

ALogger.java

```
package com.ada.utilidades.situ.log;

import java.lang.reflect.Method;
import java.net.InetAddress;
import java.util.Calendar;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import com.ada.utilidades.situ.cliente.LogExcepcionClienteWS;
import com.ada.utilidades.situ.entidad.Mannager;
import com.ada.utilidades.situ.exception.ControlException;
import com.ada.utilidades.situ.exception.GeneralException;
import com.ada.utilidades.situ.utils.Resource;

/**
 * Clase que encapsula y facilita la utilizacion del Logger
 *
 * @author Jhon De Avila Mercado
 * @version 1.0
 * @since 2021
 */
public abstract class ALogger {
    private String msgError = "Error del sistema, no se pudo completar la acción solicitada";

    private org.apache.log4j.Logger logger;
    protected static ALogger instance;

    protected final int debug = 0;
    protected final int info = 1;
    protected final int warn = 2;
    protected final int error = 3;
    protected final int fatal = 4;

    private int level = 3;

    /**
     * @param nombre del log especificado en el properties
     * @deprecated
     */
    public ALogger(String nombre) {
        logger = org.apache.log4j.Logger.getLogger(nombre);
    }

    public ALogger() {
        logger = org.apache.log4j.Logger.getLogger("rootLogger");
    }
}
```

```
/**
 * Se utiliza para escribir mensajes de depuraci  n, este log no
debe estar
 * activado cuando la aplicaci  n se encuentre en producci  n no
informa el stack
 * trace.
 *
 * @param objeto mensaje tipo debug
 */
public void debug(final Object objeto) {
    String msg = objeto != null ? objeto.toString() : msgError;
    print(msg, null);
}

/**
 * Se utiliza para escribir mensajes de depuraci  n, este log no
debe estar
 * activado cuando la aplicaci  n se encuentre en producci  n.
 *
 * @param objeto      objeto mensaje tipo debug
 * @param throwable seguimiento de la llamada
 */
public void debug(final Object objeto, final Throwable throwable) {
    String msg = objeto != null ? objeto.toString() : msgError;
    print(msg, throwable);
}

/**
 * Se utiliza para mensajes similares al modo "verbose" en otras
aplicaciones no
 * informa el stack trace.
 *
 * @param objeto mensaje de tipo info
 */
public void info(final Object objeto) {
    String msg = objeto != null ? objeto.toString() : msgError;
    print(msg, null);
}

/**
 * Se utiliza para mensajes similares al modo "verbose" en otras
aplicaciones.
 *
 * @param objeto      objeto mensaje de tipo info
 * @param throwable seguimiento de la llamada
 */
public void info(final Object objeto, final Throwable throwable) {
    String msg = objeto != null ? objeto.toString() : msgError;
    print(msg, throwable);
}
```

```
/**
 * Se utiliza para mensajes de alerta sobre eventos que se desea
mantener
 * constancia, pero que no afectan el correcto funcionamiento del
programa no
 * informa el stack trace.
 *
 * @param objeto mensaje de tipo warn
 */
public void warn(final Object objeto) {
    String msg = objeto != null ? objeto.toString() : msgError;
    print(msg, null);
}

/**
 * Se utiliza para mensajes de alerta sobre eventos que se desea
mantener
 * constancia, pero que no afectan el correcto funcionamiento del
programa.
 *
 * @param objeto    mensaje de tipo warn
 * @param throwable seguimiento de la llamada
 *
 */
public void warn(final Object objeto, final Throwable throwable) {
    String msg = objeto != null ? objeto.toString() : msgError;
    print(msg, throwable);
}

/**
 * Se utiliza en mensajes de error de la aplicaci n que se desea
guardar, estos
 * eventos afectan al programa pero lo dejan seguir funcionando,
como por
 * ejemplo que alg n par metro de configuraci n no es correcto y se
carga el
 * par metro por defecto siempre informa el stack trace.
 *
 * @param objeto mensaje de tipo error
 */
public void error(final Object objeto) {
    String msg = objeto != null ? objeto.toString() : msgError;
    print(msg, null);
}

/**
 * Se utiliza en mensajes de error de la aplicaci n que se desea
guardar, estos
 * eventos afectan al programa pero lo dejan seguir funcionando,
como por
```

```
* ejemplo que algñn parñmetro de configuraciñn no es correcto y se
carga el
* parñmetro por defecto.
*
* @param objeto    mensaje de tipo error
* @param throwable seguimiento de la llamada
*/
public void error(final Object objeto, final Throwable throwable) {
    String msg = objeto != null ? objeto.toString() : msgError;
    print(msg, throwable);
}

/**
 * Se utiliza para mensajes crñticos del sistema, generalmente
 luego de guardar
 * el mensaje el programa abortarñ siempre informa el stack trace.
 *
 * @param objeto mensaje de tipo fatal
 */
public void fatal(final Object objeto) {
    String msg = objeto != null ? objeto.toString() : msgError;
    print(msg, null);
}

/**
 * Se utiliza para mensajes crñticos del sistema, generalmente
 luego de guardar
 * el mensaje el programa abortarñ.
 *
 * @param objeto    mensaje de tipo fatal
 * @param throwable seguimiento de la llamada
 */
public void fatal(final Object objeto, final Throwable throwable) {
    String msg = objeto != null ? objeto.toString() : msgError;
    print(msg, throwable);
}

protected void print(final Object objeto, final Throwable
throwable) {
    String message = objeto != null ? objeto.toString() : "null";
    print(message, null, 0, throwable);
}

protected void print(final String pathInfo, final int status, final
Throwable throwable) {
    print(null, pathInfo, status, throwable);
}

@SuppressWarnings("unchecked")
protected void print(final String message, final String pathInfo,
```

```

final int status, final Throwable throwable) {
    try {
        final StackTraceElement[] ste =
Thread.currentThread().getStackTrace();
        int positionMethodLog = 3;
        String methodLog = null;

        methodLog = ALogger.class.getName() + "." +
ste[positionMethodLog].getMethodName();

        if (methodLog.equals(ALogger.class.getName() + ".debug")) {
            level = debug;
        } else if (methodLog.equals(ALogger.class.getName() +
".info")) {
            level = info;
        } else if (methodLog.equals(ALogger.class.getName() +
".warn")) {
            level = warn;
        } else if (methodLog.equals(ALogger.class.getName() +
".error")) {
            level = error;
        } else if (methodLog.equals(ALogger.class.getName() +
".fatal")) {
            level = fatal;
        }

        // No se ejecuta ninguna acción si la acción solicitada es
de debug y éste
        // está
        // inactivo
        if (level == debug && logger.isDebugEnabled()) {
            return;
        }

        int depth = 1;
        String trace = null;
        String resource = "";
        String solution = "";
        Mannager mannager = null;
        String dispatch = "";
        String cause = "";

        if (throwable != null) {
            Class<Throwable> throwableClazz = (Class<Throwable>)
throwable.getClass();
            Class<Throwable> causeClazz = null;
            if (throwable.getCause() != null) {
                causeClazz = (Class<Throwable>)
throwable.getCause().getClass();
            }
            dispatch = "\n\tDisparador\t::\t" +

```

```
throwable.getClass();
    Mannager mannagerParam = new Mannager(throwableClazz,
causeClazz);

    for (Mannager mannagerTemp :
Resource.getExceptionsMannagers()) {
        if (mannagerTemp.equals(mannagerParam)) {
            solution = "\n\tSolución\t::\t" +
mannagerTemp.getMessage();
            mannager = mannagerTemp;
            break;
        }
    }

    if (pathInfo != null) {
        resource = "\n\tOrigen\t\t::\t" + pathInfo;
    }
    cause = "\n\tCausa\t\t::\t";
    if (throwable != null && throwable.getCause() != null &&
throwable.getCause().getMessage() != null) {
        cause = cause + throwable.getCause().getMessage();
    } else if (status > 100 && status != 200) {
        cause = cause + "HTTP" + status;
    } else {
        cause = "";
    }

    String messageFormatted = "\n\tMensaje\t\t::\t" + (message
!= null ? message : msgError);
    if (throwable != null && throwable.getMessage() != null) {
        messageFormatted = messageFormatted + " --> " +
throwable.getMessage();
    }
    int length = 0;
    if (level >= warn) {
        length = ste.length - 1 - depth;
    } else {
        length = 4;
    }

    boolean lock = false;

    for (int i = length; length > 0 && i > 0; i--) {
        StackTraceElement element = ste[i];
        String tracePoint = element.getClassName() + "{" +
element.getMethodName() + "} --> "
            + element.getFileName() + " --> línea: " +
element.getLineNumber();
```

```

        if (!lock) {
            String methodLocker = element.getClassName() + "."
+ element.getMethodName();
            lock =
Resource.getLockersByException().contains(methodLocker);
        }

        if
        ((!tracePoint.startsWith("com.ada.utilidades.Logger") &&
tracePoint.startsWith("com.ada"))
            ||
tracePoint.startsWith("com.ada.utilidades.Logger{executeRegainer}")) {
            if (trace == null) {
                trace = resource;
                trace = trace + dispatch;
                trace = trace + cause;
                trace = trace + solution;
                trace = trace + messageFormatted;
                trace = trace + "\n\tTraza del evento::";
                trace = trace + "\t";
            }
            if (!trace.endsWith("\t")) {
                trace = trace + "\t\t\t\t\t";
            }

            trace = trace + tracePoint;
            trace = trace + "\n";
        }
    }

    if (trace != null) {
        trace = trace.substring(0, trace.lastIndexOf("\n"));
    }

    String idMensaje = null;
    if (level >= warn) {
        idMensaje = Math.abs(messageFormatted.hashCode()) + "" +
(Calendar.getInstance().getTimeInMillis());
        String idMensajeString = "\n\tID evento\t::\t" +
idMensaje;
        trace = idMensajeString + trace;
    }

    switch (level) {
    case debug:
        logger.debug(trace);
        break;
    case info:
        logger.info(trace);
        break;
    case warn:

```

```
        logger.warn(trace);
        break;
    case error:
        logger.error(trace);
        break;
    case fatal:
        logger.fatal(trace);
        break;
    }

    if (level >= warn) {
        InetAddress inetAddress = InetAddress.getLocalHost();
        String ip = inetAddress.getHostAddress();
        ip = ip.substring(ip.lastIndexOf(".") + 1);

        String resume = " :: Causa: ";
        //      String resume = " :: ID evento:" + ip + "-" +
        idMensaje + " :: Mensaje:";

        try {
            LogExcepcionClienteWS.save(null);
        } catch (GeneralException e) {
            logger.fatal(e);
        }

        if (mannager != null || lock) {
            if (lock || (!lock && mannager.isLocker())) {
                try {
                    resume = "Acción no procesada" + resume;
                    addMessage(idMensaje, level, resume, resume
+ message);

                } catch (Exception e) {
                }

                if (mannager != null) {
                    executeRegainer(mannager);
                }

                throw new ControlException(idMensaje,
throwable);
            }
        } else if (throwable != null && !(throwable instanceof
ControlException)) {
            resume = "Advertencia de posible fallo" + resume;
            try {
                addMessage(idMensaje, level, resume, resume +
message);
            } catch (Exception e) {
            }
        }
    }
```

```

    }
    } catch (ControlException e) {
        throw e;
    } catch (Exception e) {
        logger.error(e);
    }
}

/**
 * Ejecuta el método de recuperación definido para el manejador de la
 * excepción,
 * sólo se ejecuta si el método está definido en el formato
 * correcto o
 * <code>package.ClassName{method}</code> y no está en ejecución
 *
 * @param mannager Manejador de la excepción a controlar
 */
private void executeRegainer(Mannager mannager) {
    boolean excecuted = false;
    if (mannager != null && mannager.getRegainer() != null &&
!mannager.isExecutingRegainer()) {
        try {
            for (String regainerTemp : mannager.getRegainer()) {
                Pattern pattern = Pattern.compile(
                    "^([a-z]{1}[a-z+\\.]*)+\\.([a-zA-Z]{1}[a-
zA-Z0-9\\_]+[\\{[a-zA-Z]{1}[a-zA-Z0-9\\_]+\\})+");
                Matcher matcher = pattern.matcher(regainerTemp);
                if (matcher.matches()) {
                    final String regainer = regainerTemp;
                    String className = regainer.substring(0,
regainer.indexOf("{"));
                    Class<?> clazz = Class.forName(className);

                    String methodsNames =
regainer.substring(regainer.indexOf("{") + 1,
regainer.lastIndexOf("}"))
                        .replace("{}", "/");
                    String[] methods = methodsNames.split("/");

                    Object object = null;
                    mannager.setExecutingRegainer(true);
                    info("Iniciando tarea de recuperación para el
subsananar " + mannager.getThrowable().getName()
                        + " -> " + mannager.getRegainer());
                    for (String method : methods) {
                        Method exec =
clazz.getDeclaredMethod(method);
                        object = exec.invoke(object);
                        if (object != null) {
                            clazz = object.getClass();
                        }
                    }
                }
            }
        } catch (Exception e) {
            logger.error(e);
        }
    }
}

```

```
        }
        executed = true;
    } else {
        warn(mannager.getThrowable().getName() + " --> "
            + mannager.getRegainer()
            + ". Formato no válido. Formato
esperado: package.ClassName{method}");
    }
}
} catch (Exception e) {
    error(e);
} finally {
    if (executed) {
        info("Finalizada tarea de recuperación para
subsananar " + mannager.getThrowable().getName() + " --> "
            + mannager.getRegainer());
    }
    mannager.setExecutingRegainer(false);
}
}
}

/**
 * Método abastracto para ser implementado en la clase descendiente,
el cual
 * tiene el propósito de notificar visualmente al usuario acerca
los eventos que
 * están siendo reportados. En la implementación de este método, se
usará
 * cualquier tecnología de mensajería con la cual esté
familiarizado el
 * front-end del aplicativo, permitiendo así la portabilidad a
cualquier
 * framework
 */
public abstract void addMessage(final String idMessage, final int
severity, final String resume,
    final String detail);
}
```

From:
<http://wiki.adacsc.co/> - Wiki

Permanent link:
<http://wiki.adacsc.co/doku.php?id=ada:sicoferp:rentas:herramientas:logger:alogger&rev=1630331509>

Last update: 2021/08/30 13:51

