0.6.3

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, MPS Server, SLINC II / CTS, and ULSGEN.

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

References

Document	ID
Aerie Software Requirements Document (Aerie SRD)	DOC-002388
Aerie Product Guide	DOC-002537
MGSS Implementation and Task Requirements	DOC-001455
JPL Software Development Rules	57653

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 an adaptation 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 adaptation file, since activities in a plan and their parameters are validated against activity type definitions in the adaptation.

Aerie takes the adaptation and the plan file as inputs and executes a simulation. Simulation currently returns all states declared in the adaptation 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 Aerie adaptation following the instructions on Developing an Adaptation page.
3. Upload the adaptation to the adaptation service through Planning Web GUI or CLI-Command Line Interface.
4. Create a plan with the adaptation using again the Planning Web GUI or CLI
5. Trigger simulation via either interfaces listed above.

Details of each step are described in their respective pages.

Developing An Adaptation

An adaptation defines a **mission model**, giving 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 adaptation template (v0.6.1) to get started.

`package-info.java`

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

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

import gov.nasa.jpl.aerie.banananation.activities.BiteBananaActivity;
import gov.nasa.jpl.aerie.banananation.activities.ParameterTestActivity;
import gov.nasa.jpl.aerie.banananation.activities.PeelBananaActivity;
import gov.nasa.jpl.aerie.contrib.serialization.rulesets.BasicValueMappers;
import gov.nasa.jpl.aerie.merlin.framework.annotations.Adaptation;
import gov.nasa.jpl.aerie.merlin.framework.annotations.Adaptation.WithActivityType;
import gov.nasa.jpl.aerie.merlin.framework.annotations.Adaptation.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. In particular, `@WithMappers` 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. 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.

```java
// banananation/Mission.java
public class Mission extends Model {
  public final AdditiveRegister fruit;
  public final AdditiveRegister peel;
  public final Register<Flag> flag;

  public Mission(final Registrar registrar) {
    super(registrar);

    this.flag = Register.create(registrar.descend("flag"), Flag.A);
    this.peel = AdditiveRegister.create(registrar.descend("peel"), 4.0);
    this.fruit = AdditiveRegister.create(registrar.descend("fruit"), 4.0);
  }
}
```

In this case, all resources are registered by the sub-models constructed by `Mission`. In general, resources are declared using one of the overloads of `Registrar#resource`).

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 registered using the `Registrar#daemon`) 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 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
// banananation/activities/PeelBananaActivity.java
@ActivityType("PeelBanana")
public final class PeelBananaActivity {
  private static final double MASHED_BANANA_AMOUNT = 1.0;

  @Parameter
  public String peelDirection = "fromStem";

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

  public final class EffectModel extends Task {
    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 adaptation'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 an Adaptation

In order to use an adaptation 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 adaptation provides this capability out of the box, so long as your dependencies are specified with Gradle's `implementation` dependency class. The built adaptation JAR can be uploaded to Aerie through the Aerie web UI.

Activities

- To include: workarounds for activity and parameter mappers

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.

Parameters of an activity can be validated and restricted by providing one or more validation methods, designated by the `@Validation` annotation:

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

Such validation methods are picked up by the Merlin annotation processor, and whenever an instance of an activity type is created, each validation method for the type is run. If one of the validation methods returns false, the error message (provided by the annotation) is reported and activity instance creation is rejected.

Activity Effect Model

Effects of activity types must be defined in an inner `EffectModel` class. The `EffectModel` class must extend the Merlin `Task` class, and contain a `run()` method, to be called by the simulation engine when an activity instance is executed. The `run()` method should take an instance of the adaptation's `Mission` class, enabling access to resources. Extending the `Task` class allows for simple, easy calls to the following Merlin effect methods:

- `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.
- `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))`.

A full example Activity Type is given below to showcase how an effect model looks in context:

```
@ActivityType(name="RunHeater", generateMapper=true)
public class RunHeater {

    private static final int energyConsumptionRate = 1000;

    @Parameter
    public long durationInSeconds;

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

    public final class EffectModel extends Task {
        public void run(final Mission mission) {
        final double totalEnergyUsed = durationInSeconds * energyConsumptionRate;

        spawn(new PowerOnHeater());
        mission.batteryCapacity.use(totalEnergyUsed);

        delay(durationInSeconds, Duration.SECONDS);

        call(new PowerOffHeater());
    }
}
```

This activity first places a `PowerOnHeater` activity at the start of the activity. Next the total energy used by running a heater for a parameterized duration is subtracted from the batteryCapacity state (see Models and Resources for more information). Next the activity pauses until the duration has been reached before spawning a `PowerOffHeater` activity, after which the activity will complete execution.

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 adaptation 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:

```java
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:

```
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 thecontrib 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 inte
gers and floats can be represented by `long` and `double` respectively. For more complex types, you can also provide a `List<SerializedValue>` or `Map<S
tring, 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");
  }

  public final class EffectModel extends Task {
    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<>(
    registrar.descend("batterySoC"),
    () -> this.source.volume.get() - this.sink.volume.get(),
    0.0,
    new DoubleValueMapper());
```

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

Custom models

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

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

The `contrib` package is a rich source of example models. See the repository for more details.

Creating Plans

Plans can be created via Planning Web GUI, 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

-- Rewriting in progress --

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 adaptation-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 Command Line Interface

The Aerie command line interface (CLI) for planning allows manipulation of plans in batch using the commands listed below. In order to use the CLI users have must run Aerie on their localhost, where the CLI will reference Aerie interface end-points that are hard-coded in an Aerie config file. In the future, the CLI can be configured to point to any host machine to make requests against. At that point, Aerie services can run anywhere, and the CLI will work from any other machine as long as it has the proper host configured. Example input/output JSON is given in the appendix for reference.

Installation

The CLI is contained in a JAR file that can be downloaded from Artifactory. To install it, download the Aerie tar file. Note that you will need to log in to Artifactory to reach the download link. Decompress the file to your desired location. This can be done manually by going to your desired location in a terminal and entering

```
tar -xzf ~/Downloads/aerie-release-X.X.X.tar.gz
```

The JAR will now be located at `<installation-path>/merlin-cli/merlin-cli.jar`. The CLI can now be run by executing

```
java -jar <installation-path>/merlin-cli/merlin-cli.jar
```

but this is cumbersome, so we suggest setting up an alias to run the CLI. For shells such as Bash and Zsh (the default shells for recent Macintosh computers), this can be done as such:

```
alias merlin-cli="java -jar <installation-path>/merlin-cli/merlin-cli.jar"
```

At this point the CLI can be run by entering `merlin-cli` at the command prompt, however if a new terminal is opened the alias will need to be set up again. To avoid this, add the alias to your rc file, so it will be initialized every time your terminal starts up.

Below is an example of how to install the merlin CLI to `~/opt` after downloading the .tar.gz file in zsh:

```
cd ~/opt
tar -xzf ~/Downloads/aerie-release-X.X.X.tar.gz
alias merlin-cli="java -jar ~/opt/merlin-cli/merlin-cli.jar"
```

Known Issues

It is possible when running the JAR file you will see something like the following error:

```
Error: A JNI error has occurred, please check your installation and try again
Exception in thread "main" java.lang.UnsupportedClassVersionError: gov/nasa/jpl/ammos/mpsa/aerie/merlincli/MainCLI
has been compiled by a more recent version of the Java Runtime (class file version 55.0), this version of the Java
Runtime only recognizes class file versions up to 52.0
```

If this error occurs, the cause is likely due to a mismatch in the version of Java being run and the version the JAR was compiled with. The CLI uses Java 11, so Java 11 should be used to run the CLI. Your java version can be checked by entering `java -version` at the command prompt.

Usage

The CLI is currently designed to automatically connect to a local deployment of the Aerie services (see here). In the future, the CLI will be configurable to connect to remote services, but at this time only use with a local deployment is possible.

Create a single adaptation

Create an adaptation from a JAR file and metadata, returns an adaptation ID.

```
merlin-cli --create-adaptation <path-to-adaptation-jar> name=<name> version=<version> mission=<mission> owner=<owner>
```

Query all adaptations

Returns a list of all adaptations indexed by adaptation ID.

```
merlin-cli --list-adaptations
```

Query adaptation metadata

Returns metadata of an adaptation specified with ID.

```
merlin-cli -a <adaptation-id> --view-adaptation
```

Query all activity types in an adaptation

Returns a list of activity types indexed by activity type name. For each activity type, parameter names and types as well as default values are given.

```
merlin-cli -a <adaptation-id> --activity-types
```

Query an activity type in an adaptation

Returns a list of parameters as well as their default values for an activity type specified by ID.

```
merlin-cli -a <adaptation-id> --activity-type <activity-type-id>
```

Query activity type parameters

Returns a list of activity type parameters for a given activity type within an adaptation, both specified by ID.

```
merlin-cli -a <adaptation-id> --activity-type-parameters <activity-type-id>
```

Delete an adaptation

Remove an adaptation specified by ID from the adaptation service.

```
merlin-cli -a <adaptation-id> --delete-adaptation
```

Create a plan

Create a plan by uploading a plan JSON file. The JSON should contain all required fields including adaptation ID, plan name, start timestamp and end timestamp (YYYY-DDDThh:mm:ss). Adaptation ID must be for a valid adaptation, and activity instances must be valid according to the adaptation.

```
merlin-cli --create-plan <path-to-plan-json>
```

Query plans

Returns a list of existing plans indexed by plan ID.

```
merlin-cli --list-plans
```

Query an activity instance

Given a plan and activity instance ID, return activity instance metadata and parameter list.

```
merlin-cli -p <plan-id> --display-activity <activity-instance-id>
```

Update plan metadata

Update plan metadata via key/value pairs. Valid fields are: `adaptationId`, `startTimestamp`, `endTimestamp`, `name`.

```
merlin-cli -p <plan-id> --update-plan <field>=<value>
```

Update an activity instance

Update any attribute of an activity instance specified by unique ID. Note that activity instance parameters cannot be updated this way. Valid fields are: `startTimestamp` and `name`.

```
merlin-cli -p <plan-id> --update-activity <field>=<value>
```

Delete an activity instance

Delete an activity instance from a plan by ID.

```
merlin-cli -p <plan-id> --delete-activity <activity-instance-id>
```

Delete a plan

Delete a plan specified by ID.

```
merlin-cli -p <plan-id> --delete-plan
```

Update plan from a file

Upload a JSON file containing fields of a plan to be updated. All fields except adaptation ID may be updated this way. Any fields that are left out will remain unchanged. Note that individual activity instances cannot be updated this way, and including any activity instances in the plan update JSON will replace the current activity instance set with them.

```
merlin-cli -p <plan-id> --update-plan-from-file <path-to-json>
```

Append activity instances to a plan by file upload

This action will retain activity instances in the plan, but add all new activity instances specified by a JSON file.

```
merlin-cli -p <plan-id> --append-activities <path-to-json>
```

Download a plan JSON

Download plan in a JSON file format. Note that a downloaded plan JSON is different from the format for creating a plan in that activity instances are given as a map indexed by activity ID instead of a list.

```
merlin-cli -p <plan-id> --download-plan <output-path>
```

Simulate a plan

Kick off a simulation of a plan by ID, with results written to a file. Takes a sampling period (in microseconds) to determine how often to sample states for simulation results. Note that the local version of this command currently ignores the output path, and instead prints results to the console.

```
merlin-cli -p <plan-id> --simulate <sampling-period> <output-path>
```

Convert an APGen APF file to a Merlin JSON Plan file

APF files can be converted to Merlin input JSON file format. The action requires specifying a path to the APGEN adaptation `.aaf` files.

```
merlin-cli --convert-apf <path-to-apf-file> <output-path> <path-to-apgen-adaptation-directory>
```

Aerie GraphQL API

Purpose

This document describes the Aerie GraphQL API, software interface provided by the Aerie 0.6.2 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 clients 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 addition of fields to 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 visibility for auditing the frequency and combinations with which certain fields are referenced by a client’s queries and mutations. This provides Aerie development verifiable evidence for the frequency of use and thus more informed decision process when 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 GET or POST
3. Receiving the result as JSON

GET request

When making an HTTP GET request, the GraphQL query should be specified in the "query" query string. For example, if an application executes the following GraphQL query to retrieve all the id for all activity plans:

```
{
  plans { id }
}
```

This request could be sent via an HTTP GET like so:

```
https://<your_domain>:27184?query={plans{id}}
```

Query variables can be sent as a JSON-encoded string in an additional query parameter called `variables`. If the query contains several named operations, an `operationName` query parameter can be used to control which one should be executed.

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 be present on the response. If no data is returned, the `"data"` field is only be included if the error occurred during execution.

Methods of Querying 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 or GET request to be made against the API. For example, this can be done from the command line with `curl` like so:

```
curl -g 'https://aerie-develop.jpl.nasa.gov:27184?query={plans{id}}'
-H 'Accept-Encoding: gzip, deflate, br'
-H 'Authorization: SSO_COOKIE_VALUE_HERE'
-H 'Content-Type: application/json'
-H 'Accept: application/json'
-H 'Connection: keep-alive'
-H 'DNT: 1'
-H 'Origin: https://aerie-develop.jpl.nasa.gov:27184'
--data-binary '{"query":"{plans{id}}"}'
--compressed
```

With a JSON formatted result returned as such:

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

GraphQL Playground

The GraphQL API is described by a schema which describes 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 following query to obtain your `ssoCookieValue`:

```
mutation Login($username: String!, $password: String!) {
  login(username: $username, password: $password) {
    ssoCookieValue
  }
}
```

Next, take the `ssoCookieValue` to the `HTTP HEADERS` section and add it in the `authorization` header:

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

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

```
query Adaptations {
  adaptations {
    name
  }
}
```

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

Browser Developer Console

Requests can also be tested from the browser. Navigating to https://<your_domain>:27184, 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:  "{plans{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 underlying transport layer and later switch to a client library as the application becomes more complex.

GraphQL clients exist for the following programming languages;

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

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

Aerie Queries

It is important to understand the significance and power of a data graph based API. The following queries are examples of what Aerie refer to as ‘canonical queries’ because they map to commonly discussed use cases and data structures for subsystems within a mission.

The GraphQL syntax is simple and a small primer sufficient to work with the following section’s is found at https://graphql.org/learn/schema/

Query Schema for Plan

```
type Query {
  ...
  plan(id: ID!): Plan
  plans: [Plan]!
  ...
}
```

Plan Schema

```
type Plan {
  activityInstances: [ActivityInstance]!
  adaptation: Adaptation
  adaptationId: String!
  startTimestamp: String!
  endTimestamp: String!
  id: ID!
  name: String!
}
```

Activity Instance's Schema

```
type ActivityInstance {
  id: ID!
  type: String!
  parameters: [ActivityInstanceParameter]!

  startTimestamp: String!
  duration: Float

  parent: String
  children: [String!]
}
type ActivityInstanceParameter {
  name: String!
  value: ActivityInstanceParameterValue!
}
```

Adaptation's Schema

```
type Adaptation {
  activityType(name: String!): ActivityType
  activityTypes: [ActivityType]
  id: ID!
  mission: String!
  name: String!
  owner: String!
  version: String!
}
```

Activity type's Schema

```
type ActivityType {
  name: String!
  parameter(name: String!): ActivityTypeParameter
  parameters: [ActivityTypeParameter]!
}
type ActivityTypeParameter {
  default: ActivityTypeParameterDefault
  name: String!
  schema: ActivityTypeParameterSchema
}
```

Query Schema for Simulation

```
type  Query {
  simulate(planId: String!, samplingPeriod: Float!): SimulationResponse
}
```

Simulation Response & Result's Schema

```
type SimulationResponse {
  message: String
  results: [SimulationResult!]
  success: Boolean!
  violations: [Violation!]
}
type SimulationResult {
  name: String!
  start: String!
  values: [SimulationResultValue!]!
}
type SimulationResultValue {
  x: Float!
  y: SimulationResultValueY!
}
```

Mutation Schema

Mutation schema for creating an activity instances

```
type Mutation{
  createActivityInstances(
    activityInstances: [CreateActivityInstance]!
    planId: ID!
  ): CreateActivityInstancesResponse
  ...
}
```

Mutation schema for creating an adaptation

```
type Mutation{
  ...
  createAdaptation(
    file: Upload!
    mission: String!
    name: String!
    owner: String!
    version: String!
  ): CreateAdaptationResponse
}
```

Mutation schema for creating a plan

```
type Mutation{
  ...
  createPlan(
    adaptationId: String!
    endTimestamp: String!
    name: String!
    startTimestamp: String!
  ): CreatePlanResponse
}
```

Mutation schema for update an activity instance

```
type Mutation{
  ...
  updateActivityInstance(
    activityInstance: UpdateActivityInstance!
    planId: ID!
  ): Response
}
```

Mutation schema for deleting an activity instance, adaptation and plan

```
type Mutation {
  ...
  deleteActivityInstance(planId: ID!, activityInstanceId: ID!): Response
  deleteAdaptation(id: ID!): Response
  deletePlan(id: ID!): Response
}
```

Typical Usages

When writing a GraphQL query, refer to the schema for all valid fields that one can specify in a particular query.

Query all plans

Returns metadata for all Plans

```
query {
  plans{
    id
    name
    startTimestamp
    endTimestamp
    adaptationId
  }
}
```

Query a single plan

Using the id you got from "Query all plans" to obtain information for a single plan.

Returns metadata of a single Plan

```
query {
  plan(id: "5f492c60ae0dec17320e5bed") {
    id
    name
    startTimestamp
    endTimestamp
    adaptationId
  }
}
```

Query all activity instances from a plan

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

Returns activity instances metadata and parameter list

```
query {
  plan(id: "5f492c60ae0dec17320e5bed") {
    activityInstances{
      id
      startTimestamp
      parameters {
        name
        value
      }
    }
  }
}
```

Query adaptation from a plan

Returns metadata of an adaptation from a particular plan

```
query {
  plan(id: "5f492c60ae0dec17320e5bed") {
    adaptation {
      id
      mission
      name
      owner
      version
    }
  }
}
```

Query activity types within an adaptation from a plan

Returns a list of activity types. For each activity type, name and parameter list are given

```
query {
  plan(id: "5f492c60ae0dec17320e5bed") {
    adaptation {
      activityTypes{
        name
        parameters{
          name
          schema
        }
      }
    }
  }
}
```

Run simulation and query the result

You have to provide the planId and the simpling rate in order to run the simulation.

Returns a list of states with different sampling time.

```
query {
  simulate (planId: "5f492c60ae0dec17320e5bed", samplingPeriod: 10000000000 ) {
    message
    success
    results {
      name
      start
      values{
        x
        y
      }
    }
  }
}
```

Creating a activity instances

```
mutation CreateActivityInstances {
  createActivityInstances(
    planId: "5ff350fc63ee884f69adea27"
    activityInstances: [
      {
        parameters: []
        startTimestamp: "2020-001T00:00:01"
        type: "PeelBanana"
      }
      {
        parameters: [{ name: "peelDirection", value: "fromTip" }]
        startTimestamp: "2020-001T00:00:03"
        type: "PeelBanana"
      }
    ]
  ) {
    ids
    message
    success
  }
}
```

Creating a adaptation

Note: The type "Upload" is a scalar in the schema. It is up to the server data source to interpret the file properly and forward the request to the services. References: MDN Web Docs and Apollo GraphQL Docs.

```
mutation {
  createAdaptation(file: <Upload>, mission: "EUROPA", name: "Test", owner: "Test", version: "0.1") {
    message
    success
  }
}
```

Creating a plan

```
mutation {
  createPlan(adaptationId: "5f492c3b8d1d733cf46f498e",
    startTimestamp:"2020-001T00:11:11.123",
    endTimestamp: "2020-001T11:11:11.123",
    name: "graphql",)
  {
    message
    success
  }
}
```

Deleting an activity instance

```
mutation {
  deleteActivityInstance(planId:"5f492c60ae0dec17320e5bed", activityInstanceId:"5f5d0601a88a0b4299b501c9")
  {
    message
    success
  }
}
```

Deleting an adaptation

```
mutation {
  deleteAdaptation(id:"5f492c3b8d1d733cf46f498e") {
    message
    success
  }
}
```

Deleting a plan

```
mutation {
  deletePlan(id:"5f5cef46a88a0b4299b501c7") {
    message
    success
  }
}
```

Updating an activity instance

```
mutation {
  updateActivityInstance(activityInstance: {
    id:"5f4e7c596d2c237b61fca4c6",
    parameters:{name:""},
    startTimestamp:"2020-116T00:00:00",
    type: "TurnInstrumentOff"},
    planId:"5f492c60ae0dec17320e5bed" )
  {
    success
    message
  }
}
```

Schema Version 0.6.2

The Aerie GraphQL as copied at the time of the Aerie 0.6.2 release.

```
scalar ActivityInstanceParameterValue
scalar ActivityTypeParameterDefault
scalar ActivityTypeParameterSchema
scalar SimulationResultValueY
scalar StateSchema
scalar UiPanel

type ActivityInstance {
  children: [String!]
  duration: Float
  id: ID!
  parameters: [ActivityInstanceParameter]!
  parent: String
  startTimestamp: String!
  type: String!
}
```

```
type ActivityInstanceParameter {
  name: String!
  value: ActivityInstanceParameterValue
}

input ActivityInstanceParameterInput {
  name: String!
  value: ActivityInstanceParameterValue
}

type ActivityType {
  name: String!
  parameter(name: String!): ActivityTypeParameter
  parameters: [ActivityTypeParameter]!
}

type ActivityTypeParameter {
  default: ActivityTypeParameterDefault
  name: String!
  schema: ActivityTypeParameterSchema
}

type Adaptation {
  activityType(name: String!): ActivityType
  activityTypes: [ActivityType]
  id: ID!
  mission: String!
  name: String!
  owner: String!
  version: String!
}

type Constraint {
  category: String!
  message: String!
  name: String!
}

type ConstraintViolation {
  associations: ConstraintViolationAssociations!
  constraint: Constraint!
  windows: [TimeRange!]!
}

type ConstraintViolationAssociations {
  activityInstanceIds: [String!]!
  stateIds: [String!]!
}

input CreateActivityInstance {
  parameters: [ActivityInstanceParameterInput!]!
  startTimestamp: String!
  type: String!
}

type CreateActivityInstancesResponse {
  ids: [ID]!
  message: String
  success: Boolean!
}

type CreateAdaptationResponse {
  id: ID
  message: String
  success: Boolean!
}

type CreatePlanResponse {
  id: ID
  message: String
  success: Boolean!
}

type LoginResponse {
  editorUrl: String
  lbCookieName: String
  lbCookieValue: String
  message: String!
  ssoCookieName: String
  ssoCookieValue: String
  success: Boolean!
}

type Mutation {
  createActivityInstances(
    activityInstances: [CreateActivityInstance]!
    planId: ID!
```

```
    planId: ID!
  ): CreateActivityInstancesResponse
  createAdaptation(
    file: Upload!
    mission: String!
    name: String!
    owner: String!
    version: String!
  ): CreateAdaptationResponse
  createPlan(
    adaptationId: String!
    endTimestamp: String!
    name: String!
    startTimestamp: String!
  ): CreatePlanResponse
  deleteActivityInstance(planId: ID!, activityInstanceId: ID!): Response
  deleteAdaptation(id: ID!): Response
  deletePlan(id: ID!): Response
  login(username: String!, password: String!): LoginResponse
  logout: Response
  updateActivityInstance(
    activityInstance: UpdateActivityInstance!
    planId: ID!
  ): Response
}

type Plan {
  activityInstances: [ActivityInstance]!
  adaptation: Adaptation
  adaptationId: String!
  endTimestamp: String!
  id: ID!
  name: String!
  startTimestamp: String!
}

type Query {
  activityType(adaptationId: ID!, name: String!): ActivityType
  activityTypes(adaptationId: ID!): [ActivityType]!
  adaptation(id: ID!): Adaptation
  adaptations: [Adaptation]!
  plan(id: ID!): Plan
  plans: [Plan]!
  session: Response
  simulate(
    adaptationId: String!
    planId: String!
    samplingPeriod: Float!
  ): SimulationResponse
  stateTypes(adaptationId: ID!): [StateType!]!
  uiStates: [UiState!]!
  user: UserResponse
  validateParameters(
    activityTypeName: String!
    adaptationId: ID!
    parameters: [ActivityInstanceParameterInput!]!
  ): ValidationResponse
}

type Response {
  message: String
  success: Boolean!
}

type SimulationResponse {
  activities: [ActivityInstance!]
  message: String
  results: [SimulationResult!]
  success: Boolean!
  violations: [ConstraintViolation!]
}

type SimulationResult {
  name: String!
  schema: StateSchema!
  start: String!
  values: [SimulationResultValue!]!
}

type SimulationResultValue {
  x: Float!
  y: SimulationResultValueY!
}

type StateType {
  name: String!
  schema: StateSchema!
}
```

```
}

type TimeRange {
  end: Float!
  start: Float!
}

type UiState {
  id: ID!
  name: String
  panels: [UiPanel!]!
}

input UpdateActivityInstance {
  id: ID!
  parameters: [ActivityInstanceParameterInput!]
  startTimestamp: String
  type: String
}

type UserResponse {
  editorUrl: String
  filteredGroupList: [String!]
  fullName: String
  groupList: [String!]
  message: String!
  success: Boolean!
  userId: String
}

type ValidationResponse {
  errors: [String!]
  success: Boolean!
}
```

Aerie Planning UI

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

Uploading Adaptations

Adaptations can be uploaded to Aerie via the UI. To navigate to theadaptations page, click the `Adaptations` icon on the on the left navigation bar. Once an adaptation JAR is prepared, it can be uploaded to the adaptation 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 adaptation.

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

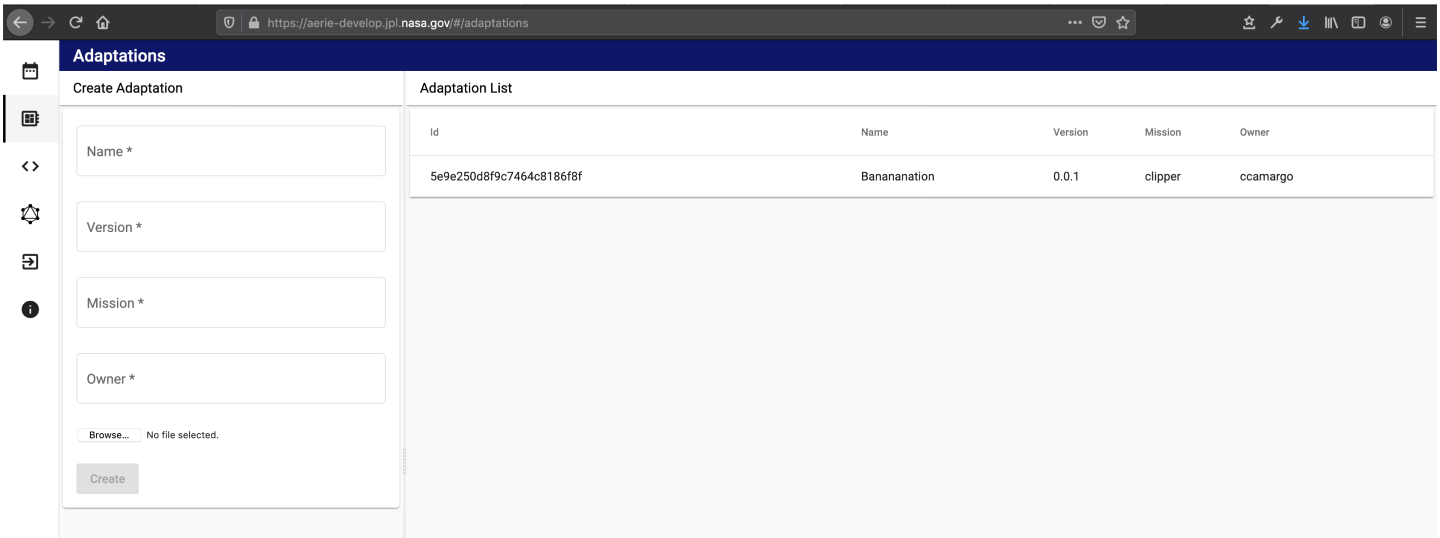


Figure 1: Upload adaptations, and view existing adaptations.

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 adaptation in the adaptation service. 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.

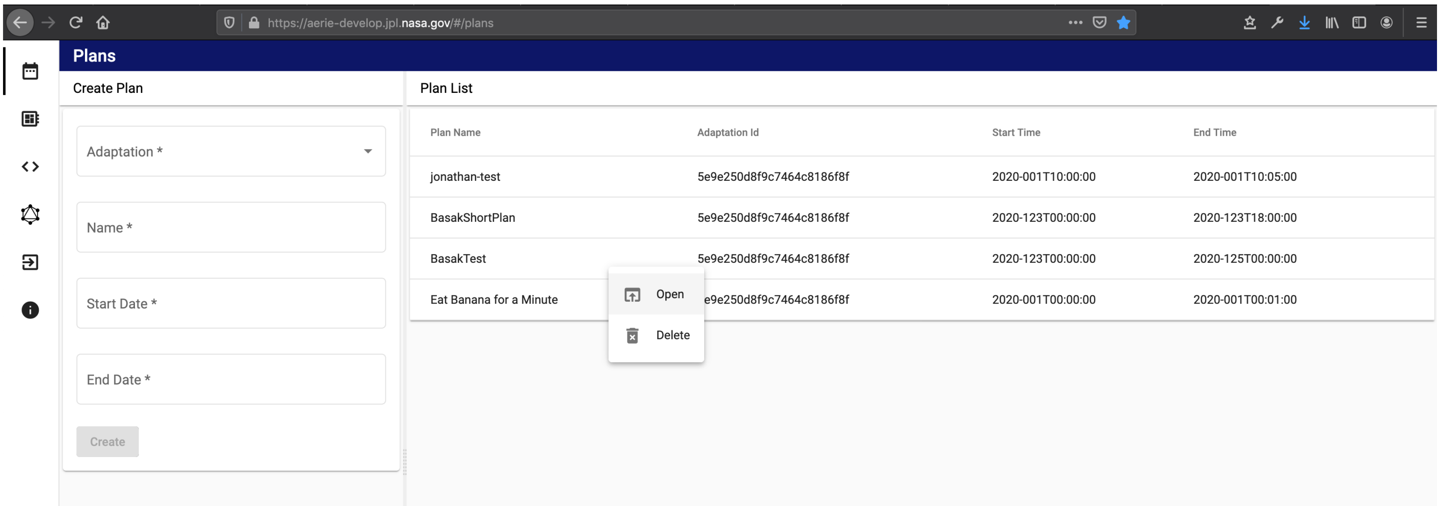


Figure 2: Create plans, and view existing plans.

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.

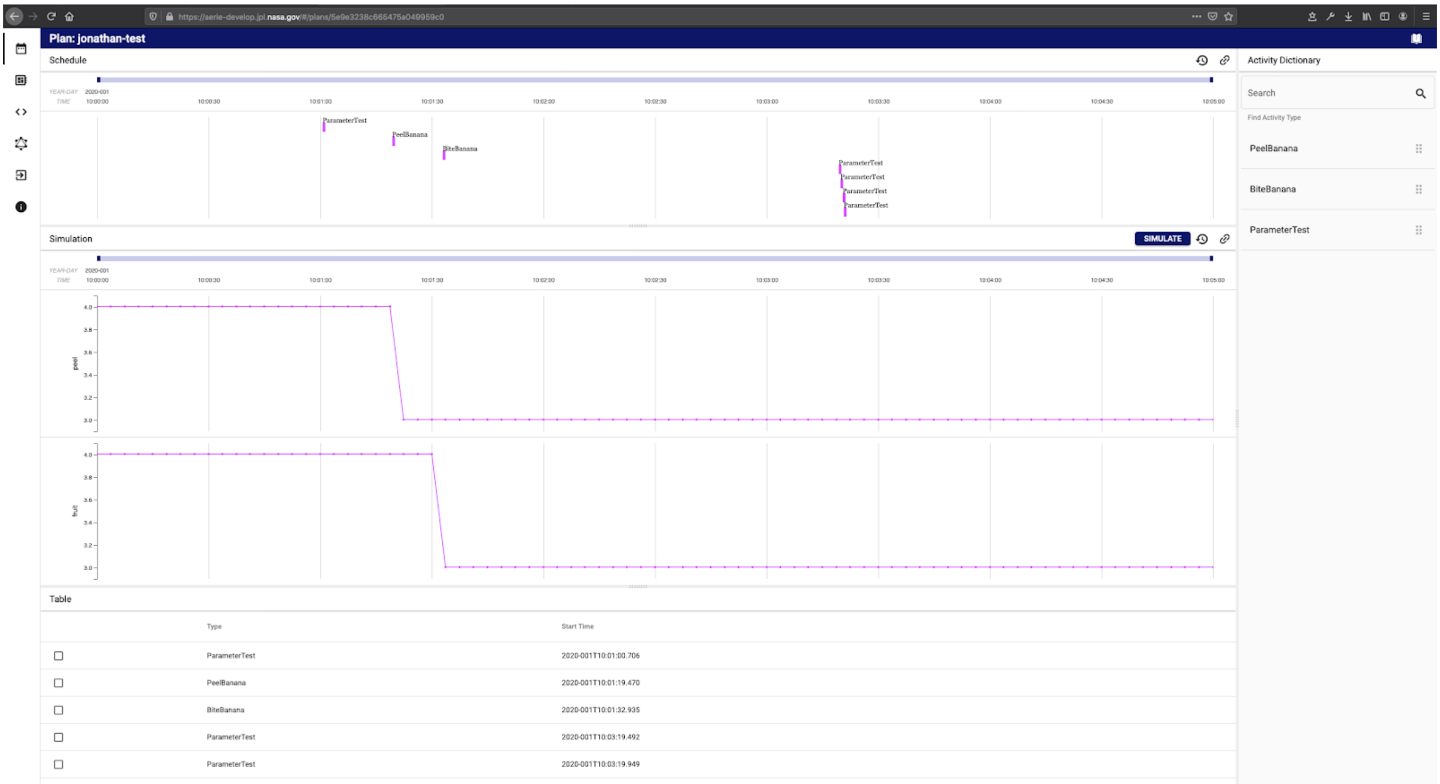


Figure 3: Default panels.

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.

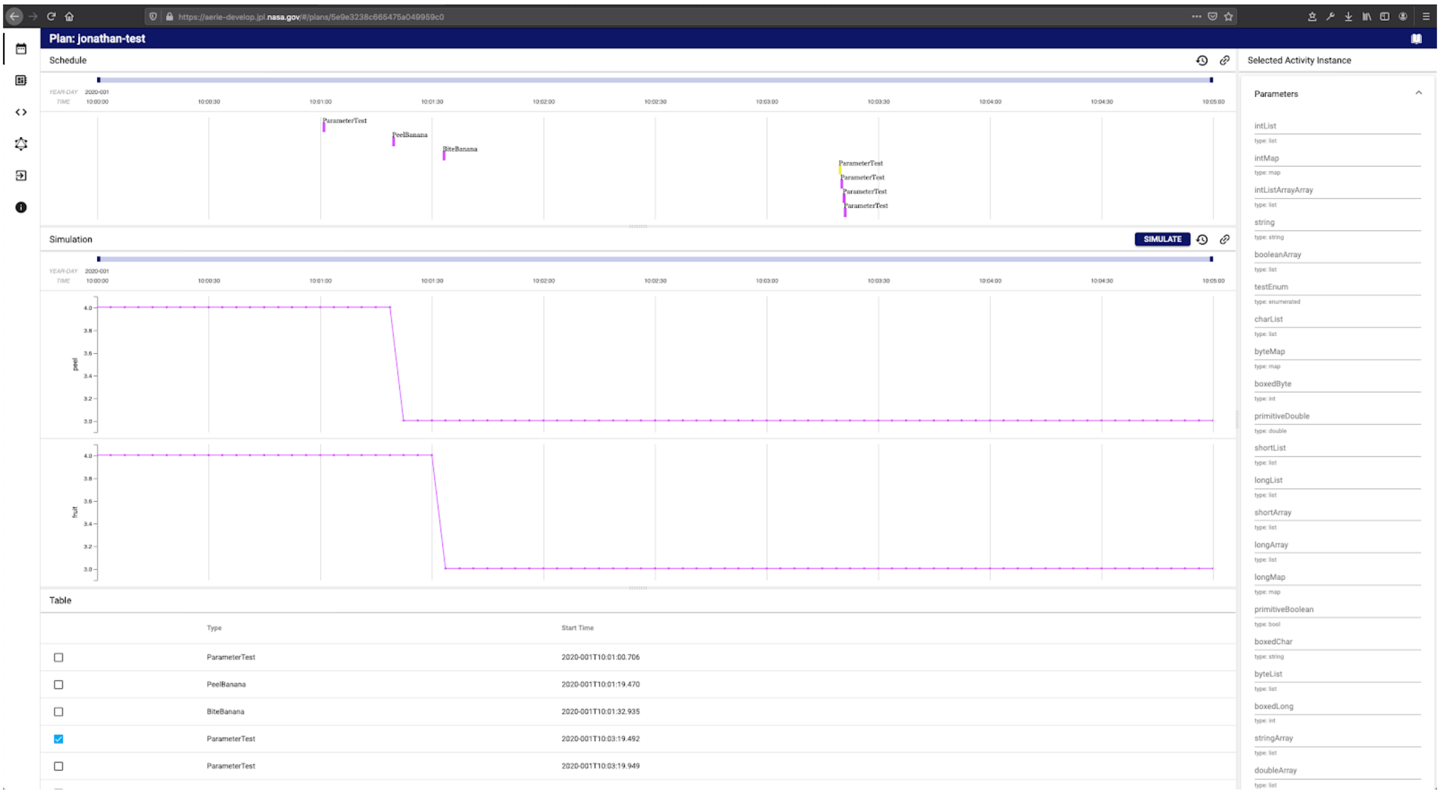


Figure 4: When an activity instance in plan is selected, its details will appear in the right drawer.

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

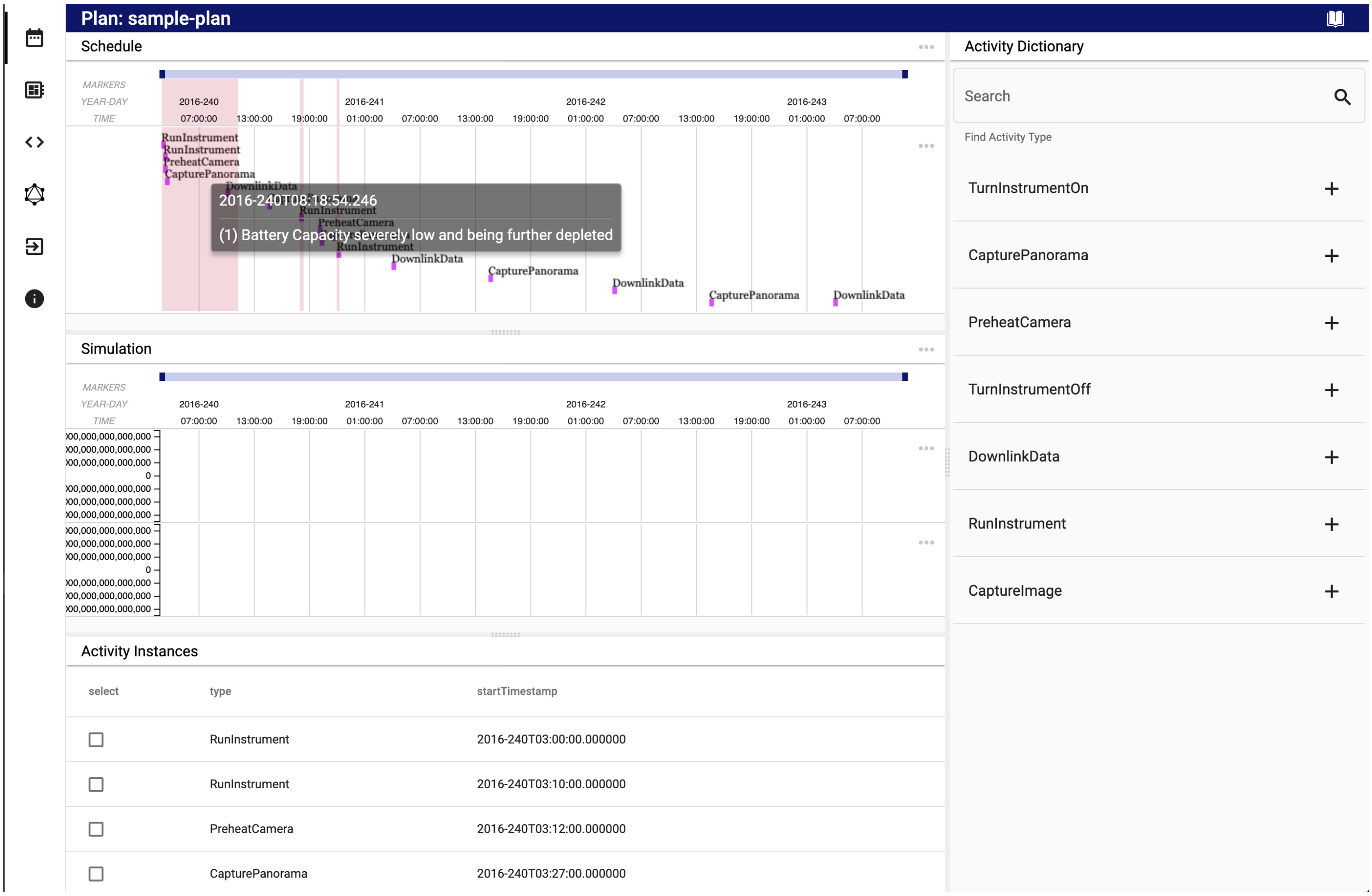


Figure 4.1: When violations occur, they will be represented as red areas on the timeline, with an accompanying hover state.

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.

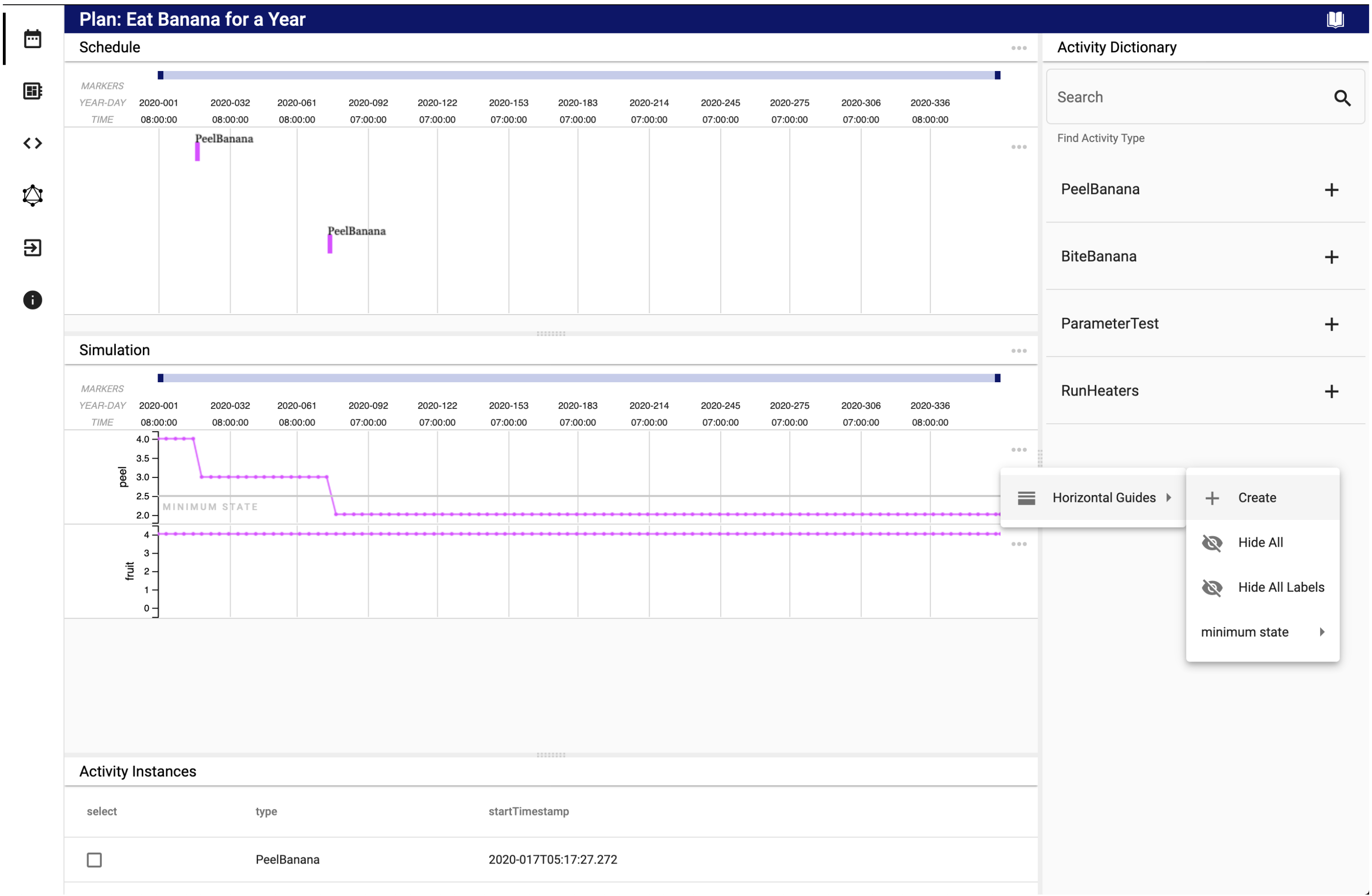


*Figure 4.2: Horizontal guides may be added to each activity schedule or simulation bands

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.

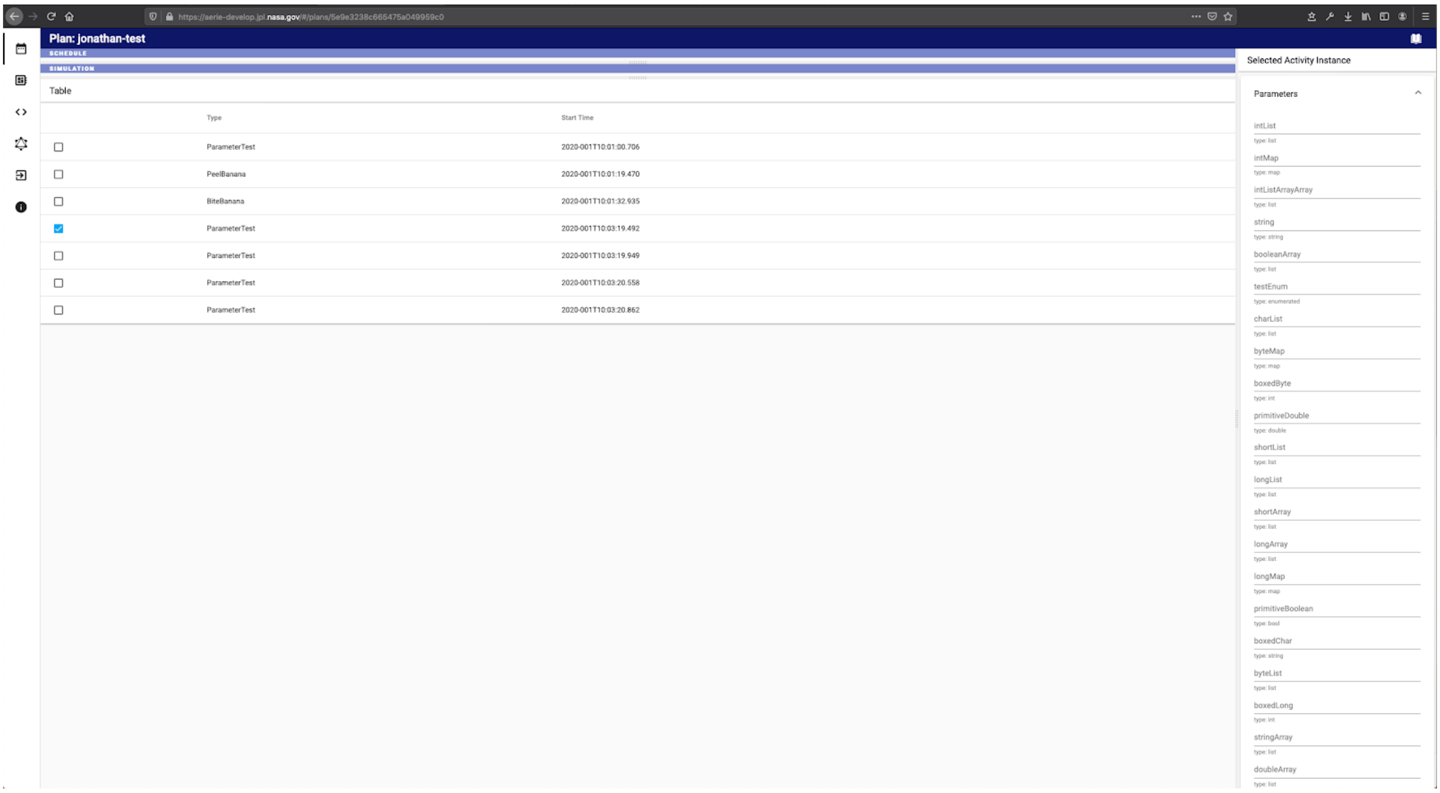


Figure 5: The bottom panels are dragged to the top edge completely leaving only one panels in view.

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.

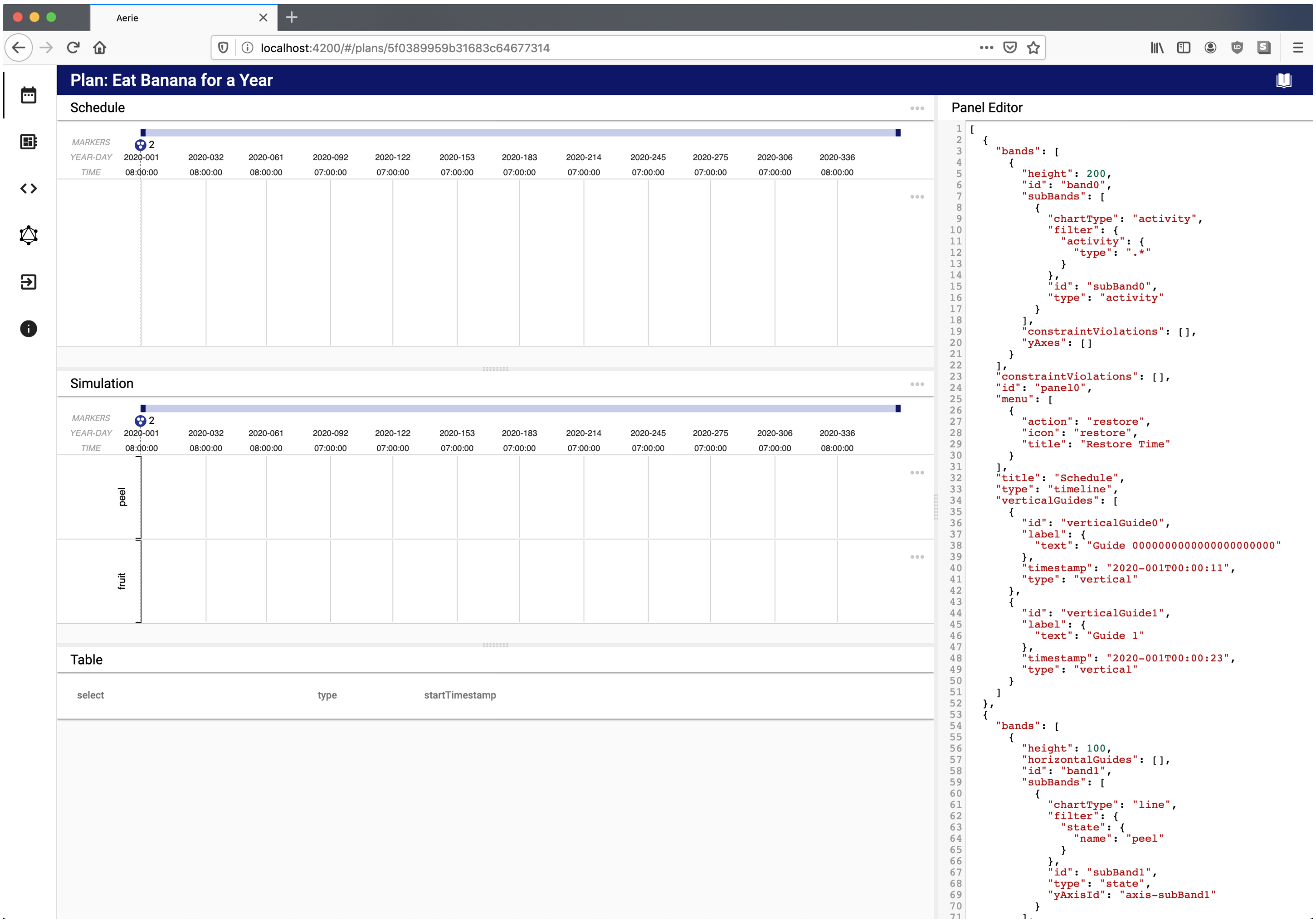
UI Configurability

Users can create custom planning views for different sub-systems (e.g. science, engineering, thermal, etc.), where only data (e.g. activities and states) 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.

Panel Editor

In order to change the configuration object, navigate to the plan page and press `⌘e (command e)`, which should open up the Panel Editor in the right drawer. Make your changes to the JSON object and press `⌘s (command s)`. You should see the UI update with your changes as long as your JSON object has no errors. Check the console if you suspect your JSON object has errors. If your errors persist, don't hesitate to ask on #mpsa-aerie-users.

Note that edited configurations are not currently saved between browser refreshes. This will be added in future releases.



Panels

The planning UI consists of a list of panels, and each `Panel` has the following interface:

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

Menu

Each panel 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 panel 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 state bands with simulation result states 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 PanelMenuItemAction = 'link' | 'restore' | 'simulate';

export interface PanelMenuItem {
  action: PanelMenuItemAction;
  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 panels: iframe, table, and timeline. The following sections will detail how to create each of these panel types.

Inline Frame (iframe)

The iframe panel allows you to embed a custom HTML page inside of the panel. To create an iframe panel 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 panel. Here is a basic example of specifying an iframe panel:

```
{
  "id": "panel3",
  "iframe": {
    "src": "https://www.chartjs.org/samples/latest/charts/line/basic.html"
  },
  "size": 100,
  "title": "Line Chart",
  "type": "iframe"
}
```

Table

The table panel allows you to view data as a table with columns and rows. You can currently only create table panels for activity data. To create a table panel 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 panel:

```
{
  "id": "panel2",
  "size": 100,
  "table": {
    "columns": [
      "select",
      "type",
      "startTimestamp"
    ],
    "type": "activity"
  },
  "title": "Activity Table",
  "type": "table"
}
```

Timeline

The timeline panel allows you to specify visualizations of time-ordered data. To create a timeline panel you need to specify `type: "timeline"`, a unique `id`, a `size`, a `title`, and a list of `bands`. Here is the minimum JSON needed to specify a timeline panel:

```
{
  "bands": [],
  "id": "panel0",
  "size": 100,
  "title": "My First Panel",
  "type": "timeline"
}
```

To visualize data in a timeline you need to add band objects to the `bands` array. A band is a layered visualization of time-ordered data. Each layer of a band is specified as an object of the `subBands` array. The interfaces for a `Band` and `SubBand` are as follows:

```
interface Band {
  height?: number;
  id: string;
  subBands: SubBand[];
  yAxes?: Axis[];
}

interface SubBand {
  chartType: 'activity' | 'line' | 'x-range'; // How we actually visualize the data
  color?: string; // Color of all data points
  filter?: {
    activity?: {
      type?: string; // JS regex filtering the type of activity we want to see
    };
    state?: {
      name?: string; // JS regex filtering the name of state we want to see
    };
  };
  id: string;
  layout: 'compact' | 'decomposition';
  type: 'activity' | 'state'; // Type of data the sub-band visualizes
  yAxisId?: string; // Optional y-axis ID link
}
```

Here is a JSON object that creates a single band with one activity sub-band. Notice there are no `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": "band0",
  "subBands": [
    {
      "chartType": "activity",
      "filter": {
        "activity": {
          "type": ".*"
        }
      },
      "id": "subBand0",
      "type": "activity"
    }
  ]
}
```

For data that has y-values (for example state data), you can specify a y-axis and link a sub-band 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 band separately from sub-bands so we can specify multi-way relationships between axes and sub-bands. For example you could have many sub-bands corresponding to a single axis.

Here is the JSON for creating a band with two overlayed `state` sub-bands. The first sub-band shows only states with the name `peel`, and uses the y-axis with ID `yAxis1`. The second sub-band shows only states with the name `fruit`, and uses the y-axis with the ID `yAxis2`.

```json
{
  "id": "band1",
  "subBands": [
    {
      "chartType": "line",
      "filter": {
        "state": {
          "name": "peel"
        }
      },
      "id": "subBand1",
      "type": "state",
      "yAxisId": "yAxis1"
    },
    {
      "chartType": "line",
      "filter": {
        "state": {
          "name": "fruit"
        }
      },
      "id": "subBand2",
      "type": "state",
      "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",
   "adaptationId": "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)


```
{
   "adaptationId": "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 ADAPTATION OUTPUT SAMPLE

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

Product Guide

- Product Installation
- System Requirements
- Administration
- Product Support

Product Installation

Installation Instructions

Installation instructions are found in the Aerie repositorydeployment 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/adaptation/release-0.5.0
docker-release-local/gov/nasa/jpl/aerie/plan/release-0.5.0
docker-release-local/gov/nasa/jpl/aerie-apollo/release-0.5.0
docker-release-local/gov/nasa/jpl/aerie-ui/release-0.5.0
```

Example Docker-Compose

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

```
## TARs

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

general/gov/nasa/jpl/aerie/aerie-release-0.5.0.tar.gz

```
The Aerie Editor (Falcon) can be found at:
```

general/gov/nasa/jpl/aerie/aerie-editor-release-0.5.0.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.

System Requirements

Software Requirements

Name	Version
DOCKER	19.X
*NODEJS	12.X LTS
*NPM	6.X
*OPEN JDK	11.X

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

Supported Browsers

Name	Version
CHROME	LATEST
FIREFOX	LATEST

Hardware Requirements

Hardware	Details
CPU	2 GIGAHERTZ (GHZ) FREQUENCY OR ABOVE
RAM	4 GB AT MINIMUM
DISPLAY RESOLUTION	2560-BY-1600, RECOMMENDED
INTERNET CONNECTION	HIGH-SPEED CONNECTION, AT LEAST 10MBPS

TCP Port Requirements

Service	Port
Aerie UI	8080
Adapataion	27182
Plan	27183
RabittMQ	15762
Aerie Apollo	27184

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 8080 from the UI container is exposed to outside.

Environment Variables

Aerie software does not have any environment variables at this point in time.

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 (8080). 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. An example of this would be ports: ["8080:80"]. The number that needs to be changed is the first port which is 8080.

Environment Variables

Port Type	Default Port Number	Description
TCP	8080	UI Port

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-Apollo: The GraphQL Apollo Server which functions as the Aerie API Gateway against which clients can submit GraphQL queries and mutations.
- Adaptation: Handles all the logic and functionality for the model Adaptation for activity planning.
- Adaptation-mongo: Holds the data for Adaptation container.
- Plan: Handles all the logic and functionality for activity planning.
- Plan-mongo: Holds the data for Plan container.
- RabbitMQ: The Aerie services' message bus message exchange instance.

The Adaptation, Plan, and Apollo servers communicate to each other via a REST API (internal to Aerie) with the ports specified in the docker-compose file. The database containers, Adaptation-mongo and Plan-mongo are isolated to connect only with their respective service container (Adaptation or Plan).

Product Support

Defect Reporting Procedure

All defect reports should go toaerie_support@jpl.nasa.gov.

Points of Contact

- Adaptation: 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