75.10 - Técnicas de Diseño

TP 1.2

1º Cuat. 2014

Grupo 8

90697 - Eisner, Ariel

89563 - Ferreyra, Oscar

Configuración de la API

La configuración de la API se puede realizar a través de un archivo PROPERTIES o de un archivo XML, la aplicación buscara siempre primero la existencia del archivo PROPERTIES, de no encontrarlos pasara a buscar el archivo XML, y en el caso de que no exista ninguna de estas opciones, la API posee una configuración por default.

Ubicación de los archivo de configuración

Los archivos de configuración deben ser ubicados en la ruta

src/main/java/tp1_7510/grupo8/Config/

con el nombre **config.properties** o con **config.xml**

Hay ocho propiedades que deber ser configuradas

1. El separador de mensaje, atributo “separator”.
2. El formato de hora, atributo “formatDate”.
3. El nivel de mensaje (DEBUG,INFO,ETC), atributo “logLevel”.
4. El formato en si del mensaje, atributo “format”.
5. El nombre del archivo al cual loguear
6. La expresión regular
7. Los filter custom
8. Tipo de Log(FILES,JSON,CONSOLES)

NOTA: se usa como nombre del Logger(necesario para que pueda ser solicitado a través del SL4J) el valor en la claves FILES,JSON y CONSOLES

Configurando el archivo Properties

La estructura del archivo properties debe ser CLAVE = VALOR y hay tres claves que deben estar, las claves “FILES” , la clave “CONSOLES” y la clave “JSON”
Cada una albergara el nombre de las salidas a través de las cuales loguearan sus mensajes usando como separador de campo el carácter “,”
En el caso de CONSOLES, el valor de la clave es ON u OFF.

Ejemplo:

```
FILES=filer1,filer2,
JSON=json1,json2,
CONSOLES=ON
```

Para configurar un atributo de un Logger en particular hay que usar como clave el nombre del Logger, seguido de un “-” y luego el nombre del atributo a modificar, luego el signo “=” y luego el valor a setear.

Ejemplo:

```
FILES=filer1,filer2,
JSON=json1,
CONSOLES=ON
filer1-fileName=aFileName1
filer1-type=FILES
filer1-levelLog=DEBUG
filer1-formatDate=dd-M-yyyy hh:mm:ss
filer1-patternMessage=%d-%n-%p-%n-%t-%n-%m
filer1-separator=-
filer1-regularExpresion=^[a-zA-Z0-9]*$
filer1-customFilter=%d,%n,%p,%t,
filer1-%d=filter%d
filer2-fileName=aFileName2
filer2-type=FILES
filer2-levelLog=TRACE
filer2-formatDate=yyyy MMM dd
filer2-patternMessage=%d-%n-%p-%n-%m
filer2-separator=*
filer2-regularExpresion=^[a-z0-9]*$
filer2-customFilter=%d,%n,%p,%t,
filer2-%t=filter%t
console-fileName=console
console-type=CONSOLES
console-levelLog=DEBUG
console-formatDate=dd-M-yyyy hh:mm:ss
console-patternMessage=%g-%n-%d-%n-%p-%n-%t-%n-%m
console-separator=-
console-regularExpresion=^[a-zA-Z[:space:]]*$
console-customFilter=%p,
console-%p=ERROR
json1-fileName=aJsonName1
json1-type=JSON
json1-levelLog=ERROR
json1-formatDate=dd-M-yyyy hh:mm:ss
json1-patternMessage=%d-%n-%p-%n-%t-%n-%m
json1-separator=-
json1-regularExpresion=^[a-zA-Z0-9]*$
json1-customFilter=%p,
json1-%p=ERROR
```

Configuración del archivo XML

La estructura del archivo inicia con un tag raíz <Loggers> y luego una seria de tags hijos <logger> donde se establecerá la configuración de cada Logger.

Ejemplo

```
<loggers>
	<logger>
		<type>FILES</type>
		<name>LogParser</name>
		<output>
			<fileName>parserLog</fileName>
			<levelLog>DEBUG</levelLog>
			<formatDate>dd/M/yyyy</formatDate>
		 <patternMessage>%L-%n-%d-%n-%g-%n-%m</patternMessage>
			<separator>-</separator>
			<regularExpresion>^[a-zA-Z0-9]*$</regularExpresion>
			<customFilter>
				<valPattern pattern="%L">^\d+$</valPattern>
				<valPattern pattern="%g">LogParser</valPattern>
				<valPattern pattern="%m">^*patter14</valPattern>
			</customFilter>
		</output>
	</logger>
	<logger>
		<type>CONSOLES</type>
		<name>LogIndexer</name>
		<output>
			<fileName>indexerLog</fileName>
			<levelLog>ERROR</levelLog>
			<formatDate>yyyy MMM dd</formatDate>
			<patternMessage>%L-%n-%d-%n-%p-%n</patternMessage>
			<separator>*</separator>
			<regularExpresion>Expresion2</regularExpresion>
			<customFilter>
				<valPattern pattern="%T">patter2.1</valPattern>
				<valPattern pattern="%L">^\d+$</valPattern>
				<valPattern pattern="%p">patter2.3</valPattern>
				<valPattern pattern="%g">patter2.4</valPattern>
			</customFilter>
		</output>
	</logger>
</loggers>
```

Configuración por defecto

Esta configuración permitirá imprimir los mensajes a través de la consola usando el nivel más algo de Log(TRACE).

Parseo de los datos de configuración

En la primera versión del Tp, al solamente contar con un formato de configuración (archivo properties) se optó por tomar la configuración y por cada Log configurado generar un hash que albergaba su configuración y volcarlo en una lista. Ahora esa forma ya no quedaba tan cómoda al tener 3 fuentes distintas desde donde se podían tomar los datos.
Para solucionar esta situación se crearon 3 clases que se encargarían se obtener los datos de cada fuente, las clases son: LoaderDefaultConfiguration, LoaderXmlConfiguration y LoaderPropertierConfiguration
La lógica de cómo tratar cada formato de archivo está encerrada en cada una y cada una de ellas implementa la interface LoaderConfiguration que es usada por la clase ParserJsonConfig que es la encargada de obtener y almacenar los datos de cada Logger en formato Json

Integración de la configuración a la clase Logger

Finalizada la etapa de configuración ,para instanciar los distintos Loggers hay que instanciar la clase Configurator, llamar al método getConfigurationLogger() el cual devolverá un vector Json con la Configuración de cada Logger, cada índice de ese vector se pasa por parámetro en la instanciación de cada Logger, como se detalle en el ejemplo siguiente.

Ejemplo:

```
Configurator configurator = new Configurator();

JSONArray jsonArrayConfigLogger = configurator.getConfigurationLogger();

for(int i=0; i<jsonArrayConfigLoggers.size();i++){
        aLogger = new Logger(jsonArrayConfigLogger.get(i));
}
```

NOTA: es un ejemplo trivial, la intencionalidad solamente es mostrar cómo se instancian los Loggers, obviamente dentro del ciclo debería de existir alguna estructura que contenga los instancias creadas.

Uso de la API de Logger

Hay 7 niveles de log, los cuales son:

- OFF
- FATAL
- ERROR
- WARN
- INFO
- DEBUG
- TRACE

Para realizar cualquier mensaje de log, hay que instanciar la clase Logger y luego llamar el método SUFIJO_NIVEL_LOG(Mensaje), por ejemplo

```
aLogger.error("mensaje de prueba ERROR");
aLogger.debug("mensaje de prueba DEBUG");
```

NOTA IMPORTANTE: antes de finalizar el logger hay que llamar el método close para que se encargue de cerrar las impresoras abiertas.

```
aLogger.close();
```

Controlando los mensajes a Loguear

En la primera etapa del tp lo único que había que controlar por cada mensaje era que el nivel de Log del mensaje estuviera dentro del rango de nivel de Log del Logger, para ello se delegaba este control a la clase LevelLog, en esta oportunidad son 3 cosas las que hay que controlar por cada mensaje

- Nivel de mensaje
- Controlar si el mensaje se corresponde con la expresión regular establecía
- Controlar que el mensaje apruebe los Filter Custom

Para realizar esto último, se implementó la clase ControllerMessage que se encarga de realizar este control a través de la clase LogLevel y de la Clase MatcherExpresionRegular

Imprimir en Formato JSON

Para llevar a cabo esto se creó la clase JsonPrinter que hereda de la clase Printer, redefiniendo el método print() para formatear los mensajes a formato Json

Controlar Expresiones Regulares

Para tal fin se creó la clase MatcherExpresionRegular, que se instancia con la expresión regular a controlar y luego se llama al método **`checkFormatMessage(aMessage)`** y devuelve un boolean indicando si el mensaje cumple con el patrón o no.

Impacto de los cambio solicitados sobre el diseño de la Aplicación

Integración a SLF4J

Es posible utilizar el logger mediante la Simple Logging Facade for Java, para lo cual se implementó la interfaz al mismo. La misma consiste en tres clases:

- LoggerAdapter: adapter para realizar la integración entre la clase propia y la interfaz de la herramienta.
- LoggerFactory: factory para crear los distintos tipos de loggers.
- LoggerLoggerBinder: singleton que realiza el enlace entre la clase propia y la herramienta.

Impacto en la etapa de configuracion

- El hecho de tener 3 formatos distintos para poder configurar la aplicación nos llevó a la necesidad de implementar un **patrón Adapter** a través de la interface LoaderConfiguration, cada clase encargada de una determinada configuración (LoaderDefaultConfiguration, LoaderXmlConfiguration y LoaderPropertierConfiguration), implementa esa interface y se comunica con la clase JsonParserConfigurator, clase encargar de contener la data de todos los Loggers, por medio de la misma.
Esta interface permite en un futuro agregar más tipos de configuraciones sin realizar prácticamente ningún cambio sobre la aplicación, solamente habría que crear una nueva clase para tomar los datos desde el nuevo formato y servir los mismos a través de la interface LoaderConfiguration

- La exigencia del nuevo **campo %g** en el formato de los mensajes de Log nos llevó a la necesidad de crear una nueva clase llamada PatternNamePrinter, la misma hereda de la clase Pattern y simplemente recibe el mensaje a loguear y le concatena el nombre del logger sobre el cual está siendo impreso.
La instanciación de clase se hace a través de una clase que implementa el **patrón Factory** la misma es FactoryPattern, por lo que para adaptarla a la nueva necesidad simplemente se agregó una nueva sentencia dentro de Switch que pregunta por el pattern %g == PatternNamePrinter

Impacto en la instanciación de los Loggers

- **Instanciación:** La clase Logger anteriormente esperaba una lista de Hashes con la configuracion de cada Logger, ahora espera un vector Json.

- **Creacion de Pattenrs:** anteriormente al crearse los patterns solamente se los volcaba sobre una lista(para luego recorrerla e ir aplicándolos sobre el mensaje a loguear) ahora además de esto último, se consulta sobre cada patrón (%g,%m,etc) si tiene un filter custom configurado, de ser asi, se instancia un FilterCustom tomando como constructor la instancia asociada al patrón (%g,%m,etc) junto con la expresión regular a testear. Este FilterCustom luego es agregado a la clase ControllerCustom, que posee una lista en donde los va cargando

Impacto en la etapa de Log

- **Nuevo nivel de Log, TRACE:** para agregar el nuevo nivel de Log simplemente se procedió a actualizar el ENUM LevelLog agregando el atributo TRACE, el control del mismo se lleva a cabo por la misma clase por lo que el cambio no se extendió por fuera de la misma.

- **Posibilidad de loguear con Excepción:** se procedió, por cada nivel de Log, a crear un método que permitiera el paso por parámetro de la excepción requerida en el mensaje de Log

- **Controlar mensaje por Nivel de Log, Exp Reg y Filter Custom:** para dicho propósito se creó la clase ControllerMessage, el funcionamiento de la misma se detalla a continuación

Por cada mensaje a loguear se llamar al método isMessegeOk de la clase ControllerMessage, dicha clase se encarga de

- controlar el nivel de mensaje pasándolo a la clase LevelLog

- controlar que cumpla con la expresión regular pasándolo a la clase MatcherExpresionRegular

- controlar que cumpla con los filter custom, llamando al método validate() de la clase ControllerCustom, dicho método se encarga se recorrer la lista que posee con todos los filters custom instanciados y ejecutar el método validate() de cada uno
Si el mensaje supera este último test estará en condiciones de loguearse

Nuevo Formato de Impresión

- La clase Printer implementa un **Patron TemplateMethod**, el método a implementar es el print() y el close(), por lo que para poder desarrollar el log en formato Json se procedió a heredar de dicha clase y simplemente implementar el método print() y close().