**Логирование -** процесс записи каких-либо событий, которые происходят в коде.

**Лог (log)** - это хронологическая запись наиболее значимой информации о работе системы. Файлы содержат отчет обо всем, что происходило с системой: **какие действия совершали конкретные пользователи, когда это происходило, как система реагировала на события и т.д.**

**Уровни логирования.** Позволяют ограничить количество выводимой информации. Зачастую просто нет необходимости все подробно выводить, это может усложнить локализацию ошибок.

* **debug** — выводится информация, которая пригодится для отладки программы; Запись масштабных переходов состояний, например, обращение к базе данных, старт/пауза сервиса, успешная обработка записи и пр.
* **info** — Общая информация о выполнении программы. Разовые операции которые повторяются редко и не регулярно.;
* **warning** — нефатальное предупреждение; нештатная ситуация, потенциальная проблема, может быть странный формат запроса или некорректный параметр вызова.
* **error** — записи ошибок;
* **fatal** — фатальная ошибка; когда нет доступа к базе данных или сети, сервису не хватает места на жестком диске.
* **off** — сообщения не выводятся;
* **trace** — информация для точной отладки (вывод всего подряд). Самая полная информация; Полезен, когда сложно локализовать ошибку.
* **all** — выводится вся информация.

Для удобства обработки логов их делят на **типы**:

* **системные**, связанные с системными событиями,
* **серверные**, отвечающие за процесс обращения к серверу,
* **почтовые**, работающие с отправлениями,
* **логи баз данных**, которые отражают процессы обращения к базам данных,
* **авторизационные и аутентификационные**, которые отвечают за процесс входа, выхода из системы, восстановление доступа и пр.

**Преимущества логеров** по сравнению с sout:

**при фатальных ошибках**, нельзя узнать **из-за чего** она была вызвана

логер может работать отдельно от программы, на отдельном сервисе, поэтому мы и можем посмотреть, что за ошибки произошли в программе

позволяет менять уровни логирования или даже отключать его на лету.

**log4j**

Представляет собой набор API с помощью которых разработчики могут вставлять в свой код выражения, которые выводят отладочную информацию и конфигурировать этот вывод с помощью конфигурационных файлов.

**log4j** состоит из трех основных компонентов:

* **logger:** журналирующий компонент
* **appender:** элемент, определяющий куда нужно журналировать.
* **layout:** формат в котором будет производится журналирование

Logger журналирует какие-либо события в место назначения, определяемое элементом appender, в специально определенном формате, который задается элементом layout.

Log4j поддерживает запись в файлы (**FileAppender**), на консоль (**ConsoleAppender**), в базы данных (**JDBCAppender**), в журнал событий операционных систем семейства Windows NT/2000/XP (**NTEventLogAppender**), на SMTP-сервера (**SMTPAppender**), на удаленные сервера (**SocketAppender**) и не только.

Мы привязываем appender к logger, таким образом различные логгеры могут писать в различные места.

Каждый класс вашего приложения может иметь свой собственный logger или же быть привязанным к общему logger. Log4j предусматривает **корневой** **logger**, от которого будут наследоваться все создаваемые вами объекты этого типа.

Самый простой способ получить логгер - вызвать статический метод класса Logger, который получает объект по указанному имени. Если такой объект не существует, он будет создан. Но может существовать только один логгер с одним и тем же именем.

Log4j предоставляет несколько предопределенных layout’ов. Можно также создавать свои собственные. Layout может определять включать ли в выводимые записи дату и время, информацию об используемом объекте logger, информацию о номере текущей строки.

Каждому логгеру присваивается свой уровень. **По умолчанию** логгер берет уровень **своего родителя**.

Log4j предусматривает пять уровней журналирования:

* DEBUG
* INFO
* WARN
* ERROR
* FATAL

Таким образом логгер может выводить только события текущего и более высоких уровней. Мы можем **извне менять уровень** каждого из логгеров без необходимости вносить какие-то изменения в исходный код, **без** **необходимости перекомпилировать и разворачивать приложение заново**.

**Logback**

По сравнению с Log4j предлагает более быструю реализацию, предоставляет больше возможностей для конфигурации и более гибкий подход к архивированию старых файлов журналов.

Так же как и log4j состоит из трех основных компонентов:

* **Logger**
* **Appender**
* **Layout**

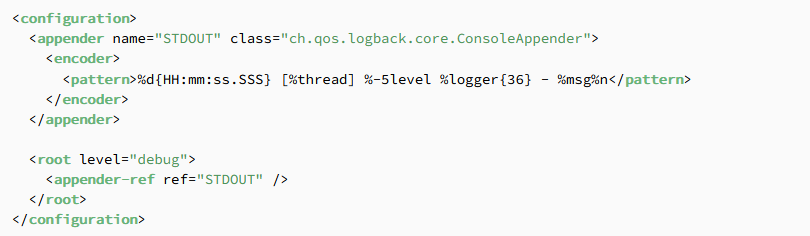
**Logger** – класс, с которым взаимодействует приложение для записи сообщений в лог.

**Appender** – помещает сообщение в место назначения (файл, консоль). Логгер может иметь более одного аппендера.

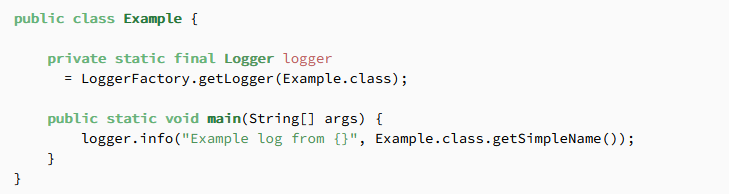
**Layout** – форматирует сообщение перед выводом. Logback поддерживает создание пользовательских классов форматирования и настройки существующих классов.

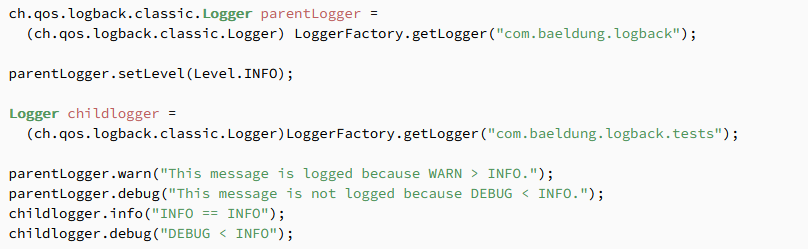
Logback использует Simple Logging Facade for Java (SLF4J) в качестве собственного интерфейса.

**Конфигурация XML**



Использование в классе





При создании логера мы передаем фабрике класс, в котором используется логгер. Это дает логгеру имя (имя можно также указать в качестве параметра).

Логгеры образуют иерархию:

* Логгер является предком, если его имя является префиксом потомка.
* Все логгеры являются потомками предопределенного корневого логгера.

Логгер имеет уровень, который может быть задан либо через конфигурацию, либо с помощью **Logger.setLevel().** Установка уровня в коде переопределяет конфиги.

**Уровни** в LogBack:

* TRACE
* DEBUG
* INFO
* WARN
* ERROR

Каждый уровень имеет соответствующий метод в логгере, который мы используем для вывода сообщения на этом уровне.

Если логгеру явно не присвоен уровень, он наследует уровень своего ближайшего предка. Корневой логгер по умолчанию имеет значение DEBUG.

Для **проверки уровня логирования** есть булевские методы:

**isDebugEnabled()** и аналогичные методы для каждого уровня.

Большинство полезных сообщений журнала требуют добавления строк.



Вышеприведенный пример влечет за собой выделение памяти, сериализацию объектов, конкатенацию строк и, возможно, последующую очистку мусора. Мы несем **затраты на создание сообщения** независимо от того, регистрирует ли логгер его или нет.

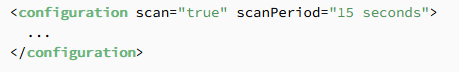
Logback в качестве альтернативы предлагает **параметризованные сообщения**.



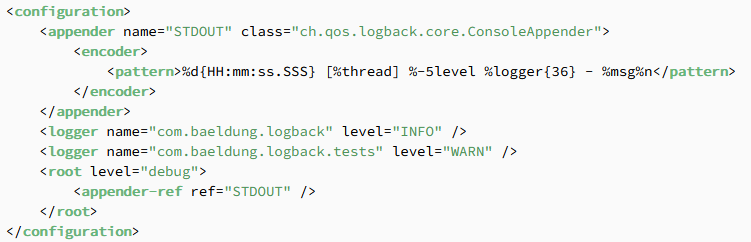
Скобки принимают любой объект и используют его метод toString() для построения сообщения только **после проверки** того, что сообщение необходимо.

Если передать Exception в качестве последнего аргумента, logback выведет **stacktrace**.

Указав параметр **scan** мы можем перезагрузить конфигурацию логирования во время работы приложения. По умолчанию конфигурационный файл сканируется каждые 60 секунд. Мы можем изменить этот интервал указав параметр **scanPeriod**.



В конфигурации мы можем **указать уровень** для любого логгера:

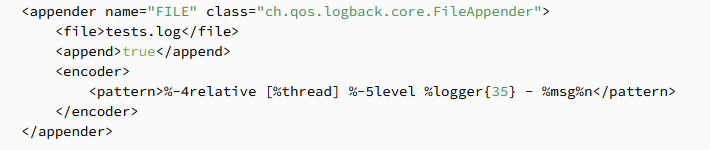
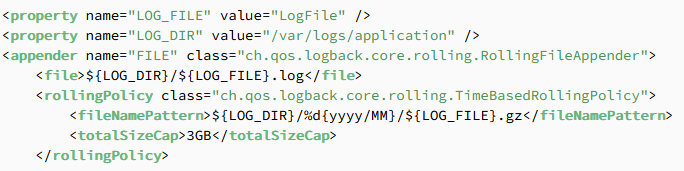


Мы можем использовать **переменные** в конфигурации, либо системные java свойства.





**Типы Appenders**

* **ConsoleAppender –** выводит сообщение в System.out или в System.err. Он использует OutputStreamWriter для буферизации ввода/вывода.
* **FileAppender –** добавляет сообщение в файл.  
  
* **RollingFileAppender –** записывает в файл. По выполнении какого либо условия заносит файл в архив, и создает новый основной файл. Это может быть размер файла, либо по истечении срока.  
  

В данном примере в начале каждого месяца основной файл переносится в /Current Year/Current Month/LogFile.gz и RollingFileAppender создает новый активный файл. Когда общий размер архивных файлов достигает 3гб, RollingFileAppender удаляет архивы по принципу fifo.

RollingFileAppender также имеет встроенную поддержку **сжатия файлов**. Он сжимает наши свернутые файлы, потому что мы назвали их LogFile.gz.

Logback также предлагает **SizeAndTimeBasedRollingPolicy**, которая будет выполнять сворачивание на основе текущего **размера** файла журнала, а также времени. Он также предлагает **FixedWindowRollingPolicy**, которая перекатывает имена файлов журнала при каждом запуске регистратора.

Мы также можем написать свою **собственную RollingPolicy**.

Мы можем создавать **собственные аппендеры**, расширяя один из базовых классов аппендеров Logback.

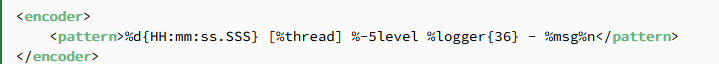
* **SocketAppender**
* **SMTPAppender**

**Layouts** так же являются расширяемыми, и мы можем создавать свои собственные. Однако стандартный **PatternLayout** предлагает то, что нужно большинству приложений и даже больше.

Обычно указывается шаблон, на основе которого PatternLayout форматирует сообщения.

PatternLayout распознает слова с символом %.

* **%d{HH:mm:ss.SSS}** – временная метка с часами, мин, сек и миллисек.
* **[%thread]** - имя потока, генерирующего сообщение журнала, заключенное в квадратные скобки
* **%-5level** - уровень события протоколирования, расширенный до 5 символов
* **%logger{36}** - имя регистратора, усеченное до 35 символов
* **%msg%n** - сообщения журнала, за которыми следует символ разделителя строк, зависящий от платформы



**Популярные ошибки в логировании**

* **Логирование всего подряд**. Например, если залогировать чьи-то личные данные и это каким-то образом всплывет на поверхность, будут реальные проблемы.
* **Избыток логирования**. Не стоит логировать каждый шаг, который чисто теоретически может быть важным. Есть правило: логи могут нагружать работоспособность не более, чем на 10%. Иначе будут проблемы с производительностью.
* **Логирование всех данных в один файл**. Это приведет к тому, что в определенный момент чтение/запись в него будет очень сложной, не говоря о том, что есть ограничения по размеру файлов в определенных системах.
* **Использование неверных уровней логирования**. У каждого уровня логирования есть четкие границы, и их стоит соблюдать. Если граница расплывчатая, можно договориться какой из уровней использовать.

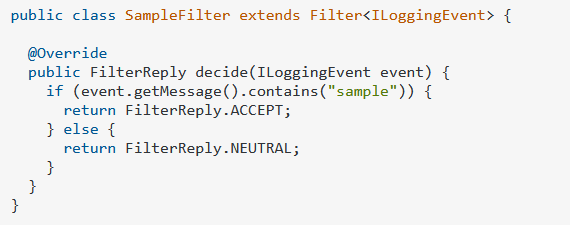
**Фильтры**

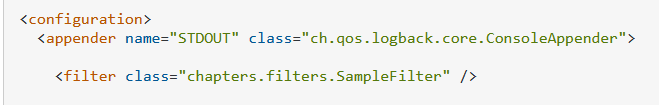
Обычные фильтры в logback ширяют абстрактный класс **Filter**, в котором есть метод **decide(),** принимающего в качестве параметра экземпляр **ILoggingEvent**. Фильтры организованы в виде упорядоченного списка.

Метод decide возвращает одно из значений перечисления **FilterReply**:

* **DENY** – событие журнала немедленно отбрасывается без обращения к остальным фильтрам.
* **NEUTRAL** – обращается к следующему фильтру в списке. Если больше нет фильтров событие логирования обрабатывается.
* **ACCEPT** – событие журнала обрабатывается немедленно, пропуская вызов остальных фильтров.

**Фильтры** **могут быть добавлены** к экземплярам Appender. Можно фильтровать события по содержимому сообщения, времени суток и любой другой части события.





Есть встроенные **LevelFilter**, **ThresholdFilter**

**TurboFilter**

Объекты TurboFilter привязаны к контексту протоколирования. Следовательно, они вызываются не только при использовании данного аппендера, но и каждый раз, когда выдается запрос на ведение журнала. Их область применения шире, чем фильтров, привязанных к аппендеру.

Что еще более важно, они **вызываются** **до создания** объекта **LoggingEvent**. Объекты TurboFilter не требуют инстанцирования события протоколирования для фильтрации запроса протоколирования. Таким образом, турбофильтры **предназначены** для **высокопроизводительной фильтрации** событий протоколирования еще **до создания этих событий**.

Есть встроенные фильтры DuplicateMessageFilter.

**Log Analyzers**

Специальные программы, которые собирают, обрабатывают и хранят логи. Собирать статистику и анализировать. Позволяют оценивать данные с нескольких серверов. Упрощают поиск .

**Webalizer**, **Loganalizer**, **Analog**, **AWStats**