ALICE Offline week

Contextual Logging in jAliEn

Tim Hallyburton (EP-AIP-GTP)

Contents

- Current Logging in jAliEn
- Contexts in jAliEn
- Contextual Logging
- Usage
- Custom Annotations

Current Logging in jAliEn

- Uses java.util.logging.Logger

Severity levels from *finest* to *severe* are used, each with it's own output file.

Loggers are retreived by using `ConfigUtils.getLogger` which caches the loggers created by `Logger.getLogger`

There is no convention regarding the 'correct' severity level.

Current Logging in jAliEn

Output looks like this

```
lyb@lyb-ThinkPad-X1-Carbon-4th ~/workspace/jalien $ ls -alF | grep log
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-config0.log
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-config0.log.1
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-config0.log.1.lck
-rw-r--r--  1 lyb lyb   523 Feb 24 15:55 alien-fine0.log
-rw-r--r--  1 lyb lyb   514 Feb 24 15:55 alien-fine0.log.1
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-fine0.log.1.lck
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-finer0.log
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-finer0.log.1
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-finer0.log.1.lck
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-finest0.log
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-finest0.log.1
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-finest0.log.1.lck
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-info0.log
-rw-r--r--  1 lyb lyb 22051 Feb 24 15:55 alien-info0.log.1
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-info0.log.1.lck
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-severe0.log
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-severe0.log.1
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-severe0.log.1.lck
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-warning0.log
-rw-r--r--  1 lyb lyb  2583 Feb 24 15:55 alien-warning0.log.1
-rw-r--r--  1 lyb lyb     0 Feb 24 15:55 alien-warning0.log.1.lck
```

Current logging in jAliEn

Problems:

- Log files are hard to follow
- Unclear origin of log-messages in shared parts of the program
- Access control files not cleaned up
- Logs are categorized by severity-level only
- Left-over access control files
- File rotation is enabled (and we don't want that)

Contexts in jAliEn

Context.java provides a thread-wide shared HashMap for key-indexed values of any type

```
static Map<Thread, Map<String, Object>> context
  = new ConcurrentHashMap<>();
```

Also spawns a cleanup-thread to clear maps of terminated threads.

Contextual Logging

Consists of three major components:

- Service-dedicated loggers
- Log rules (conditions)
- A 'logging context'

CL - Service loggers

Every major component of the system is assigned to a dedicated logger. These components (services) can be as general or specialized as needed.

These loggers point to individual files which can be followed more easily.

The origin of log messages from shared classes/methods is no longer ambigious.

CL - Log Rules

Log rules are defined in the `logging.properties` file, along with their respective service loggers.

```
Jsh.logrules = JSh, Production, log_enabled
Test.logrules = log_enabled

(implicit: default.logrules = ) <- none
```

The JSh-Service Logger will only accept messages issued at a time when the logging context contains all three tags JSh, Production and log_enabled.

CL - Logging Context

Basically extends the jAliEn-Context by adding a final *loggingTag* and providing additional methods for easy usage (see Usage).

Also uses a logging-context-stack to keep track of overwritten logging-contexts.

Usage

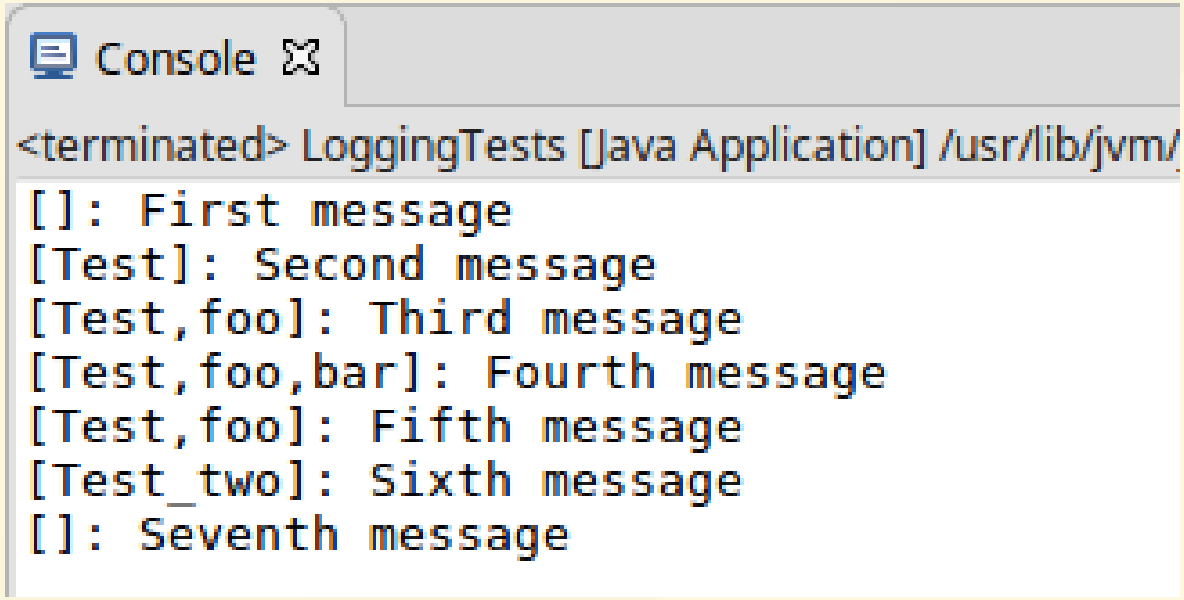
Navigating logging contexts

- `addToLoggingContext(String tag)`
  `addToLoggingContext(String... tag)`
- `removeFromLoggingContext(String tag)`
  `removeFromLoggingContext(String... tag)`
- `overwriteLoggingContext(String tag)`
  `returnToPreviousLoggingContext()`

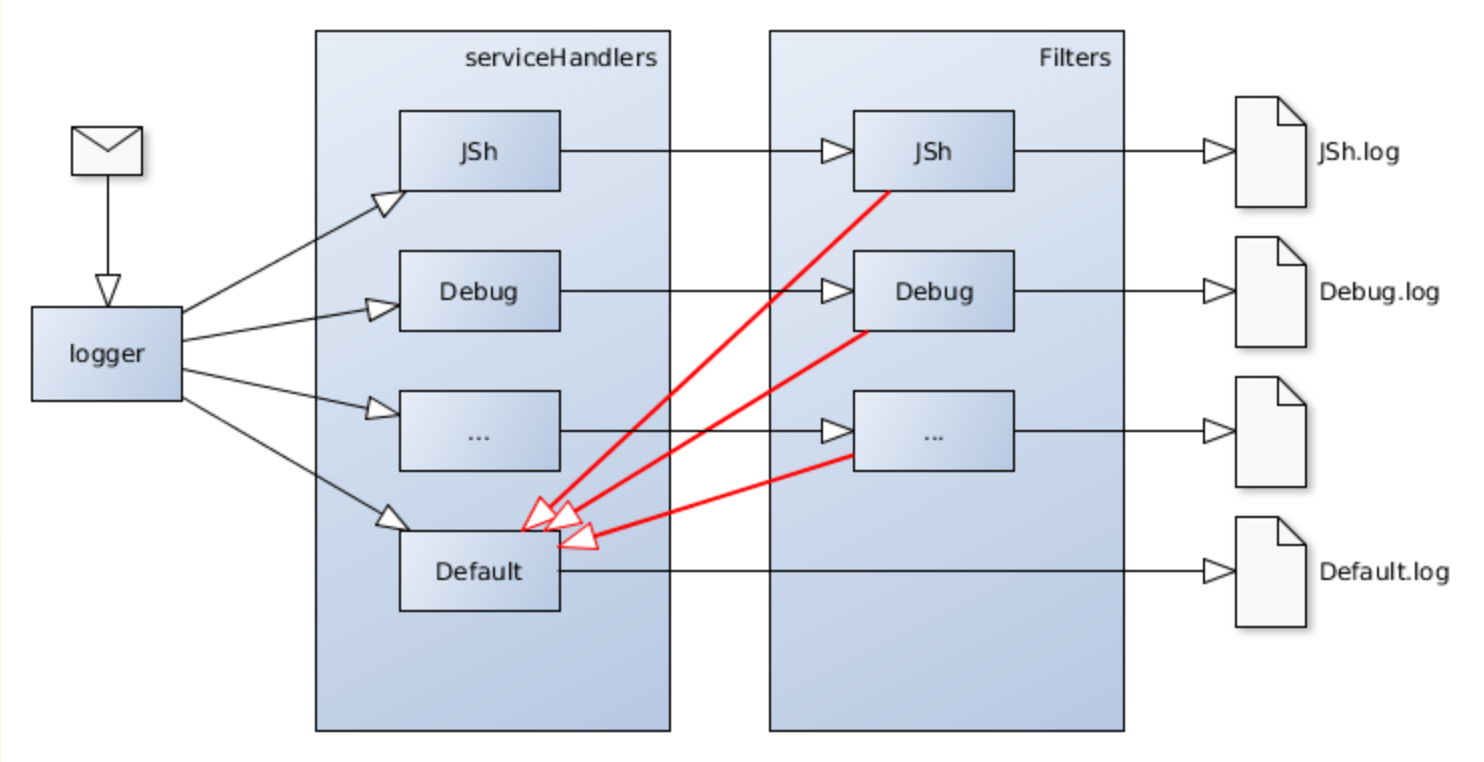
Usage - Example

```java
package alien;
import alien.config.Context;

public class LoggingTests {

  public static void main(String[] args) {
    printLoggingWithContext("First message");
    Context.addToLoggingContext("Test");
    printLoggingWithContext("Second message");

    foo();

    printLoggingWithContext("Third message");

    bar();

    printLoggingWithContext("Fifth message");

    Context.overwriteLoggingContext("Test_two");

    printLoggingWithContext("Sixth message");

    Context.resetLoggingContext();

    printLoggingWithContext("Seventh message");
  }

  public static void printLoggingWithContext(String msg) {
    System.out.println("[" + Context.getLoggingContext() + "]: " + msg);
  }

  public static void foo() {
    Context.addToLoggingContext("foo");
  }

  public static void bar() {
    Context.addToLoggingContext("bar");
    printLoggingWithContext("Fourth message");
    Context.removeFromLoggingContext("bar");
  }
}
```

Usage - Example



Usage - Example




```
defaultLoggerRef.notifyRecordConsumed()
defaultLoggerRef.notifyRecordRejected()
```

Custom Annotations

`@ContextAnnotation` can be used to override or extend the logging context when entering a method.

It basically modifies the AST to prepend `Context.overrideLoggingContext` to the method definition block and statements to return to the previous context before each return-statement.

Custom Annotations

The ContextAnnotation will be processed by the custom annotation processor.

The `ContextAnnotationProcessor` is registered in javac and will modify the AST of the annotated methods.

This reduces code replications. Moving context-related operations into Annotations also supports seperation of concerns.

Custom Annotations

```
public class Test {

  static transient final Logger logger
    = ConfigUtils.getLogger("Test-Logger");

  @ContextAnnotation(ctx="Debug-Foo")
  public static void foo()
  {
    // ...
    logger.log(Level.FINE, "debug message");
  }

}
```

Custom Annotations

```
  @ContextAnnotation(ctx="Debug-Foo")
  public static void foo()
  {
    // ...
    logger.log(Level.FINE, "debug message");
  }

}
```

```
  public static void foo()
  {
    Context.overwriteLoggingContext("Debug-Foo");
    // ...
    logger.log(Level.FINE, "debug message");
    Context.returnToPreviousLoggingContext();
  }
```

Custom Annotations

Arguments:

```
@ContextAnnotation(ctx="tag",
        preserveContext=true,
        append=false)
```

Signature:

```
public @interface ContextAnnotation { //...
```

Custom Annotations

Modification of the AST is done within the `ContextAnnotationProcessor` and casts the provided `javax.lang.model.element.Element` entities to their internal compiler types `JCStatement, JCExpression, JCTree, JCBlock ...` .

By doing this, it can be avoided to create new source files and run additional compilation rounds.

Custom Annotations

There are also `TimeProfilingAnnotations` in development. These will wrap the annotated method with a `closable` object which will log the runtime of the wrapped method.

This will introduce another implicite service-logger, for a `TimeProfilingService`.

Thank you!