ErrorHandling and Application Logger

The logging system of translate5 is currently completely reworked.

The architecture is similar as the Zend Logger from Zend_Log. On investigation if Zend_Log can be used for Translate5 needs, it turned out that translate5s special needs to the logger was not implementable by just reusing / extending the Zend_Log facility. So a conceptional adoption was done.

Wording

Since not only errors are logged, instead of "Error" or "ErrorCodes" we are talking about "Event" and "EventCodes" in the context of logging.

In the context of Exception "error" can be used, since exceptions are errors.

Basic Class Architecture

Logger itself

DebugLevels

The following levels are defined as class constants prefixed with LEVEL_ in ZfExtended_Logger class:

Level	As Number	Usage example
FATAL	1	FATAL PHP errors, other very fatal stuff, wrong PHP version or so
ERROR	2	common unhandled exceptions
WARN	4	User locked out due multiple wrong login, A user tries to edit a task property which he is not allowed to change
INFO	8	user entered wrong password, wanted exceptions BadMethodCall on known non existent methods (HEAD), other known and wanted exceptions
DEBUG	16	debugging on a level also useful for SysAdmins, for example user logins
TRACE	32	tracing debugging on a level only useful for developers

ZfExtended_Logger

- The logger itself. Methods of that class are used to log custom messages / data and exceptions / errors.
- A reusable default instance is located in the registry: Zend_Registry::get('logger');
- The default instance is initialized via the Resource Plugin ZfExtended_Resource_Logger and is therefore configured from the .ini configuration system as other Zend Resources too.
    - Own instances with custom configuration can be instanced when needed
    - The default instance is also accessible via Zend getResource stuff
- A Logger instance can have multiple writer instances which writes the received log messages to different targets - as configured.
    - One can also define additional Logger instances if needed with a different config as the default one
- provides functions like fatal, error, warn, info, debug, trace for direct log output
- provides a function exception to log exceptions, this is automatically invoked in the final exception handler but can also be used manually for catched exceptions, which have to be logged but should not stop the request then.

```
// use the default logger instance:
$logger = Zend_Registry::get('logger');
/* @var $logger ZfExtended_Logger */
$logger->info('E1234', 'This is the message for the log where {variable} can be used.', [
        'mydebugdata' => "FOOBAR can be an object or array too",
    'variable' => "This text goes into the above variable with curly braces",
]);
```

ZfExtended_Logger_Event

- A container class for the event information.
- Is used internally to communicate the event from the logger to the different writers.

Log Writers

A ZfExtended_Logger can have multiple instances of writers. The writers are responsible for filtering (according to the configuration) and writing the log message to the specific backend.

So each writer can have a different filter configuration, one can configure one writer to listen on all import errors on error level debug and send them to one address, another writer can listen to all events on level > warn to log them as usual.

ZfExtended_Logger_Writer_Abstract

Abstract class for each specific writer.

ZfExtended_Logger_Writer_Database

- Logs the received events into the central log database table Zf_errorlog.
- This Writer can be configured with additional writers again, which receive events only once. All repeated events are stored in the database, but for example a email is send only once.

ZfExtended_Logger_Writer_DirectMail

Sends the event directly to the configured email address. No repetition recognition of events (spam protection) is done here.

Multiple E-Mail Receivers can be configured. By multiple DirectMail instances with different filters, user A can receive errors about topic A and user B about topic B.

ZfExtended_Logger_Writer_ErrorLog

Writes the events to the error_log configured in PHP.

ZfExtended_Logger_Writer_ChromeLogger

Writes the events to the chrome developer toolbar JS log. The Addon "Chrome Logger" must be installed therefore in the browser.

Recommended not to be used in production environments!

editor_Logger_TaskWriter

translate5 specific writer which logs all events with a field "task" containing a editor_Models_Task instance in the extra data to a special task log table.

editor_Models_Logger_Task extending ZfExtended_Models_Entity_Abstract and editor_Models_Db_Logger_Task extending Zend_Db_Table_Abstract are used to save to and load from the task log table.

Configuration of LogWriters

The documentation about the configuration of the LogWriters is here.

Logging Database Tables

Currently there are two database tables receiving events:

- Zf_errorlog via ZfExtended_Logger_Writer_Database
- LEK_task_log via editor_Logger_TaskWriter

Whats the difference?

Zf_errorlog receives basicly ALL events, limited by the configured log level of events to be received.

LEK_task_log receives:

- only events which contain a task in its extra data
- only events higher as the configured log level for that writer
- The content of that table is available via API to provide information to the tasks in the GUI

usage hint of dealing with large data

```
-- instead of
select * from Zf_errorlog;
-- use for better readability:
select * from Zf_errorlog\G
```

Exceptions

In translate5 to less different exception types were used till now. Mostly just a ZfExtended_Exception was used.

To be more flexible in filtering on logging exceptions but also on internal exception handling with try catch more exceptions are needed.

See the following page for the usage and the idea behind exceptions in translate5.

EventCodes

The EventCodes used in exceptions and other logging usages are defined via the ErrorCodes listed and maintained in confluence..

Prevent Log Duplications

One problem are duplicated log entries flooding the log if for example an attached service is not reachable anymore.

To prevent this, specific event codes can be configured, so that for that codes no additional log is send, if the same error pops up multiple times. The time interval for recognition as duplicate is currently fixed to 5 minutes. Since this affects in general use only some errors, the design reason to set this per event code, is just to prevent logging loss in the case that wanted errors happens multiple times.

```
// activates the duplication recognition by the formatted and rendered error message.
// for places where event codes are added with editor_Services_Connector_Exception::addCodes just add the
addDuplicates calls directly afterwards.
// so the messages 'Test Error for "FOO"' and 'Test Error for "BAR"' are considered different, even if the
event code is the same, and also the message template is the same: 'Test Error for "{variable}"'
ZfExtended_Logger::addDuplicatesByMessage('E1317', 'E1318');

//Configured that way, the variables in the message will play no role anymore:
ZfExtended_Logger::addDuplicatesByEcode('E1319', 'E1234');

// if duplication recognition should be used for events defined in exceptions, just override the template
function ZfExtended_ErrorCodeException::setDuplication:

class Demo_Exception extends ZfExtended_ErrorCodeException {

    static protected $localErrorCodes = [
        'E1234' => 'Demo Error.',
    ];

    protected function setDuplication() {
        parent::setDuplication();
        ZfExtended_Logger::addDuplicatesByMessage('E1234');
    }
}
```

Domains

The domains are used to classify and therefore filter exceptions and log entries. In the filter process the event code is added to the domain internally, that enables filtering for concrete event codes too.

How domains should be defined and used see the following examples. In general the domains are dot separated strings. Starting on the left side from the general identifier to more specific identifier on the right.

Three starting domains are defined so far: "editor" for the editor module, "core" for core code, "plugin" for plugin code.

Domain Examples:

```
core                                ./library/ZfExtended/Exception.php
core                                ./library/ZfExtended/Logger.php
core.api.filter                                          ./library/ZfExtended/Models/Filter/Exception.php
core.entity                         ./library/ZfExtended/Models/Entity/Exceptions/IntegrityConstraint.php
core.entity                         ./library/ZfExtended/Models/Entity/Exceptions/IntegrityDuplicateKey.php
core.worker                         ./library/ZfExtended/Worker/TriggerByHttp.php
core.authentication
editor.customer                     ./application/modules/editor/Controllers/CustomerController.php
editor.export.difftagger            ./application/modules/editor/Models/Export/DiffTagger/Exception.php
editor.export.fileparser            ./application/modules/editor/Models/Export/FileParser/Exception.php
editor.import                       ./application/modules/editor/Models/Import/Exception.php
editor.import.configuration         ./application/modules/editor/Models/Import/ConfigurationException.php
editor.import.fileparser            ./application/modules/editor/Models/Import/FileParser/Exception.php
editor.import.fileparser.csv        ./application/modules/editor/Models/Import/FileParser/Csv/Exception.php
editor.import.fileparser.sdlxliff   ./application/modules/editor/Models/Import/FileParser/Sdlxliff/Exception.php
editor.import.fileparser.xlf        ./application/modules/editor/Models/Import/FileParser/Xlf/Exception.php
editor.import.metadata              ./application/modules/editor/Models/Import/MetaData/Exception.php
editor.import.metadata.pixelmapping ./application/modules/editor/Models/Import/MetaData/PixelMapping.php
editor.import.relais                ./application/modules/editor/Models/RelaisFoldertree.php
editor.languageresource.service     ./application/modules/editor/Services/Manager.php
editor.languageresource.service     ./application/modules/editor/Services/NoServiceException.php
editor.languageresource.taskassoc   ./application/modules/editor/Controllers
/LanguageresourcetaskassocController.php
editor.segment                      ./application/modules/editor/Models/Segment/Exception.php
editor.segment                      ./application/modules/editor/Models/Segment/UnprocessableException.php
editor.segment.pixellength          ./application/modules/editor/Models/Segment/PixelLength.php
editor.terminology                                      ./application/modules/editor/Models/Term
/TbxCreationException.php
editor.task                         ./application/modules/editor/Controllers/TaskController.php
editor.task.exceleximport           ./application/modules/editor/Models/Excel/ExImportException.php
editor.workflow                     ./application/modules/editor/Logger/Workflow.php
editor.workflow.notification        ./application/modules/editor/Workflow/Actions/Abstract.php
plugin.groupshare                   ./application/modules/editor/Plugins/GroupShare/Exception.php
plugin.groupshare.httpapi.tmservice ./application/modules/editor/Plugins/GroupShare/HttpApiTMService.php
plugin.groupshare.httpapi.web       ./application/modules/editor/Plugins/GroupShare/HttpApiWebAPI.php
plugin.groupshare.token             ./application/modules/editor/Plugins/GroupShare/ExceptionToken.php
plugin.matchanalysis                ./application/modules/editor/Plugins/MatchAnalysis/Exception.php
plugin.okapi                        ./application/modules/editor/Plugins/Okapi/Exception.php
```

Use the logger facility just to log stuff

The default logger instance is generally available in the registry:

```
$log = Zend_Registry::get('logger');
/* @var $log ZfExtendend_Logger */
$log->error("TODO"); // logs an error
// must be enabled in the filters
$log->debug("TODO"); // logs a debug statement
$log->trace("TODO"); // logs a debug statement with stack trace
$log->logDev($data1, $data2, ...); // a convienent replacement for error_log(print_r($data, 1)); Only for
development, should not be comitted!

// the WorkflowLogger - dedicated to translate5 tasks and workflow stuff - must be instanced manually:
// TODO
```

Use the ZfExtended_Logger_DebugTrait for debugging

The ZfExtended_Logger_DebugTrait provides a reusable, unified way to get a logger instance into a class for logging and debugging.

Example:

```
class Class_Or_Plugin_In_Which_I_Want_To_Log {
    use ZfExtended_Logger_DebugTrait; // use the debug trait to get several things

        public function __construct() {
                //before first usage init the logger:
        $this->initLogger('E1100', 'plugin.name.special.sub.stuff', 'plugin.name', 'Plug-In NAME: ');
                //        E1100  specific error code for the domain to be logged (or general code if applicable)
                //        plugin.name.special.sub.stuff  the domain of the current scope to be logged
                //  plugin.name  the root domain which is used as check in the logger writer configuration. If
this string occurs somewhere in the writer domain filters, the debug and trace function will do something
                //  Plug-In NAME:  a string prefix which is added to each debug and trace message.
    }

        public function doSomethingToUseTheLogger() {
                $this->log->warn("EXXXX", "Message", ['data' => 123]);  // by the trait $this->log is
initialized with a logger using the domain "plugin.name.special.sub.stuff"
                $this->debug('Debug message', ['debugdata' => 123]);    // the trait provides the debug
function, which uses the above configured ECode and prefix. Also it uses the configured domain.

// this debug statement can remain in the code.

// It will only do something if one of the log writers is configured with a filter with a domain containing
'plugin.name' and level debug

// so multiple debug statements should not pull down the performance and will not waste the log by the default
                $this->trace('Message', ['debugdata' => 123]);                        // basicly the same as
debug() but with complete stack trace in the log. filter level must be trace so that this functions is soing
something.
    }
```

TODO

Exception Definition / Usage:

- Before the refactoring we had only a few Exceptions

- Now for each domain (import / export / Plugin XY) one or more Exceptions should be defined.

- Each Exception has a different domain (example: import.fileparser.sdlxliff) for filtering.

- Also the Exception carries Semantik through its type, only if needed. Example:

  Plugin_Demo_Exception > A general Exception in the Demo Plugin Scope

Plugin_Demo_NoConnectionException > On more specific errors (no connection established) specific exceptions can be created. Why that: If we can / want to handle that exceptions differently we can do that:

```
try {
    $this->foo();
}
catch(Plugin_Demo_NoConnectionException $e) {
    //handle the no connection error
    logger::exceptionHandled($e); //logs the exception on level debug and marks the logged Exception as Handled.
}
// Plugin_Demo_Exception are handled via the final handler and stops the PHP request
```

debug