- To include: Documentation on `SerializedParameter` for assistance with creating custom Activity Mappers. We should expand on `ParameterSchema` as well, though we will likely want to change the way we approach the topic once we add support for generating Activity Mappers that use custom Parameter Mappers.

What is an Activity Mapper

An Activity Mapper is a Java class that both implements the `ActivityMapper` interface, and contains the `ActivitiesMapped` annotation. It is required that each Activity Type in an adaptation have an associated Activity Mapper, to provide provide three capabilities:

- Describe the structure of an activity type
- Serialize instances of associated Activity Types
- Deserialize instances of associated Activity Types

`@ActivitiesMapped` Annotation

The `@ActivitiesMapped` annotation tells what Activity Types the associated Activity Mapper provides the aforementioned capabilities for. Below is an example of the annotation as it would appear in an adaptation for an Activity Mapper called `ExampleMapper` that maps a single Activity Type called `ExampleActivity` (It is important to note that the Activity Types listed in the annotation be denoted by their class name, which may sometimes differ from the actual Activity Type name).

```
@ActivitiesMapped({ExampleActivity.class})
public class ExampleMapper implements ActivityMapper { ... }
```

Activity Schema

Each Activity Mapper must describe the structure of its associated Activity Types by implementing the `getActivitySchemas()` method whose signature is shown below:

```
Map<String, Map<String, ParameterSchema>> getActivitySchemas();
```

The `Map` returned by this method should contain a schema for each associated Activity Type, indexed by Activity Type name. Each activity schema is represented by another `Map`, supplying parameter names as keys, and the corresponding `ParameterSchemas` as values.

Serialization

Activity Mappers must supply a `serialize()` method as specified by the following signature:

```
Optional<SerializedActivity> serializeActivity(Activity activity);
```

The `Optional` returned by this method should be empty if the provided `Activity` is not an instance of an Activity Type mapped by this Activity Mapper. However, if the provided `Activity` is indeed an instance of one of the Activity Types the Activity Mapper handles, then a `SerializedActivity` representing the instance should be returned.

Deserialization

The `deserialize()` method must also be provided by Activity Mappers, and is described by the following method signature:

```
Optional<Activity> deserializeActivity(SerializedActivity serializedActivity);
```

The `Optional` returned by this method should be empty if no instance for any of the associated Activity Types can be constructed from the provided `SerializedActivity`. Otherwise, an instance of the associated Activity Type that fits the provided `SerializedActivity` should be provided.

It is important to note that the result of an Activity Mapper's `serialize()` method be accepted by the mapper's `deserialize()` method, and vice-versa.

Creating Activity Mappers

Even if one understands what an Activity Mapper should contain, it is another challenge entirely to actually create one. In many cases, it may be enough to simply have Merlin generate one for Activity Types. If this is the case, `generateMapper` should be set to `true` in the `@ActivityType` annotation for the Activity Type (this is set to `false` by default).

Currently, Merlin can generate Activity Mappers for Activity Types which contain only supported parameter types. This means that for Activity Types containing custom parameter types, adaptation engineers will need to supply custom activity mappers (we are planning to add support for custom activity mappers in the future, though some work by the adaptation engineer will still be required). ### Using Custom Parameter Types in Activity Types If a custom parameter type needed to be utilized in an Activity Type, custom mappers has to be created. This is a two step process: 1. Custom activity mapper has to be created for the activity in which you want to add the custom parameter. 2. Custom parameter mapper has to be created. To expedite the first mapper generation, adapters can rely on the automatic activity mapper generator by

excluding the custom parameters in a first pass, and then manually edit the mapper to include the custom parameters. Entities with custom mappers should not include annotations to avoid conflict between automatic generation process. In the future, mapper generation for custom parameters will be automated.

To create a custom Activity Mapper, a Java class meeting the criteria for Activity Mappers should be created. A skeleton for all Activity Mappers is given below:

```
@ActivitiesMapped({ExampleActivity.class})
public class ExampleMapper implements ActivityMapper {

    @Override
    public Map<String, Map<String, ParameterSchema>> getActivitySchemas() {
        ...
    }

    @Override
    public Optional<Activity> deserializeActivity(SerializedActivity serializedActivity) {
        ...
    }

    @Override
    public Optional<SerializedActivity serializeActivity(Activity activity) {
        ...
    }
}
```

A custom Activity Mapper may be constructed from scratch, though one can get a head start by generating a partial mapper for an Activity Type. To do this, temporarily comment out any `@Parameter` annotations for custom parameters, and add `generateMapper=true` to the `@ActivityType` annotation, then run the Merlin annotation processor. This will generate a mapper for the Activity Type that handles all supported parameter types. The adaptation engineer can then copy this generated mapper to their source folder and add logic for custom parameter types, using the generated code as a guide. Note that when using this procedure, the `@generated` annotation should be removed from the Activity Mapper, and the temporary changes to the Activity Type should be removed.

Parameter Mappers

In creating an Activity Mapper, it may be useful to use Parameter Mappers. A `ParameterMapper` is similar to an `ActivityMapper` in, but focuses on a single Activity Parameter. Properly used, Parameter Mappers can make Activity Mappers quite simple, handling all the dirty work at the parameter level. In fact, Merlin's generated Activity Mappers work exclusively using Parameter Mappers.

One of the most convenient things about using Parameter Mappers is the fact

that Merlin comes with them already defined for all supported parameter types. Furthermore, Parameter Mappers for combinations of supported types can easily be created by passing one `ParameterMapper` into another during instantiation.

Although we provide Parameter Mappers for supported parameter types, it is entirely acceptable to create custom Parameter Mappers for use in custom Activity Mappers. This can be done by writing a Java class which implements the `ParameterMapper` interface. Below is a Parameter Mapper for a String type parameter as an example:

```
public final class StringParameterMapper implements ParameterMapper<String> {
    @Override
    public ParameterSchema getParameterSchema() {
        return ParameterSchema.STRING;
    }

    @Override
    public Result<String, String> deserializeParameter(final SerializedParameter serializedP
        return serializedParameter
            .asString()
            .map(Result::<String, String>success)
            .orElseGet(() -> Result.failure("Expected string, got " + serializedParameter.to
    }

    @Override
    public SerializedParameter serializeParameter(final String parameter) {
        return SerializedParameter.of(parameter);
    }
}
```

Example Activity Mapper

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

Activity Type

```
@ActivityType(name=RunHeaters)
public class RunHeatersActivity {

    @Parameter
    public double heatDuration;

    @Parameter
    public int heatIntensity;

    @Override
```

```
    public void modelEffects() { ... }
}
```

Activity Mapper

```
@ActivitiesMapped({RunHeatersActivity.class})
public class RunHeatersMapper implements ActivityMapper {

    // Instantiate a ParameterMapper for each parameter of the RunHeater activity
    // These allow us to delegate to ParameterMappers for parameter-specific work
    private static final ParameterMapper<Double> mapper_heatDuration = new NullableParameter
    private static final ParameterMapper<Integer> mapper_heatIntensity = new NullableParamet

    @Override
    public Map<String, Map<String, ParameterSchema>> getActivitySchemas() {

        // Instantiate the Activity Schemas map
        Map<String, Map<String, ParameterSchema>> activitySchemas = new HashMap<>();

        // Build the Activity Schema for the RunHeaters activity
        Map<String, ParameterSchema> runHeatersSchema = new HashMap<>();
        runHeatersSchema.put("heatDuration", ParameterSchema.DOUBLE);
        runHeatersSchema.put("heatIntensity", ParameterSchema.INTEGER);

        // Populate the Activity Schemas map
        // Note that the Activity Type name is used for the Activity Schema map
        // instead of the Activity Type class
        activitySchemas.put("RunHeaters", runHeatersSchema);

        return activitySchemas;
    }

    @Override
    public Optional<Activity> deserializeActivity(SerializedActivity serializedActivity) {
        // If activity is not the supported type, return empty optional
        if (!serializedActivity.getTypeName().equals("RunHeaters")) {
          return Optional.empty();
        }

        // Create the Activity Instance and set parameter values if present
        final var activity = new RunHeatersActivity();
        for (final var entry : serializedActivity.getParameters().entrySet()) {
            switch (entry.getKey()) {
                case "heatDuration":
                    activity.heatDuration = this.mapper_heatDuration.deserializeParameter(er
```

```
                    break;
                case "heatIntensity":
                    activity.heatIntensity = this.mapper_heatIntensity.deserializeParameter(
                    break;
                default:
                    // Unexpected parameter key present, return empty optional
                    return Optional.empty();
            }
        }

        return Optional.of(activity);
    }

    @Override
    public Optional<SerializedActivity serializeActivity(Activity activity) {

        // If activity is not suppoorted, return empty Optional
        if (!(activity instanceof RunHeatersActivity)) return Optional.empty();

        final RunHeatersActivity activity = (RunHeatersActivity)activity;

        final var parameters = new HashMap<String, SerializedParameter>();
        parameters.put("heatDuration", this.mapper_heatDuration.serializeParameter(activity.
        parameters.put("heatIntensity", this.mapper_heatIntensity.serializeParameter(activit

        return Optional.of(new SerializedActivity(ACTIVITY_TYPE, parameters));
    }
}
```