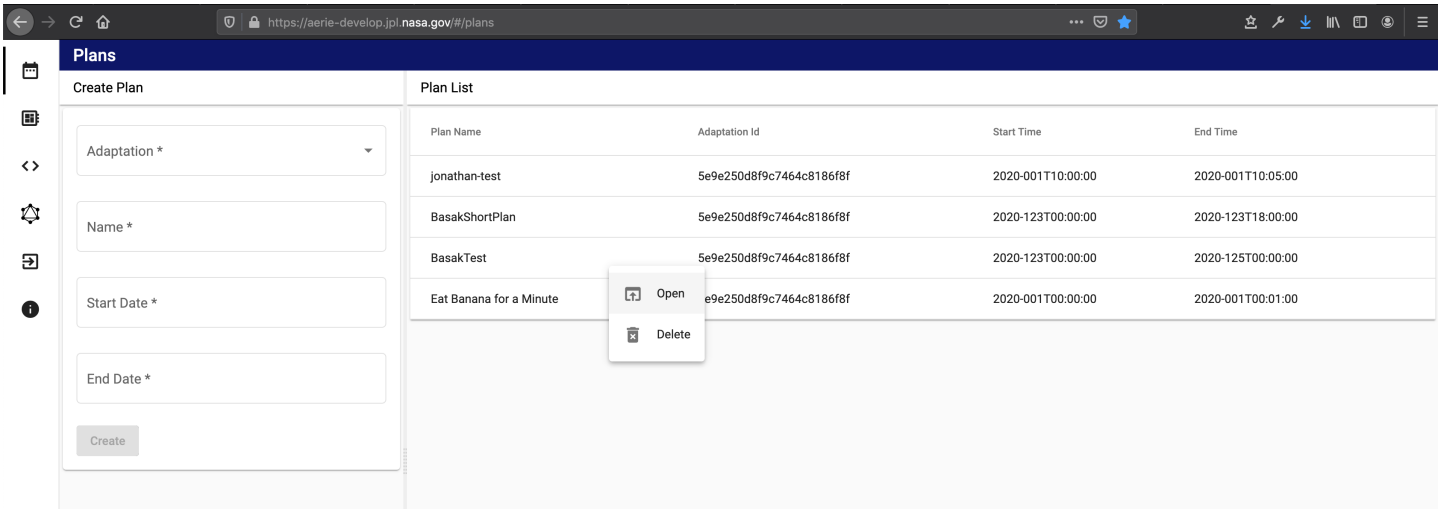


*Figure 2: Create plans, and view existing plans.*

## 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.

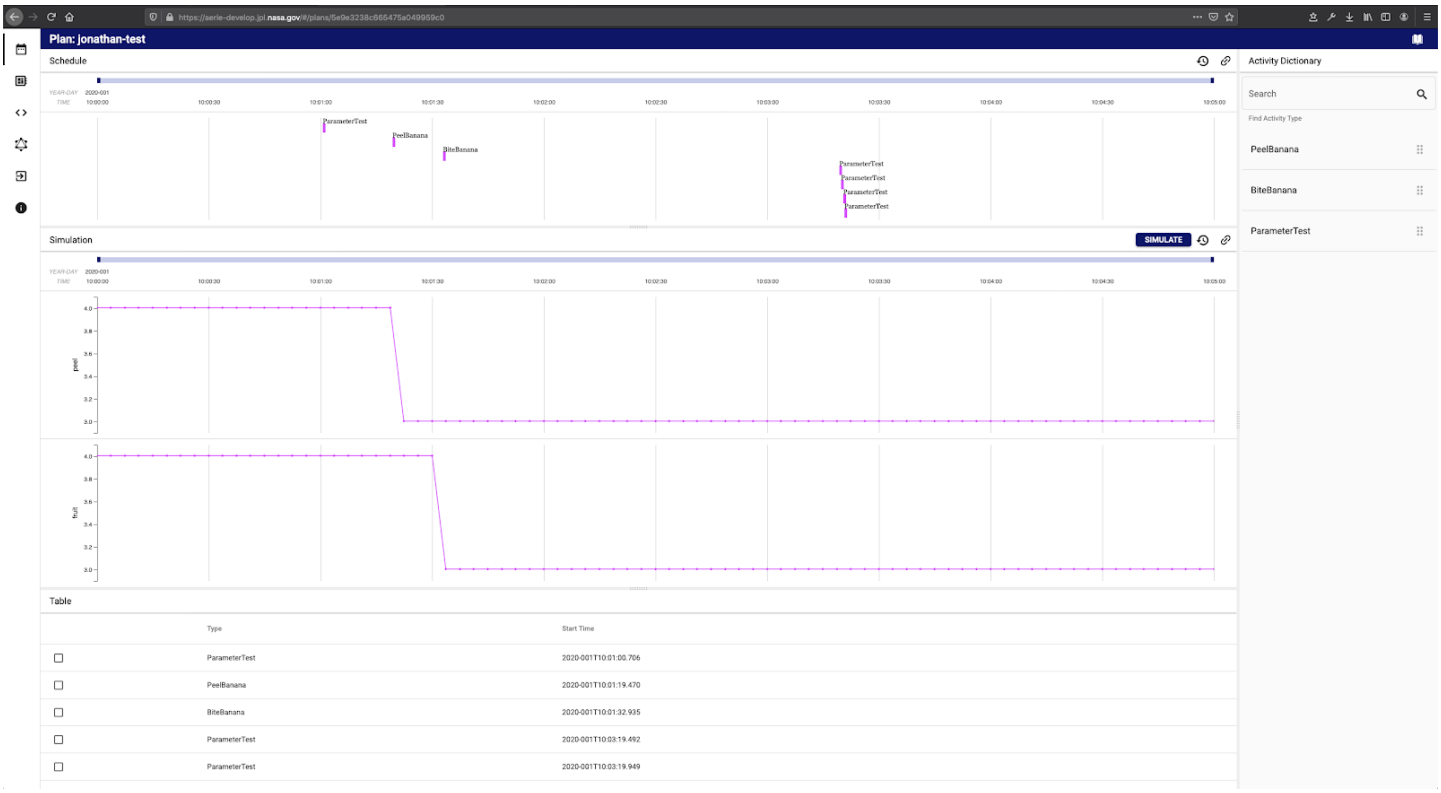


*Figure 3: Default panels.*

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.

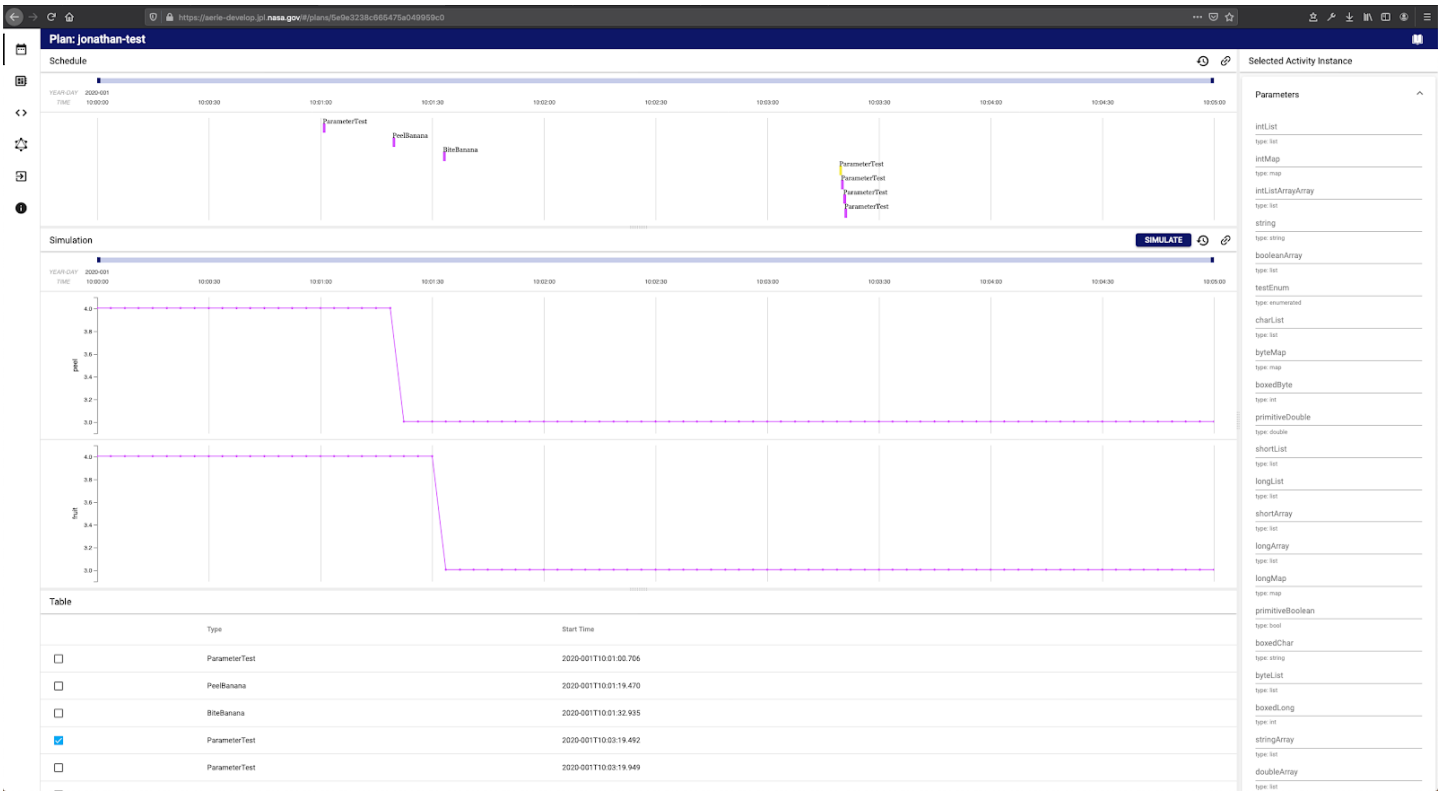


*Figure 4: When an activity instance in plan is selected, its details will appear in the right drawer.*

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

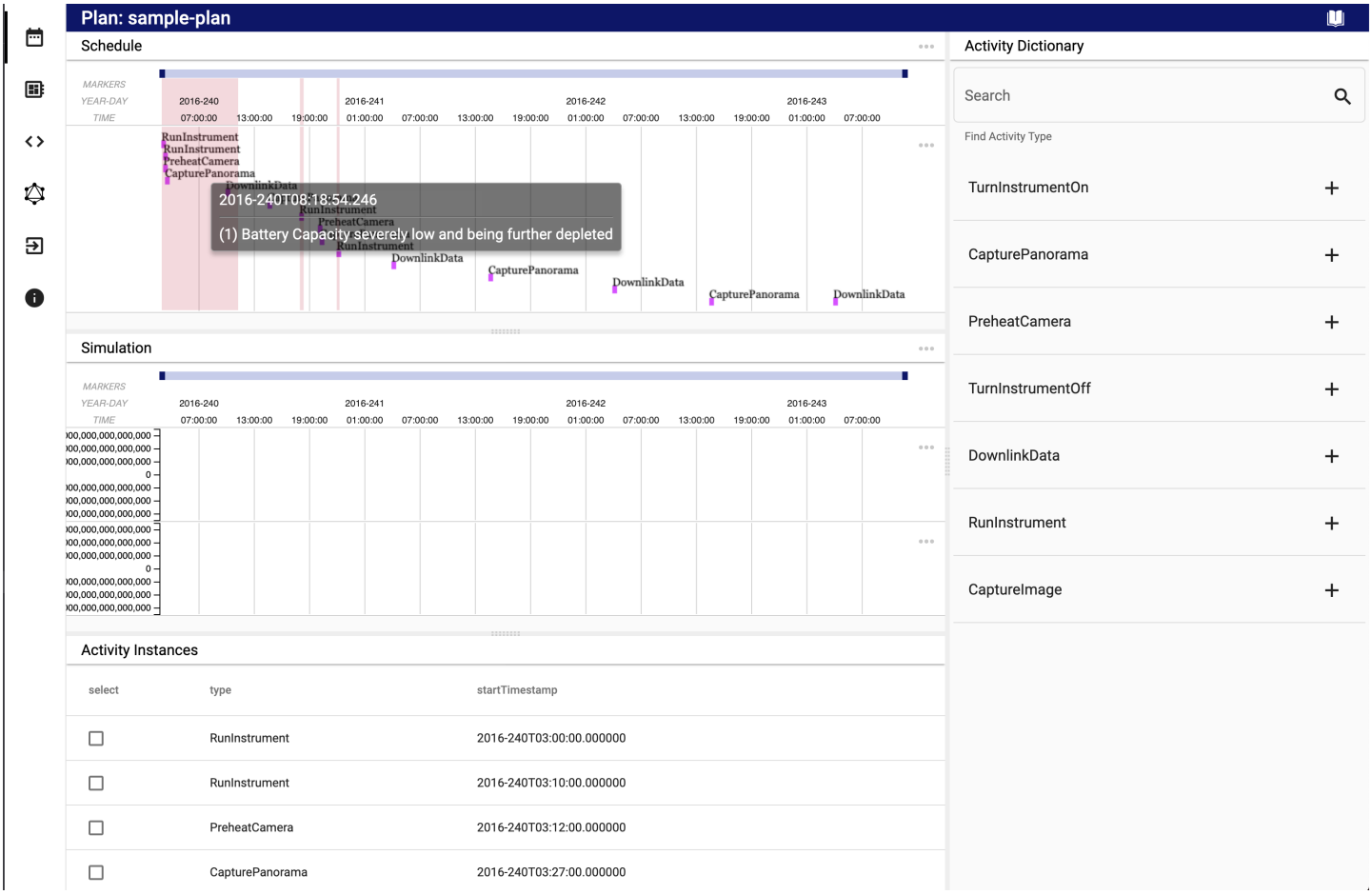


*Figure 4.1: When violations occur, they will be represented as red areas on the timeline, with an accompanying hover state.*

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.

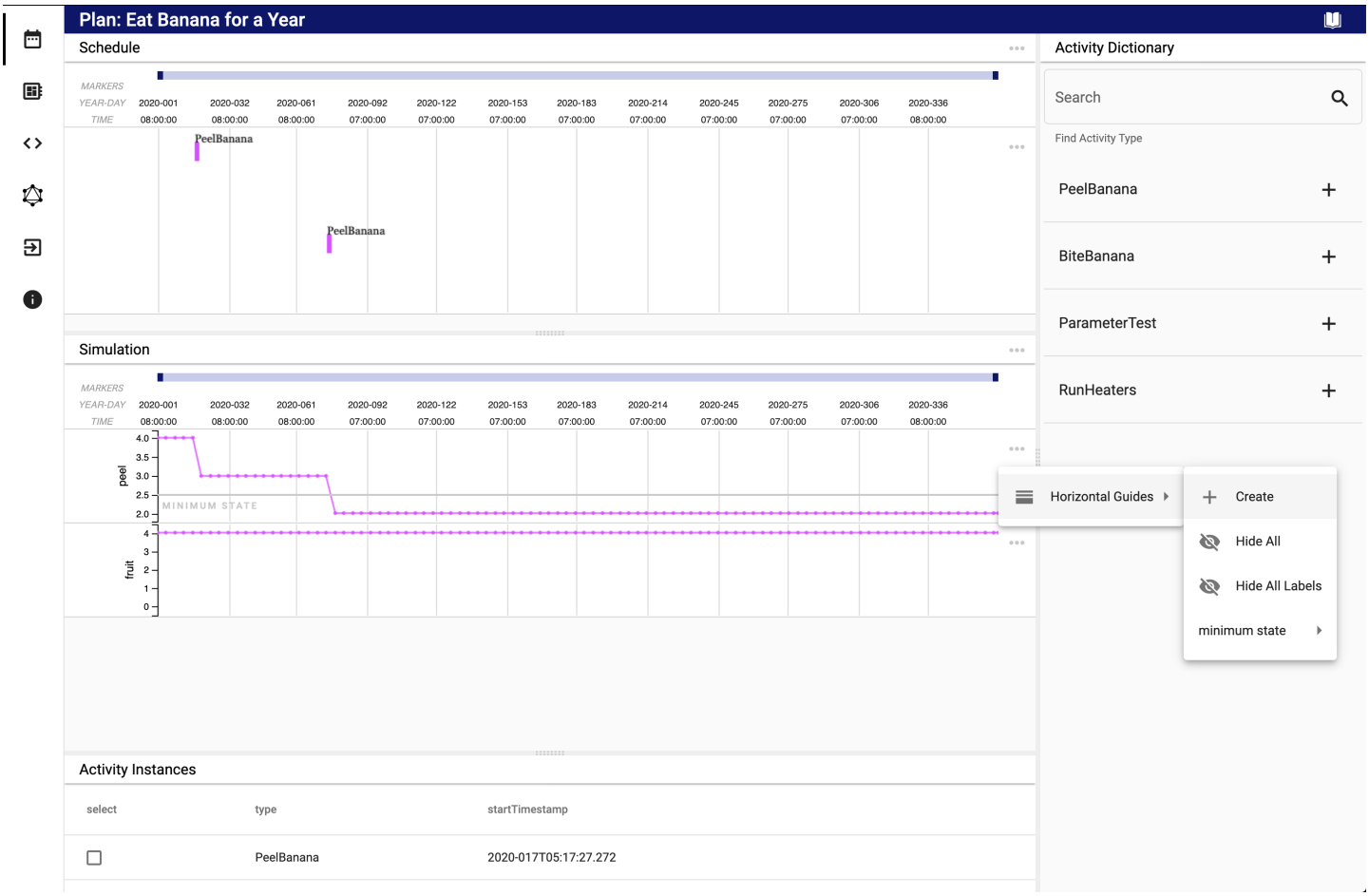


*Figure 4.2: Horizontal guides may be added to each activity schedule or simulation bands

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.

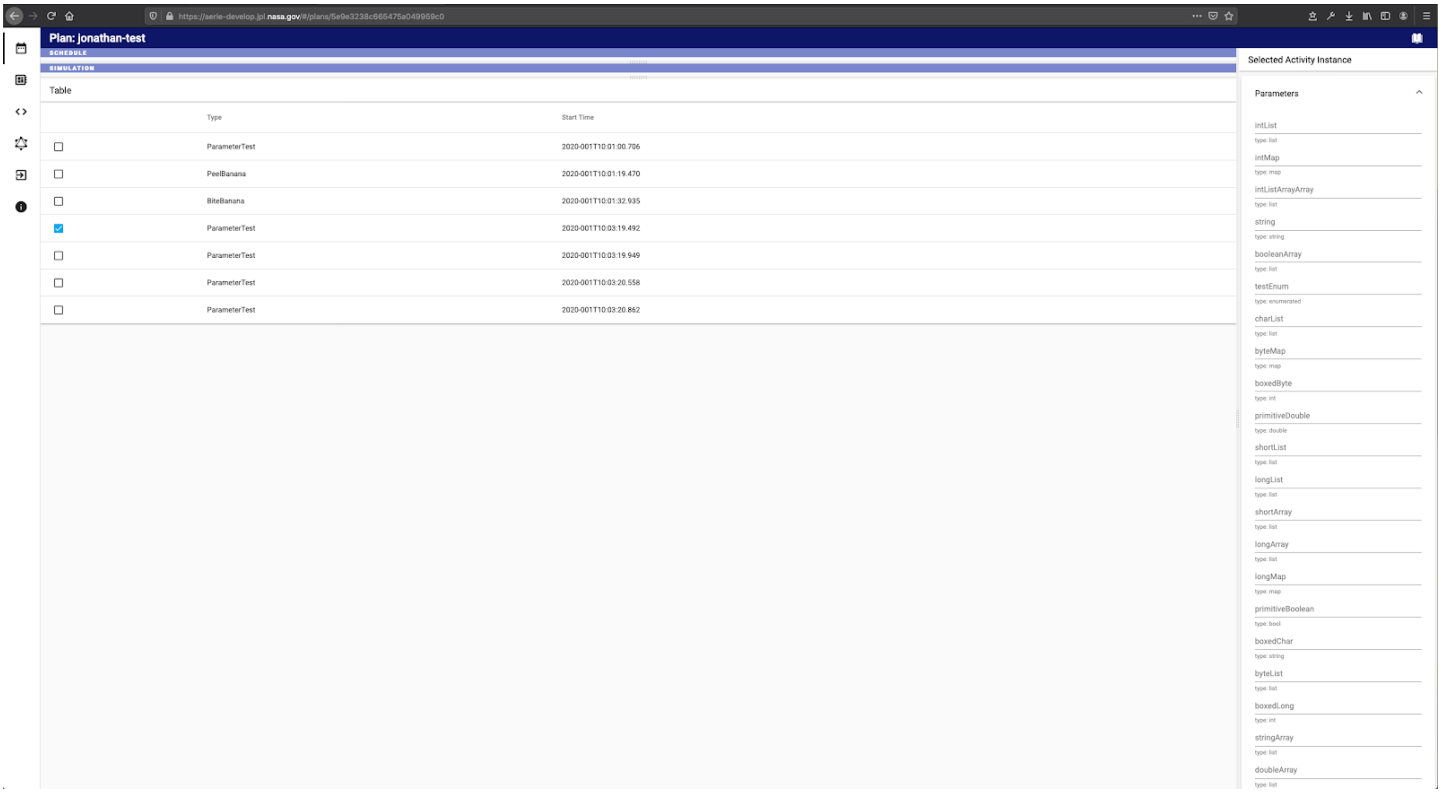


*Figure 5: The bottom panels are dragged to the top edge completely leaving only one panels in view.*

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.

# UI Configurability

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 a mission-authored JSON file. The format of that file and how to update it is the subject of this document.

## View

This is the main interface for the planning UI view.

```
interface View {
  id: string;
  meta: ViewMeta;
  name: string;
  sections: ViewSection[];
}

interface ViewMeta {
  owner: string;
  timeCreated: number;
  timeUpdated: number;
}
```

## View Sections

A planning UI view consists of a list of sections, and each `ViewSection` has the following interface:

```
interface ViewSection {
  id: string;
  iframe?: { // For type 'iframe'
    src: string;
  };
  menu?: SectionMenuItem[];
  size: number; // Size percentage of the section (0% - 100%)
  table?: { // For type 'table'
    columns: string[];
    type: 'activity'; // Only 'activity' typed tables are allowed right now
  };
  timeline?: Timeline; // For type 'timeline'
  title: string;
  type: 'iframe' | 'table' | 'timeline';
}
```

### Menu

Each section can have an associated menu specified with an `action`, `icon`, and `title`. The current actions we support are:

1. `link` adds a link that opens a new browser tab to the specified `url` in the `data` object.
2. `restore` is useful if the section is type `timeline`. It resets the timeline to it's max-time-rage (i.e. zooms all the way out).
3. `simulate` runs a simulation. Any rows with simulation result resources will be updated after a simulation.

More actions will be supported in the future and they will be more customizable. The icon should be a material icon. The menu interface looks like this:

```
export type ViewSectionMenuItemAction = 'link' | 'restore' | 'simulate';

export interface ViewSectionMenuItem {
  action: ViewSectionMenuItemAction;
  data?: {
    url?: string;
  };
  icon: string;
  title: string;
}
```

A couple example menu JSON objects looks like this:

```
[
  {
    "action": "link",
    "data": {
      "url": "https://google.com"
    },
    "icon": "link",
    "title": "Google"
  },
  {
    "action": "restore",
    "icon": "restore",
    "title": "Restore Time"
  }
]
```

## Types

There are currently three types of supported sections: iframe, table, and timeline. The following sections will detail how to create each of these section types.

### Inline Frame (iframe)

The iframe section allows you to embed a custom HTML page inside of the section. To create an iframe section you need to specify `type: "iframe"`, a unique `id`, a `size`, a `title`, and an `iframe` object with a `src`. The `src` is a URL of the HTML page you are embedding in the section. Here is a basic example of specifying an iframe section:

```
{
  "id": "section3",
  "iframe": {
    "src": "https://www.chartjs.org/samples/latest/charts/line/basic.html"
  },
  "size": 100,
  "title": "Line Chart",
  "type": "iframe"
}
```

### Table

The table section allows you to view data as a table with columns and rows. You can currently only create table sections for activity data. To create a table section you need to specify `type: "table"`, a unique `id`, a `size`, a `title`, and a `table` object. The table object specifies the columns you want to see in your data. For example if you want to see an activities `startTimestamp` property you specify it in the column. There is also a special `select` column that allows the column to be selected. Here is a basic example of specifying a table section:

```
{
  "id": "section2",
  "size": 100,
  "table": {
    "columns": [
      "select",
      "type",
      "startTimestamp"
    ],
    "type": "activity"
  },
  "title": "Activity Table",
  "type": "table"
}
```

### Timeline

The timeline section allows you to specify visualizations of time-ordered data. To create a timeline section you need to specify `type: "timeline"`, a unique `id`, a `size`, a `title`, and a list of `rows`. Here is the interface of a timeline:

```
interface Timeline {
  id: string;
  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: string;
  layers: Layer[];
  yAxes?: Axis[];
}

interface Layer {
  chartType: 'activity' | 'line' | 'x-range';
  color?: string;
  filter?: {
    activity?: {
      type?: string;
    };
    resource?: {
      name?: string;
    };
  };
  id: string;
  type: 'activity' | 'resource';
  yAxisId?: string;
}
```

Here is a JSON object that creates a single row with one activity layer. Notice there are nc`yAxes`, as activities do not typically have y-values. Also 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.

```
{
  "id": "row0",
  "layers": [
    {
      "chartType": "activity",
      "filter": {
        "activity": {
          "type": ".*"
        }
      },
      "id": "layer0",
      "type": "activity"
    }
  ]
}
```

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 {
  id: string;
  color?: string;
  label?: Label;
  scaleDomain?: number[];
  tickCount?: number;
}

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 axis.

Here is the JSON for creating a row with two overlayed`resource` layers. The first layer shows only resources with the name`peel`, and uses the y-axis with ID `yAxis1`. The second layer shows only resources with the name`fruit`, and uses the y-axis with the ID `yAxis2`.

```
{
  "id": "row1",
  "layers": [
    {
      "chartType": "line",
      "filter": {
        "resource": {
          "name": "peel"
        }
      },
      "id": "layer1",
      "type": "resource",
      "yAxisId": "yAxis1"
    },
    {
      "chartType": "line",
      "filter": {
        "resource": {
          "name": "fruit"
        }
      },
      "id": "layer2",
      "type": "resource",
      "yAxisId": "yAxis2"
    }
  ],
  "yAxes": [
    {
      "id": "yAxis1",
      "label": {
        "text": "peel"
      }
    },
    {
      "id": "yAxis2",
      "label": {
        "text": "fruit"
      }
    }
  ]
}
```

# Aerie Editor -Falcon

Please see the Aerie Editor (Falcon) user guide here.

# User Guide Appendix

## Appendix

### CLI JSON DOWNLOAD PLAN FORMAT SAMPLE

```
{
   "name": "example_plan",
   "missionModelId": "5df16e65a920f467637bac3a",
   "startTimestamp": "2018-331T00:00:00",
   "endTimestamp": "2018-332T00:00:00",
   "activityInstances": {
      "5e1caea5176cfc2d58b2c54a": {
         "type": "BiteBanana",
         "startTimestamp": "2018-331T04:00:00",
         "parameters": {
            "biteSize": 7.0
         }
      },
      "5e1caea5176cfc2d58b2c54b": {
         "type": "PeelBanana",
         "startTimestamp": "2018-331T04:00:00",
         "parameters": {
            "peelDirection": "fromStem"
         }
      }
   }
}
```

### CLI JSON UPLOAD PLAN FORMAT SAMPLE (No unique ID for activity instances)

```
{
   "missionModelId": "5df16e65a920f467637bac3a",
   "endTimestamp": "2018-331T00:00:00",
   "name": "example_plan",
   "startTimestamp": "2018-332T00:00:00",
   "activityInstances": [
    {
         "type": "BiteBanana",
         "parameters": {
               "biteSize": 7.0
         },
         "startTimestamp": "2018-331T04:00:00"
    },
     {
         "type": "PeelBanana",
         "parameters": {
            "peelDirection": "fromStem"
         },
        "startTimestamp": "2018-331T04:00:00"
    }
   ]
}
```

### CLI JSON APPEND ACTIVITY INSTANCES FORMAT SAMPLE

```
[
   {
      "type": "BiteBanana",
      "parameters": {
            "biteSize": 7.0
      },
      "startTimestamp": "2018-331T04:00:00"
   },
   {
      "type": "PeelBanana",
      "parameters": {
         "peelDirection": "fromStem"
      },
      "startTimestamp": "2018-331T04:00:00"
   }
]
```

### QUERY AN ACTIVITY TYPE FOR AN MISSION MODEL OUTPUT SAMPLE

```
{
  "parameters": {
    "peelDirection": {
      "type": "string"
    }
  },
  "defaults": {
    "peelDirection": "fromStem"
  }
}
```

# FAQ

This page contains frequently asked questions and answers for Aerie.

## Questions



## I am developing an adaptation. When I build the mission model, how do I get it to automatically update in the UI without having to re-upload it every time?

You will need to run Aerie locally to do this. See thedeployment documentation for specifics. On your local host machine, create a directory to store mission model files. Here is where I did it on my machine:

```
mkdir /Users/ccamargo/Projects/mission_model_files
```

Edit the docker-compose.yml file to mount your mission model files in your newly created directory instead of inside a docker volume. See the `volumes` section:

```
merlin:
  ...
  volumes:
    - /Users/ccamargo/Projects/adaptation_files:/usr/src/app/mission_model_files
```

Start the system with docker-compose:

```
docker-compose up --build --detach
```

Build your mission model and note the name of the output `.jar`. In this example we will use `mission-model.jar`:

```
cd mission-model # Change to your mission model directory
./gradlew build # Outputs 'build/libs/mission-model.jar'
```

Next upload your mission model to Aerie. Make sure the `name` field of the mission model matches the name of the `.jar`. In this example `name == 'mission model'`. You can do this in the UI (recommended), or with a GQL query (you will need to use a GQL client that supports file upload):

```
mutation CreateAdaptation($file: Upload!) {
  createAdaptation(
    file: $file
    mission: "test"
    name: "adaptation"
    owner: "test"
    version: "1.0.0"
  ) {
    message
    success
  }
}
```

After the upload succeeds, you should see your mission model `.jar` in your mounted directory. In this example:

```
/Users/ccamargo/Projects/adaptation_files/mission-model.jar
```

Create and open a plan in the UI using this mission model and make sure the activity dictionary loads correctly.

Finally, make edits, rebuild your mission model, overwrite the `mission-model.jar` file in your mounted `mission_model_files` directory, and restart the Merlin service. You can follow these commands (or add them to a local build script for quick iteration):

```
cd adaptation
./gradlew build
rm /Users/ccamargo/Projects/mission_model_files/mission-model.jar
cp ./build/libs/mission-model.jar /Users/ccamargo/Projects/mission_model_files
docker restart docker-compose-aerie_merlin_1
```

This will automatically update the mission model `.jar`, and thus all associated plans with that mission model. Refresh the plan in the UI to see the mission model changes.

## How do I upload an mission model to Aerie using the Python `requests` library?

You need to build a request using the GraphQL multipart request specification. Here is an example:

```
import json
import requests

url = 'http://localhost:27184' # URL of GraphQL Apollo server.
fileName = 'adaptation.jar'
fileJar = open(fileName, 'rb')
fileType = 'application/java-archive'
headers = { 'authorization': '' } # Add your auth token here.
query = """
  mutation CreateAdaptation(
    $file: Upload!
    $mission: String!
    $name: String!
    $owner: String!
    $version: String!
  ) {
    createAdaptation(
      file: $file
      mission: $mission
      name: $name
      owner: $owner
      version: $version
    ) {
      id
      message
      success
    }
  }
"""
operations = json.dumps({
  'query': query,
  'variables': {
    'file': None,
    'mission': 'test',
    'name': 'adaptation',
    'owner': 'test',
    'version': '1.0.0'
  }
})
map = json.dumps({
  'file': ['variables.file']
})
data = {
  'operations': operations,
  'map': map
}
files = {
  'file': (fileName, fileJar, fileType)
}

response = requests.post(url, files=files, data=data, headers=headers)
result = response.json()
print(result)
```

# Product Guide



# Product Installation

## Installation Instructions

Installation instructions are found in the Aerie repository deployment documentation. If you have any questions or issues, don't hesitate to ask on #mpsa-aerie-users.

## Docker Containers

Goto the Artifactory Aerie Docker repository and log in with your JPL credentials. The latest released containers are:

```
docker-release-local/gov/nasa/jpl/aerie/merlin/release-0.9.1
docker-release-local/gov/nasa/jpl/aerie-gateway/release-0.9.1
docker-release-local/gov/nasa/jpl/aerie-ui/release-0.9.1
docker-release-local/gov/nasa/jpl/aerie-code-server/release-0.9.1
docker-release-local/gov/nasa/jpl/aerie-editor/release-0.9.1
```

## Example Docker-Compose

An example Docker Compose file is available for deployment. You can use these instructions to help you deploy.

`general-develop/gov/nasa/jpl/aerie/aerie-docker-compose.tar.gz`

## TARs

If you just want the Aerie JAR files you can find them at:

```
general/gov/nasa/jpl/aerie/aerie-release-0.9.1.tar.gz
```

The Aerie Editor (Falcon) can be found at:

```
general/gov/nasa/jpl/aerie/aerie-editor-release-0.9.1.tar.gz
```

## 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 | Port |
|---|---|
| Merlin Server | 27183 |
| Aerie UI | 80 |
| Aerie Gateway | 9000 |
| Postgres | 5432 |
| Hasura | 8080 |

# Administration

This product is using Docker containers to run the application. There are total of five Docker containers that are internally bridged (connected) to run the application. Containers can be restarted in case of any issues using Docker CLI. Only port 80 from the UI container is exposed to outside.

## 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.

# NFS Deployment

The provided `docker-compose.yml` configuration deploys all Aerie components on the same physical machine. This is sufficient for small Aerie deployments, such as for mission model development, but any running simulations will compete for system resources, potentially degrading the entire system significantly.

Other container deployment systems, like Kubernetes or AWS, can support containers deployed across multiple physical machines. However, simulations may require access to input files (such as SPICE kernels) that are managed by the Merlin API server. These files will need to be shared between physical machines.

Aerie supports the use of NFS (Network File System) to "seamlessly" share these files between multiple hosts. Below, we describe one way to deploy NFS using an additional docker-compose stack. This example is suitable for demonstration purposes, but a production deployment will vary depending on need and the environment -- we recommend consulting with an experienced system administrator for specifics.

## NFS Clients

Clients may mount the shared directory by making use of a Docker volume:

```
volumes:
  mission_model_files:
  postgres_data:
  model_data:
    driver: local
    driver_opts:
      type: nfs4
      o: addr=172.27.0.2,rw,noatime
      device: ":/"
```

In this case, the IP (`172.27.0.2`) is the static IP of the NFS server.

The volume may now be attached to an existing service's `volumes` list with:

```
volumes:
  - model_data:/usr/src/app/data
```

# Mission Model Development

Aerie provides four libraries that can be imported by a project from JPL Artifactory. They are,

- **contrib** at maven-libs-release-local/gov/nasa/jpl/aerie/contrib/0.9.1/
- **merlin-framework** at maven-libs-release-local/gov/nasa/jpl/aerie/merlin-framework/0.9.1/
- **merlin-framework-processor** at maven-libs-release-local/gov/nasa/jpl/aerie/merlin-framework-processor/0.9.1/
- **merlin-framework-junit** at maven-libs-release-local/gov/nasa/jpl/aerie/merlin-framework-junit/0.9.1/
- **merlin-sdk** at maven-libs-release-local/gov/nasa/jpl/aerie/merlin-sdk/0.9.1/
- **merlin-driver** at maven-libs-release-local/gov/nasa/jpl/aerie/merlin-driver/0.9.1/
- **parsing-tulities** at maven-libs-release-local/gov/nasa/jpl/aerie/parsin-utilities/0.9.1/

## Software Requirements

| Name | Version |
|---|---|
| *Node.js | 14.X LTS |
| *NPM | 6.X |
| *Open JDK | 17.X |

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

# Product Support

## Defect Reporting Procedure

All defect reports should go toaerie_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
- #mpsa-aerie-users: User help Slack channel