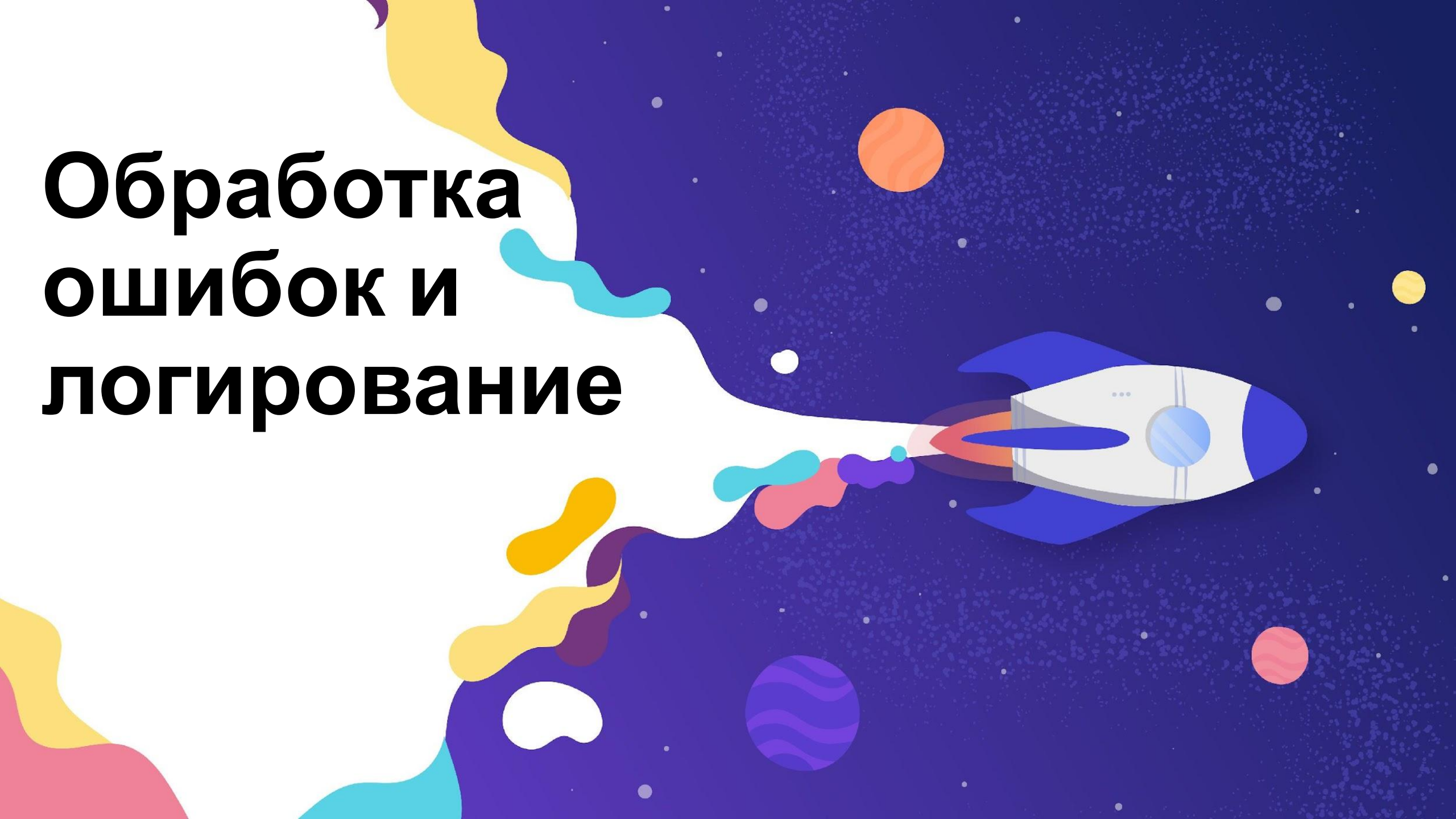
Обработка ошибок и логирование



Обработка ошибок

Первоначальная поддержка обработки исключений была введена в язык с 5 версии PHP, с двумя простыми встроенными классами исключений - Exception и ErrorException, с поддержкой дополнительных классов через SPL.

Хотя PHP 7 предоставляет классы Error и Exception, рассмотрим интерфейс Throwable .

Оба класса и Error, и Exception классы реализуют Throwable интерфейс - это основа для любого объекта , который может быть брошен с помощью оператора throw.

Он не может быть реализован непосредственно в классах пользовательского пространства, только через расширение класса Exception.

Кроме того, он обеспечивает единую точку для отлова обоих типов ошибок в одном выражении.

Обработка ошибок

```
try {
  // код, который может вызвать исключение
} catch(Throwable $exception) {
  // обработка исключения
}
```

Эта конструкция в общем варианте состоит из двух блоков - try и catch. В блок try помещается код, который потенциально может вызвать исключение. А в блоке catch помещается обработка возникшего исключения.

Причем можно задать логику обработки для каждого типа исключения. Конкретный тип исключения, который необходимо обработать, указывается в круглых скобках после оператора catch

Обработка ошибок

```
try {
    // код, который может вызвать исключение
} catch(MyAwesomeException $exception) {
    // обработка исключения
} catch(YourAwesomeException $exception) {
    // обработка исключения
} catch(Throwable $exception) {
    // обработка исключения
}
```

После названия типа указывается переменная этого типа (в данном случае $exeption), которая будет хранить информацию об исключении и которую мы можем использовать при обработке исключения.
Блоки catch с более конкретными типами ошибок и исключений должны идти в начале, а более с более общими типа - в конце.

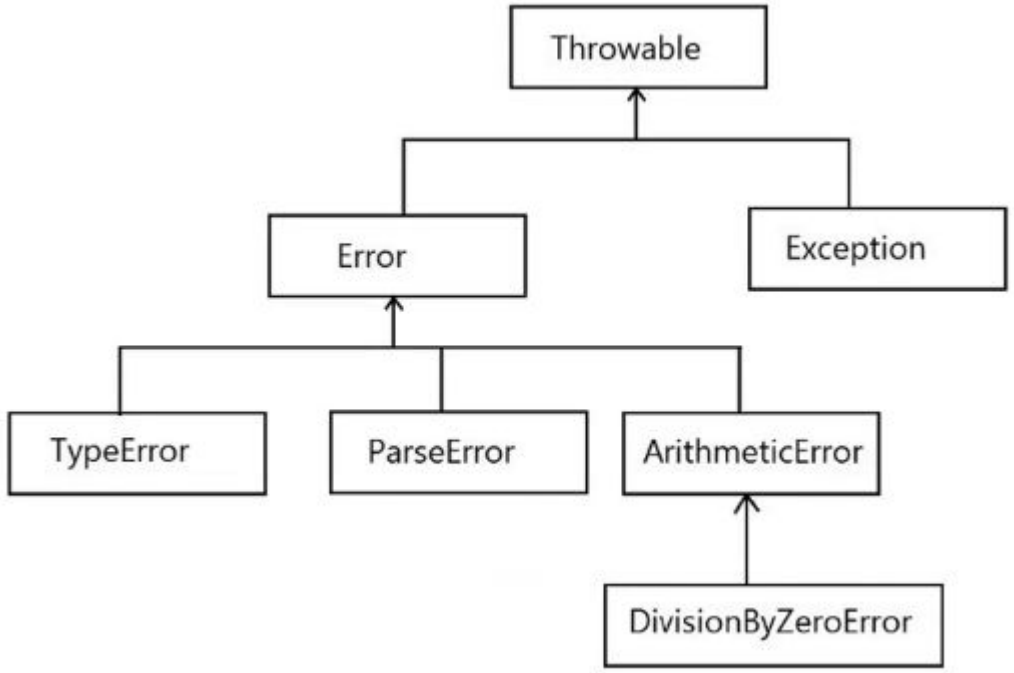
Обработка ошибок

Если в блоке try при выполнении кода возникает ошибка, то блок try прекращает выполнение и передает управление блоку catch, который обрабатывает ошибку. А после завершения выполнения кода в блоке catch программа продолжает выполнять инструкции, которые размещены после блока catch.

Если в блоке try при выполнении кода не возникает ошибок, то блок catch не выполняется, а после завершения блока try программа продолжает выполнять инструкции, которые размещены после блока catch.

Обработка ошибок

В PHP для разных ситуаций есть множество типов, которые описывают ошибки. Все эти встроенные типы применяют интерфейс Throwable



Все типы делятся на две группы: собственно ошибки (класс Error) и собственно исключения (класс Exception). А от классов Error и Exception наследуются классы ошибок и исключений, которые описывают конкретные ситуации. Например, от класса Error наследуется класс ArithmeticError, который описывает ошибки, возникающие при выполнении арифметических операций.

А от класса ArithmeticError наследуется класс DivisionByZeroError, который представляют ошибку при делении на ноль.

Обработка ошибок. Информация об ошибке

Интерфейс Throwable предоставляет ряд методов, которые позволяют получить некоторую информацию о возникшем исключении:

- getMessage(): возвращает сообщение об ошибке
- getCode(): возвращает код исключения
- getFile(): возвращает название файла, в котором возникла ошибка
- getLine(): возвращает номер строки, в которой возникла ошибка
- getTrace(): возвращает трассировку стека
- getTraceAsString(): возвращает трассировку стека в виде строки

Обработка ошибок. Информация об ошибке

```php
try {
    $result = 5 / 0;
    echo $result;
} catch (DivisionByZeroError $exception) {
    echo "Сообщение об ошибке: " . $exception->getMessage() . "<br>";
    echo "Файл: " . $exception->getFile() . "<br>";
    echo "Номер строки: " . $exception->getLine() . "<br>";
}
```

Результат:
Сообщение об ошибке: Division by zero

Файл: index.php

Номер строки: 11

Обработка ошибок. Блок finally

Конструкция try..catch также может определять блок finally. Этот блок выполняется в конце - после блока try и catch вне зависимости, возникла или нет ошибка. Нередко блок finally используется для закрытия ресурсов, которые применяются в блоке try.

```
try {
    $result = 5 / 0;
    echo $result;
} catch (DivisionByZeroError $exception) {
    echo "Ошибка при выполнении программы";
} finally {
    echo "Блок finally<br>";
}
echo "Конец работы программы";
```

Результат:

Ошибка при выполнении программы
Блок finally
Конец работы программы

Обработка ошибок. Константы

В PHP много констант, которые используются в отношении ошибок. Эти константы используются при настройке PHP для скрытия или отображения ошибок определенных классов.
Вот некоторые из наиболее часто встречающихся кодов ошибок:

- E_DEPRECATED - интерпретатор сгенерирует этот тип предупреждений, если вы используете устаревшую языковую функцию. Сценарий обязательно продолжит работать без ошибок.
- E_STRICT - аналогично E_DEPRECATED, - указывает на то, что вы используете языковую функцию, которая не является стандартной в настоящее время и может не работать в будущем. Сценарий будет продолжать работать без каких-либо ошибок.
- E_PARSE - ваш синтаксис не может быть проанализирован, поэтому ваш скрипт не запустится. Выполнение скрипта даже не запустится.

Обработка ошибок. Константы

- E_NOTICE - движок просто выведет информационное сообщение. Выполнение скрипта не прервется, и ни одна из ошибок не будет выдана.
- E_ERROR - скрипт не может продолжить работу, и завершится. Выдает ошибки, а как они будут обрабатываться, зависит от обработчика ошибок.
- E_RECOVERABLE_ERROR - указывает на то, что, возможно, произошла опасная ошибка, и движок работает в нестабильном состоянии. Дальнейшее выполнение зависит от обработчика ошибок, и ошибка обязательно будет выдана.

Обработка ошибок. Функция обработки

Функция set_error_handler() используется, чтобы сообщить PHP, как обрабатывать стандартные ошибки, которые не являются экземплярами класса исключений Error. Невозможно использовать функцию обработчика ошибок для фатальных ошибок. Исключения ошибок должны обрабатываться с помощью операторов try/catch.

set_error_handler() принимает callback функцию в качестве своего параметра. Callback-функции в PHP могут быть заданы двумя способами: либо строкой, обозначающей имя функции, либо передачей массива, который содержит объект и имя метода (именно в этом порядке).

Обработка ошибок. Функция обработки

Возможным является указать защищенные и приватные методы для callable в объекте. Также возможно передать значение null, чтобы указать PHP вернуться к использованию стандартного механизма обработки ошибок.

Если обработчик ошибок не завершает программу и возвращает результат, сценарий будет продолжать выполняться со строки, следующей за той, где произошла ошибка.

PHP передает параметры в пользовательскую функцию обработчика ошибок, которые могут быть опционально заданы в сигнатуре этой функции.

Обработка ошибок. Функция обработки

```php
function myCustomErrorHandler(
    int $eNumber,
    string $eMessage,
    string $fileName,
    int $line
) {
    echo "Error #[$eNumber] occurred in [$fileName] at line [$line]: [$eMessage]";
}

set_error_handler('myCustomErrorHandler');

try {
    why;
} catch (Throwable $e) {
    echo 'Моя ошибка: ' . $e->getMessage();
}
```

Результат:

Моя ошибка: Error #[2] occurred in [index.php] at line [3]: [Use of undefined constant why - assumed 'why' (this will throw an Error in a future version of PHP)]

Обработка исключений. Функция обработки

Любое исключение, которое не было перехвачено, приводит к фатальной ошибке.
Чтобы перехватывать исключения, которые не перехватываются в блоках перехвата, необходимо установить функцию в качестве обработчика исключений по умолчанию.

Для этого необходимо использовать функцию set_exception_handler() , которая принимает вызываемый элемент в качестве параметра. Сценарий завершится после того, как вызов будет выполнен.

Функция restore_exception_handler() вернет обработчик исключений к его предыдущему значению.

Обработка исключений. Функция обработки

```php
class NotImplementedFieldException extends Exception
{
    public function __construct(string $fieldName)
    {
        $this->message = "Поля {$field} не существует";
    }
}

try {
    echo $user->phone;
} catch (SomeOtherException $exception) {
    echo $exception->getMessage();
}

function exceptionHandler(Throwable $exception) {
    echo "Uncaught exception: " , $exception->getMessage(), "\n";
}

set_exception_handler('exceptionHandler');
```

В данном примере NotImplementedField исключение не будет перехвачено, т.к. в блоке catch производится обработка другого типа исключения.

Но благодаря использованию set_exception_handler, куда передается пользовательская функция обработки исключений, данное исключение не приведет к фатальной ошибке

Генерация исключений

PHP по умолчанию представляет ситуации, в которых автоматически генерируются ошибки и исключения, например, при делении на ноль. Но иногда возникает необходимость самим вручную сгенерировать исключение.

В качестве примера можно привести работу магических методов __get и __set: в случае невозможности получить или установить значение пробрасывалось исключение типа Exception.

Хорошим стилем является проброс конкретного исключения для конкретной ситуации, либо группы ситуаций. В случае, описанном выше, было бы корректно пробросить исключение типа NotImplementedFieldException, которое бы обладало более специфической информацией об ошибке.

Но такого класса исключения не существует в PHP. Для таких целей можно создавать собственные класс, унаследовав его от базового.

Генерация исключений

```php
class NotImplementedFieldException extends Exception
{
    public function __construct(string $fieldName)
    {
        $this->message = "Поля {$fieldName} не существует";
    }
}

class User
{
    public function __get(string $name)
    {
        if (!in_array($name, $this->fields)) {
            throw new NotImplementedFieldException( $name);
        }
    }
}
try {
    echo $user->phone;
} catch (NotImplementedFieldException $exception) {
    echo $exception->getMessage();
}
```

Результат:
Поля phone не существует

Логирование

Для того, чтобы объяснить что такое логирование или журналирование, необходимо сначала дать понятие процессу отладки: ведь прямая задача логирования - упростить и вообще, сделать возможным процесс отладки приложения.

Отладку работы приложения можно сравнить с диагностикой заболевания врачом. При приеме пациента выясняются сама проблематика, симптомы, уточняется, какие действия или бездействия совершал пациент до того, как возникла проблема.

Аналогичным образом происходит диагностика проблем в работе приложения: выясняется в чем проблема, каковы ее последствия, и самое важное - что происходило в системе до ее возникновения.

Логирование

В отличии от пациента компьютер не может рассказать, например, болит или нет при совершении какого-то действия.

Поэтому, при разработке программ, необходимо вести запись операций, которые потенциально могут вызвать проблемы в системе.

Например, если приложение взаимодействует со сторонними сервисами (очень высокий риск отказа), стоит вести запись в виде запрос - ответ.

Ведение подобных записей называется журналированием или логированием, а сами записи - логами.

Логирование

Хорошо организованное логирование позволяет, как минимум, следующее:

- Знать о том, что что-то идёт не так, как задумано (есть ошибки)
- Знать подробности ошибки, которые помогут сказать, с кем и где произошла ошибка, и не допустить повтора
- Знать о том, что всё идёт как задумано (access.log, debug-, info-уровни)

Сама по себе запись в лог вам всего этого не скажет, но с помощью логов будет возможность самостоятельно узнать подробности событий, либо настроить систему мониторинга логов, которая будет иметь возможность оповещать о проблемах. Если сообщения в логах сопровождаются достаточным объемом контекста, то это значительно упрощает отладку, потому что вам будет доступно больше данных о ситуации, в которой произошло событие.

Логирование

Чем писать и что писать

Частью php-сообщества разработаны рекомендации по некоторым задачам написания кода. Одна из таких рекомендаций PSR-3 Logger Interface. Она как раз описывает то, чем нужно логировать. Для этого разработан интерфейс `Psr\Log\LoggerInterface` пакета "psr/log". При его использовании нужно знать о трёх составляющих события:

1. Уровень — важность события
2. Сообщение — текст, описывающий событие
3. Контекст — массив дополнительной информации о событии

Логирование. Уровни логирования

В различных средах работы приложения (например, среда разработки, тестирования или продакшн) необходимо разделять объем и суть информации, которую нужно хранить в логах.

Одно дело, когда разработчик, работая в своей локальной среде, делает запись в лог какой-то операции для отладки, которая потом не пригодится, а другое - такую запись совершат тысячи пользователей в продакшн среде.

Избыточность логирования - один из краеугольных камней, решить который помогает внедрение различного уровня логирования.

Как в примере выше, на локальной среде допустимо логирование отладочной информации, а на продакшн - нет. Соответственно, для локальной среды должен быть включен данный уровень логирования, а на продакшн - отключен.

Логирование. Уровни логирования

В общепринятой концепции RFC 5424 — The Syslog Protocol выделяют следующие уровни логирования:

- emergency — это уровень для внешних систем, которые могут посмотреть на систему и точно определить, что она полностью не работает, либо не работает её самодиагностика.
- alert — система сама может продиагностировать свое состояние, например, задачей по расписанию, и в результате записать событие с этим уровнем. Это могут быть проверки подключаемых ресурсов или что-то конкретное, например, баланс на счету используемого внешнего ресурса.

Логирование. Уровни логирования

- critical — событие, когда сбой дает компонент системы, который очень важен и всегда должен работать. Это уже сильно зависит от того, чем занимается система. Подходит для событий, о которых важно оперативно узнать, даже если оно произошло всего раз.
- error — произошло событие о, котором при скором повторении нужно сообщить. Не удалось выполнить действие, которое обязательно должно быть выполнено, но при этом такое действие не попадает под описание critical. Например, не удалось сохранить аватарку пользователя по его запросу, но при этом система не является сервисом аватарок, а является чат-системой.

Логирование. Уровни логирования

- warning — события, для немедленного уведомления о которых нужно набрать значительное их количество за период времени. Не удалось выполнить действие, невыполнение которого ничего серьезного не ломает. Это всё ещё ошибки, но исправление которых может ждать рабочего расписания. Например, не удалось сохранить аватарку пользователя, а система — интернет-магазин. Уведомление о них нужно (при высокой частоте), чтобы узнать о внезапных аномалиях, потому что они могут быть симптомами более серьезных проблем.

Логирование. Уровни логирования

- notice — это события, которые сообщают о предусмотренных системой отклонениях, которые являются частью нормального функционирования системы. Например, пользователь указал неправильный пароль при входе, пользователь не заполнил отчество, но оно и не обязательно, пользователь купил заказ за 0 рублей, но у вас такое предусмотрено в редких случаях. Уведомление по ним при высокой частоте тоже нужно, так как резкий рост числа отклонений может быть результатом допущенной ошибки, которую срочно нужно исправить.

Логирование. Уровни логирования

- info — события, возникновение которых сообщает о нормальном функционировании системы. Например, пользователь зарегистрировался, пользователь приобрел товар, пользователь оставил отзыв. Уведомление по таким событиями нужно настраивать в обратном виде: если за период времени произошло недостаточное количество таких событий, то нужно уведомить, потому что их снижение могло быть вызвано в результате допущенной ошибки.

Логирование. Уровни логирования

- debug — события для отладки какого-либо процесса в системе. При добавлении достаточного количества данных в контекст события можно произвести диагностику проблемы, либо заключить об исправном функционировании процесса в системе. Например, пользователь открыл страницу с товаром и получил список рекомендаций. Значительно увеличивает количество отправляемых событий, поэтому допустимо убирать логирование таких событий через некоторое время. Как результат, количество таких событий в нормальном функционировании будет переменным, тогда и мониторинг для уведомления по ним можно не подключать.

Логирование. Сообщение и контекст

Сообщение об ошибке должно быть кратким, четким и емким, как приказ в армии. Например, “Ошибка: пользователь не смог загрузить аватарку”. Остальная информация должна быть доступна из контекста.

Контекст события по PSR-3 это массив (можно вложенный) значений переменных, например, ID сущностей. Контекст можно оставить пустым, если по сообщению всё понятно о событии. В случае логирования исключения следует передавать всё исключение, а не только `getMessage()`.

Логирование. Способы логирования

Самый простой способ - логирование в файл. Плюсы данного подхода состоят в том, что нет необходимости устанавливать и настраивать какие-либо системы для работы с логами, а достаточно дать права на запись в лог-файл пользователю, от имени которого производится данное действие. Также к плюсам можно отнести надежность - нет необходимости заботиться о возможных отказах внешних систем логирования.

Недостатками такого способа является то, что со временем файл становится очень большим, что затрудняет поиск по нему, который к тому же не оптимизирован.

Логирование. Способы логирования

Другим способом логирования является использование специального ПО для сбора логов и работе (чтению) с ними.

В качестве примера можно привести стэк ELK (ElasticSearch + Logstash + Kibana), Prometheus или Loki + Grafana. Суть таких подходов состоит в наличии обработчика и опимизированного хранилища (ElasticSearch) логов, а также визуального интерфейса для просмотра, поиска и построения графиков (Kibana или Grafana).

Как правило, применяют комбинированный подход: в качестве основного инструмента выбирают специальный сервис, а в качестве резевного - логирование в файлы.