Oblog Documentation

Release 0.0.1

Eli Barnett

Jan 25, 2022

CONTENTS

Welcome to the documentation for Oblog, an annotation-based Shuffleboard logging tool for FRC teams!

For an overview of what Oblog is and why you should use it, see the *introduction page*. For a quick start guide, see *Getting Started with Oblog*.

The Oblog API docs can be found here.

An example project (based on the WPILib FrisbeeBot example project) demonstrating the usage of several Oblog features can be found here.

CHAPTER

ONE

CONTENTS

1.1 Introduction to Oblog

1.1.1 What Is Oblog?

Oblog is an annotation-based Shuffleboard telemetry logging utility for FRC teams. With Oblog, logging your robot state to Shuffleboard is relatively painless and has a minimal code footprint - ideal for teams that want the benefits of extensive telemetry logging without the hassle of designing their code around it.

Oblog is currently only available for Java, though it will also work seamlessly with Kotlin with no additional configuration.

1.1.2 Why Use Oblog?

Shuffleboard is the supported WPILib dashboard and telemetry-logging solution. Shuffleboard has many extremely convenient features, including tab-based display, an assortment of layout types, event logging, and more. It is a fantastic solution for both dashboard telemetry display and telemetry logging.

However, the Shuffleboard API provided through WPILib is highly verbose and somewhat clunky. The fluent builder pattern used results in rather long method call chains for relatively simple dashboard configuration, and widget parameters are specified entirely through string-valued key-value pairs, which are error-prone and not self-documenting.

Oblog improves this by replacing the fluent builder API with one based on annotations:

```
// With Oblog, this will send loggedInteger to the dashboard - it's that easy!
@Log
int loggedInteger;
```

Rather than calling methods to log data explicitly, Oblog allows users to tag fields, getters, and/or setters with widget-specific annotations, which are then automatically sent to the dashboard. This results in an extremely small code footprint, and allows logging functionality to easily be added to existing code with no major changes - simply annotate the fields you want to be logged, and Oblog does the rest.

For complex robot projects, such as those using the WPILib Command-based framework, it often becomes desirable to structure both the dashboard layout and the log file structure to mirror the structure of the code. Oblog supports this, as well - through use of the `Loggable` interface, users can specify classes that will constitute tabs or layouts on their dashboard. Oblog will automatically recurse down the tree of `Loggable` objects at start time, building a corresponding tree of tabs, layouts, and sub-layouts on the dashboard. Users can thus, for example, easily specify separate tabs for each of their robot subsystems or commands, without significantly increasing their code verbosity - the annotations remain unchanged, and the annotated fields are automatically placed in the correct tabs or layouts!

1.1.3 Contributing to Oblog

Oblog is an open-source project, and suggestions/contributions from users are welcome on the github page.

1.2 Getting Started with Oblog

Integrating Oblog with a robot project is easy - this page will provide a brief guide on the steps required to do it.

1.2.1 Adding Oblog as a Dependency

The first step to using Oblog is to add it to your robot project as a dependency. Oblog is distributed using jitpack.

To add Oblog as a dependency, we only need to add a couple lines to your project's `build.gradle`. First, add the following:

```
repositories {
  maven { url 'https://jitpack.io' }
}
```

Secondly, add the following to the `dependencies` list:

```
implementation "com.github.Oblarg:Oblog:RELEASE_TAG"
```

where `RELEASE_TAG` is the latest release version tag (e.g. `3.0.3`).

If you wish to automatically build with the latest version of Oblog, you can use `master-SNAPSHOT` instead of a release tag - keep in mind, however, that this may break your code without warning when Oblog is updated.

1.2.2 Configuring the Logger

Now that Oblog has been added as a dependency, it's time to start logging!

The first thing to do is to determine which project class will serve as your "root container." This is the "base class" of your robot, and will constitute the primary tab of your dashboard. For a basic robot project, this will likely be `Robot.java`. For a command-based project, it will likely be `RobotContainer.java`. For the purposes of this guide, we'll assume a simple project is being used, and so `Robot.java` is the root container.

Now, we'll add the following call to the `robotInit` method of `Robot.java` (note: you will need to import the `Logger` class from the `io.github.oblarg.oblog` package):

```
// The first argument is the root container
// The second argument is whether logging and config should be given separate tabs
Logger.configureLoggingAndConfig(this, false);
```

The `configureLoggingAndConfig` call takes the aforementioned "root container" as a parameter (since we've chosen `Robot.java` as our root container, we simply pass `this` since this call is located in `Robot.java` itself).

Finally, we add the following call to the `robotPeriodic` method of `Robot.java`:

```
Logger.updateEntries();
```

The logger is now configured, and we're ready to do some basic logging.

1.2.3 Logging a Field

To log a field in the root container, simply annotate it with the `@Log` annotation (imported from the `io.github.oblarg.oblog.annotations` package):

```
@Log
int exampleField = 5;
```

Note that the root container is specified by the `configureLoggingAndConfig` call. If you want to log items from classes other than the root container, you must make those classes implement the `Loggable` interface. For more information, see *Creating Loggable Classes*.

The field will be automatically added logged on the dashboard, and will track the value of the field in code. It's that easy!

Of course, this is only a very simple example. For a description of the full set of logging features supported by Oblog, see *Oblog Log Annotations*.

1.2.4 Configuring a Field

Oblog doesn't only support logging - it also supports configuring the value of fields from the dashboard! To bind a value of a field to an interactive dashboard widget, simply annotate a setter function for the field with the `@Config` annotation:

```
@Config
public void setExampleField(int value) {
  exampleField = value;
}
```

Oblog will automatically call the setter with the new value any time its value is changed on the dashboard!

The `@Config` annotation can also be used directly on fields that implement the WPILib `Sendable` interface. For a full description of the config features supported by Oblog, see *Oblog Config Annotations*.

1.2.5 Creating Additional Tabs

As our robot program becomes more complex, it becomes less and less tenable to just log everything in the root container's tab. Oblog's solution to this problem is to automatically infer the tab structure of your dashboard from the structure of your robot code. To enable it to do this, we use the `Loggable` interface. Any field of your root container that implements the `Loggable` interface will automatically be given its own Shuffleboard tab.

For an in-depth description of the use of the `Loggable` interface, see *Creating Loggable Classes*.

1.3 Oblog Log Annotations

Oblog supports logging of telemetry data through the `@Log` annotation and a variety of widget-specific sub-annotations.

1.3.1 Using the Log Annotations

The @Log Annotation

The `@Log` annotation can be used on both fields and getters, as follows:

```
@Log
int exampleField;
```

```
@Log
int getExampleValue() {
  return exampleValue;
}
```

Like all Oblog annotations, this works regardless of access modifiers (e.g. `public` and `private`).

The `@Log` annotation is the most-general annotation available in Oblog. It will use a default widget type inferred from the type of the field or the return type of the getter, and will work with any of Oblog's supported data types.

If an alternative widget is preferred - or if you wish to configure the logging further with widget-specific parameters - one of the widget-specific sub-annotations can be used. For a list of these, see the API docs.

Sendable Widgets

Many Shuffleboard widget types correspond to objects that implement the WPILib `Sendable` interface. To log these types of data, *only* field annotations should be used, rather than getter annotations, as Oblog will assume that they are persistent objects.

1.3.2 Parameters

The various logging annotations take allow configuration of the logged value through a number of parameters. This section describes those parameters that are common to all of the logging annotations; for a detailed description of the widget-specific parameters, see the API docs.

Value Name

The name of the value on shuffleboard (and of its corresponding NetworkTables key) can be set with the `name` parameter.

If the name parameter is not provided, Oblog will use the name of the annotated field or getter.

Tab Name

The `tabName` parameter can be used to explicitly override Oblog's inferred tab structure. If the `tabName` parameter is set, the logged value will be displayed under a Shuffleboard tab of the specified name.

Method Name

Sometimes it is desirable to extract a value of a supported data type from a field of a complex data type. This can be done using the `methodName` parameter. If the `methodName` parameter is set, rather than the field itself being logged, the result of calling the named method on the field will be logged, instead. The `methodName` functionality only works with field annotations.

Display Size

The size of the displayed Shuffleboard widget can be set with the `width` and `height` parameters. If none are provided, the default Shuffleboard size for the widget type is used. Size is measured in Shuffleboard grid units.

Display Location

Warning: If the position of a single widget or layout on a tab is manually specified, the position of *all* widgets and layouts on the tab should be manually specified. Failure to do so will likely result in overlapping/hidden widgets, since Shuffleboard's auto-placement algorithm does not interact nicely with manually-positioned widgets.

The location of the displayed widget in its tab can be set with the `rowIndex` and `columnIndex` parameters. The position is measured in Shuffleboard grid units. If position is not set, Shuffleboard will automatically place the widget.

1.4 Oblog Config Annotations

Oblog supports data-binding of code values through the `@Config` annotation and a variety of widget-specific sub-annotations.

1.4.1 Using the Config Annotations

The @Config Annotation

Note: The Shuffleboard values displayed for data-binding via setter methods are one-way; changes to the value in code will *not* be reflected on the dashboard. This holds true even if the value is modified by Oblog itself; if two data-binding widgets are configured for the same setter, their displayed values will not automatically remain synchronized. The value in code will correspond to whichever widget was most-recently updated.

The `@Config` annotation may be used to perform data-binding on boolean- or numeric-valued setters and `Sendable` fields, as follows:

```
@Config
void setExampleValue(double value) {
  exampleValue = value;
}
```

```
@Config
PIDController exampleController;
```

Like all Oblog annotations, this works regardless of access modifiers (e.g. `public` and `private`).

The `@Config` annotation is the most-general annotation available in Oblog. It will use a default widget type inferred from the parameter type of the setter or the type of the field, and will work with any of Oblog's supported data types.

If an alternative widget is preferred - or if you wish to configure the logging further with widget-specific parameters - one of the widget-specific sub-annotations can be used. For a list of these, see the API docs.

Multi-Parameter Setters

Warning: If individual parameter names are not specified with parameter-level annotations, Oblog will attempt to infer them from the parameter names in code. This functionality requires the optional `-parameters` java

compiler flag - be sure to add this to your `build.gradle` if you wish to use this functionality. If this is not set, and individual names are not provided, the widgets will be titled `arg0`, `arg1`, `arg2`, etc.

The `@Config` annotation also works with multi-parameter setters:

```
@Config
void setPID(double p, double i, double d) {
  controller.setPID(p, i, d);
{
```

If a multi-parameter setter is annotated, it will display on the dashboard as a layout, with the individual parameters as widgets in the layout. The widgets of the individual parameters can be controlled as one normally would a single-parameter setter by annotating each of the method parameters individually with `@Config`.

Sendable Widgets

Many Shuffleboard widget types correspond to objects that implement the WPILib `Sendable` interface. To configure these types of data, *only* field annotations should be used, rather than setter annotations, as Oblog will assume that they are persistent objects.

1.4.2 Parameters

The various configurating annotations take allow configuration of the bound value through a number of parameters. This section describes those parameters that are common to all of the configuration annotations; for a detailed description of the widget-specific parameters, see the API docs.

Value Name

The name of the value on shuffleboard (and of its corresponding NetworkTables key) can be set with the `name` parameter.

If the name parameter is not provided, Oblog will use the name of the annotated setter or field.

Tab Name

The `tabName` parameter can be used to explicitly override Oblog's inferred tab structure. If the `tabName` parameter is set, the bound value will be displayed under a Shuffleboard tab of the specified name.

Method Name/Types

The `methodName` and `methodTypes` parameters can be used to extract a setter from a complex type for data-binding. To do this, use the parameters to specify the name and argument types of the setter. For example:

```
public class HasSetter {
  public void setFoo(double foo) {
    // what is done with the argument is unimportant
  }
}
```

```
@Config(methodName = "setFoo", methodTypes = {double.class})
HasSetter hasSetter;
```

Default Values

When binding data through a setter, it's important to provide a default value for the bound variable. This can be specified with the `defaultValueNumeric` or `defaultValueBoolean` parameter. When the logger runs its initial configuration method, the default value will be passed to the setter. If a default value is not provided by the user, Oblog will use `0` for numeric data types or `false` for boolean data types.

Multi-Parameter Layout Type

When performing *data-binding on multi-parameter setters*, users have a choice of using either a list layout or a grid layout. This can be set with the `multiArgLayoutType` parameter, which can take values of either `"listLayout"` or `"gridLayout"`. If this is not specified, it will default to a list layout.

If a grid layout is used, the number of rows/columns in the internal grid can be set with the `numGridRows` and `numGridColumns` parameters. Both will default to a value of `3` if not provided.

Display Size

The size of the displayed Shuffleboard widget can be set with the `width` and `height` parameters. If none are provided, the default Shuffleboard size for the widget type is used. Size is measured in Shuffleboard grid units.

Display Location

> **Warning:** If the position of a single widget or layout on a tab is manually specified, the position of *all* widgets and layouts on the tab should be manually specified. Failure to do so will likely result in overlapping/hidden widgets, since Shuffleboard's auto-placement algorithm does not interact nicely with manually-positioned widgets.

The location of the displayed widget in its tab can be set with the `rowIndex` and `columnIndex` parameters. The position is measured in Shuffleboard grid units. If position is not set, Shuffleboard will automatically place the widget.

1.5 Creating Loggable Classes

One of Oblog's most powerful features is the ability to infer a tab/layout structure from the structure of the user's code, removing the need for the user to worry about specifying the location of their logged widgets on the dashboard.

In order to do this, users must use the Loggable interface.

1.5.1 Using the Loggable Interface

Making a Class Loggable

All of the methods in the `Loggable` interface are defaulted. Thus, marking a class as loggable, in the simplest case, requires nothing more than declaring that it implements the `Loggable` interface:

```
public class Foo implements Loggable {
  // your class here
}
```

Once a class has been declared `Loggable`, Oblog will automatically search it for annotated fields, getters, and setters to populate the dashboard - as long as it is reachable from the specified root container through a direct sequence of `Loggable` parent classes (the logger will recursively search down the tree of all `Loggable` fields from the root container).

`Loggable` classes located directly in the root container will be given their own tabs. `Loggable` classes located *inside of other loggable classes* will be given layouts in the tabs (or layouts) corresponding to their parents.

Configuring Loggable Names

The name of the `Loggable` class's tab or layout can be configured by overriding the `configureLogName` method to return the desired name. By default, the simple class name is used.

To avoid namespace collisions when many instances of the same `Loggable` class are present, it is highly recommended to override this method to return a name that is indexed based on a constructor parameter (such as a port number).

Configuring Loggable Layouts

When a `Loggable` class is located inside of another `Loggable` class, it is displayed as a layout. Shuffleboard contains two types of layouts - list layouts and grid layouts - and each layout type has a number of configuration options. A number of `Loggable` methods can be overridden to specify layout configuration.

Layout Type

Layout type can be configured by overriding the `configureLayoutType()` method to return the desired layout type. A list layout is used by default.

Layout Size

Layout size can be configured by overriding the `configureLayoutSize()` method to return a two-element integer array corresponding to the desired layout width and height. If left defaulted, Shuffleboard will automatically determine layout size.

Layout Position

> **Warning:** If the position of a single widget or layout on a tab is manually specified, the position of *all* widgets and layouts on the tab should be manually specified. Failure to do so will likely result in overlapping/hidden widgets, since Shuffleboard's auto-placement algorithm does not interact nicely with manually-positioned widgets.

Layout position can be configured by overriding the `configureLayoutPosition()` method to return a two-element integer array corresponding to the desired layout column and row. If left defaulted, Shuffleboard will automatically position the layout.

Note that this will determine the layout position for *all instances of this class*. Accordingly, when using this feature, it is recommended to pass in the position values as constructor parameters to your class. Unfortunately, it is not feasible to provide `Loggable` layout positioning through annotations, as this does not integrate smoothly with detection of arrays/lists of `Loggable` fields, so some hard coupling to user code is unavoidable here.

Skipping Layouts

Sometimes it is not worth giving a `Loggable` class its own layout. To place all widgets from a `Loggable` class directly into the container of its parent, override the `skipLayout()` method to return `true`.

Adding Custom Logging

If Oblog's annotation-supported logging functionalities are ever insufficient, a `Loggable` class can be given a custom Shuffleboard logging routine by overriding the `addCustomLogging` method. This allows access to the full `Shuffleboard` API while retaining Oblog's inferred tab/layout structure.

1.5.2 Excluding and Re-Including Loggables

Sometimes a `Loggable` class will occur as a field of many other `Loggable` classes (such as subsystems that are injected into many different commands). It then becomes desirable to "pick-and-choose" where the `Loggable` will occur in the dashboard, since by default, it will be duplicated in every single place in which it occurs as a field.

To do this, annotate the `Loggable` class with the `Log.Exclude` or `Config.Exclude` annotation. `Loggable` classes excluded in this way will never appear on the dashboard unless they are explicitly re-included with a `Log.Include` or `Config.Include` annotation on the specific field whose display on the dashboard is desired.

Alternatively, the `Exclude` annotations can also be used on individual fields, rather than on the entire class.

1.5.3 Arrays/Lists of Loggables

Oblog will also correctly handle arrays and lists of `Loggable` classes; in fact, Oblog dynamically checks the runtime type of each array/list element during startup, so `Loggable` elements of arrays/lists will be handled correctly even if the declared array type is not itself `Loggable`.

1.6 The Oblog Logger

Oblog's Logger class is responsible for the "magic" of Oblog. It performs the initial recursive search for loggable fields and methods, and is responsible for updating the display values of the widgets.

1.6.1 Initial Logger Configuration

As seen in the *getting started* section, for Oblog to work the user must call one of `Logger`'s configuration methods at robot startup.

Each configuration method requires the user to specify a "root container." This is the "root node" from which the recursive search for `Loggable` fields will be performed. In simple terms, this should be the class that contains most of the important robot fields - generally, this will be either `Robot.java` or `RobotContainer.java` for standard WPILib project structures.

Users with more "distributed" project structures are free to call the configuration methods multiple times on different root containers.

The available configuration methods are summarized below:

Configuring Logging and Config

Calling the `configureLoggingAndConfig` method will parse the object tree for both `@Log` and `@Config` annotations. The second parameter is a boolean which will determine whether `@Log`- and `Config`-generated widgets are given separate tabs, or combined on the same set of tabs.

Configuring Logging Only

The `configureLogging` method will parse the object tree *only* for `@Log` annotations.

Configure Logging Only (NT Only)

The `configureLoggingNTOnly` method will parse the object tree *only* for `@Log` annotations, and also bypass all Shuffleboard widgets and only push the raw logged values to NetworkTables. This is useful for teams that wish to use a custom dashboard implementation, or otherwise consume the NetworkTables information through a means other than Shuffleboard.

Configuring Config Only

The `configureConfig` method will parse the object t ree *only* for `@Config` annotations.

1.6.2 Updating the Logged Fields

Initial logging sets up the Shuffleboard widgets and populates them with initial values; however, in order for the displayed widgets to remain synchronized with the values in code, Oblog needs to periodically update them.

Accordingly, the `updateEntries` method must be called periodically from the user program. Generally, this is done from the `robotPeriodic` method of `Robot.java`.

This *can* be done from elsewhere, but it is *essential* for the purposes of thread-safety that the `updateEntries` method be called from the same thread context in which the logged fields are declared. For complicated user programs with multiple thread contexts, the relevant *configuration method* should be called separately for each thread context.

1.6.3 Disabling Cycle Warnings

The `Logger` will automatically terminate a branch of the recursive search when it detects a cyclic reference of `Loggable` objects (otherwise it would recurse indefinitely). By default, it will print a user warning when this happens.

However, some typical design patterns (such as singletons) rely heavily on cyclic references, and so these warnings can become a nuisance. To disable them, call the `setCycleWarningsEnabled` method.