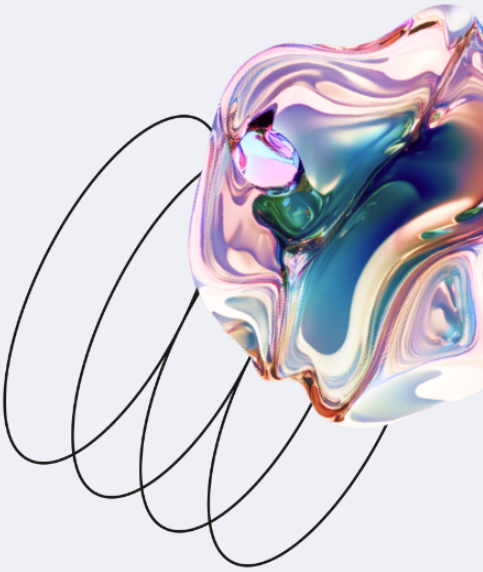
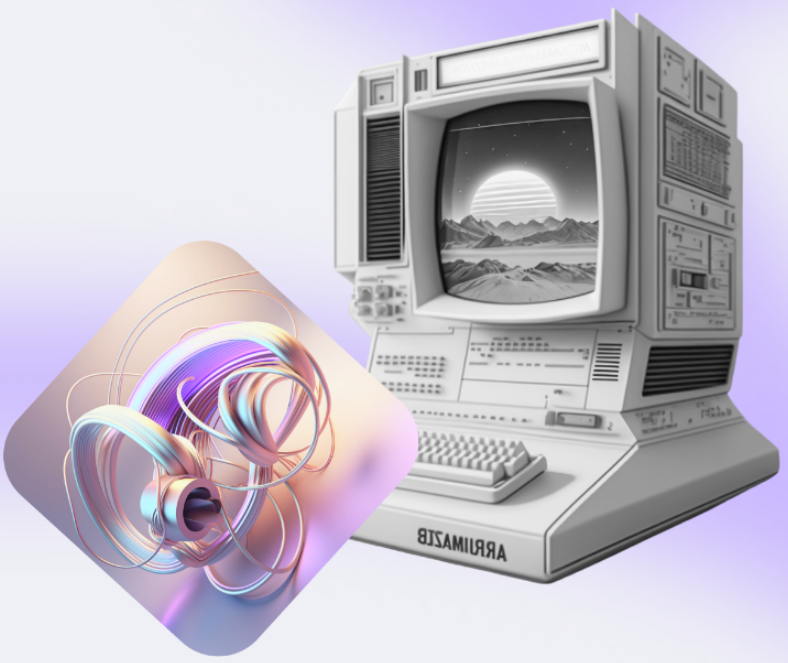
GeekBrains



Логирование, профилирование, дебаг

Основы PHP



Оглавление

Введение

За прошедшие несколько лекций наше приложение стало больше и сложнее. Мы добавляем к нему все больше функциональности, а значит становится сложнее его поддерживать и развивать.

Поэтому в рамках этой лекции мы:

- Научимся отслеживать работу нашего приложения в реальном времени
- Познакомимся с принципами отладки приложений
- Сформируем возможности для журналирования работы приложения

В процессе работы над домашними заданиями вы наверняка уже сталкивались с тем, что ваша программа ведет себя не так, как вы ожидаете. И для того, чтобы найти причины этого поведения, вам приходилось тратить много времени.

Поэтому анализ работы приложения – не менее важная часть разработки наравне с непосредственно созданием кода.

Логирование мы будем использовать для записи информации о работе приложения или системы. Оно позволяет сохранять события, ошибки, предупреждения и другую полезную информацию в журналах (логах), которые могут быть полезными при

анализе работы приложения, поиске и устранении ошибок, отслеживании действий пользователей и многое другое. Логирование помогает разработчикам понять, что происходит в системе во время работы и предоставляет ценную информацию для диагностики и отладки.

Профилирование же используется для измерения и анализа производительности приложения или системы. Оно помогает разработчикам идентифицировать узкие места, оптимизировать производительность и улучшить ресурсоемкость программы.

Профилирование позволяет получить информацию о времени выполнения отдельных участков кода, использовании памяти, вызовах функций и других аспектах работы приложения. Это полезный инструмент при оптимизации и настройке приложения для достижения максимальной производительности.

Логирование

Наиболее простой задачей, с которой мы начнем работу, является журналирование или логирование (от английского слова «log» – журнал).

Рассмотрим следующую ситуацию. Вы разрабатываете сложную часть вашего приложения. Вы написали много кода, но на выходе не выдаёт ожидаемый вами результат. Как же проверить, что происходит внутри?

Если вы самостоятельно разрабатываете своё приложение, то всегда можете применить, например, var_dump.

Но как понять, что происходило в момент пользовательского вызова? Ведь пользователь мог иметь специфические входные данные. И восстановить их без специальной подготовки не получится.

Одним из фундаментальных решений является анализ входных и выходных данных.

Он проводится следующим образом:

1. Выбираем функцию, которая вызывает выполнение нужной нам части кода
2. Журналируем входные данные (ведь они могут изменяться от вызова к вызову)
3. Журналируем выходные данные
4. Если результат отличается от наших ожиданий, то анализируем код функции

5. При необходимости выбираем вызываемые внутри функции и начинаем с пункта 2.

> 💡 Таким образом, анализируя получаемый результат, мы можем делать предположения о том, как же работает наша функция.

К примеру, если мы рассмотрим наш метод run в классе Application, то увидим, что в нем много конструкций типа if-else. Будет ли выполнен каждый из блоков внутри, зависит от того, какие данные пришли от пользователя?

Поэтому в случае, если пользователи жалуются на поведение, которого они не ожидали, либо вам нужно разобраться в том, почему отрабатывает тот или иной блок, можно и нужно применять логирование.

Итак, логирование в приложении имеет несколько важных целей:

1. Отладка и обнаружение ошибок: логирование позволяет записывать информацию о работе приложения, такую как сообщения об ошибках, предупреждения и исключения. Это помогает разработчикам быстро определить проблемные места в коде, выявить ошибки и неправильное поведение, а затем исправить их. Логи могут служить полезным инструментом для решения проблем в процессе разработки, тестирования и эксплуатации приложения.

2. Мониторинг и производительность: логирование позволяет отслеживать работу приложения в реальном времени и получать информацию о его производительности. Запись данных о времени выполнения определенных операций, использовании ресурсов и задержках помогает выявить узкие места и оптимизировать производительность приложения.

3. Аудит и безопасность: логирование может использоваться для регистрации действий пользователей и событий, происходящих в приложении. Это может быть полезно для аудита, отслеживания изменений в данных или выявления подозрительной активности. Логирование также может помочь в исследовании инцидентов безопасности и определении источников проблем.

4. Аналитика и улучшение продукта: логирование может предоставить ценную информацию об использовании приложения пользователями. Регистрация событий, взаимодействия и показателей использования позволяет проводить анализ и понимать, как пользователи взаимодействуют с приложением. Это может помочь в определении популярных функций, обнаружении проблем пользовательского опыта и принятии решений по улучшению продукта.

5. Поддержка и обслуживание: ооги могут быть полезными для команды поддержки, позволяя разработчикам или администраторам просматривать и анализировать проблемы, с которыми сталкиваются пользователи. Это облегчает процесс обнаружения и устранения проблем, а также предоставляет информацию для обратной связи и улучшения работы приложения.

> 💡 Логи в самом простом варианте могут записываться в текстовые файлы.

Но если система большая, то логи сохраняются в специализированные хранилища типа **Elastic Search**, которые позволяют более эффективно анализировать хранящуюся в них информацию.

> 🔥 Но мы начнем с работы именно с файловой системы – ведь если научиться сохранять логи с её применением, то дальше будет несложно заменить её на любое другое хранилище.

Сразу же стоит отметить, что хранение логов в базе данных – это **плохая идея**.

Дело в том, что база и без того хранит много данных, активно используя для этого оперативную память. А логов всегда будет гораздо больше, чем хранимых данных. Поэтому такое хранение будет сильно засорять базу.

> 🔥 Также обратите внимание на то, что логирование в файлы – это достаточно долгая операция, так как для её выполнения происходит обращение к жесткому диску.

А по скорости работы с данными серверное оборудование градируется от самого быстрого к самому медленному следующим образом:

1. Операции на уровне CPU
2. Операции на уровне RAM
3. Операции на уровне HDD

Поэтому, если вы хотите журналировать операции на production, где производительность приложения крайне важна, лучше всего иметь некий «выключатель», который позволяет включить сбор данных на короткое время, набрать достаточный для анализа объем и выключить сбор.

Реализация логирования

Конечно же, можно написать свой класс логирования, но эта задача является достаточно стандартной и здесь не стоит «изобретать велосипед». Поэтому мы будем подключать к нашему приложению библиотеку Monolog, которая предоставляется разработчиками фреймворка Symfony.

> 💡 Огромным плюсом тут также будет возможность в любой момент заменить файловое хранилище на какое-либо другое.

Monolog представляет собой мощный и гибкий инструмент для логирования информации о работе приложения, обладающий следующими особенностями:

1. Разнообразные обработчики (Handlers): Monolog поддерживает множество обработчиков, которые определяют, куда и каким образом будет записываться лог. Это может быть:
    a. запись в файл,
    b. отправка на электронную почту,
    c. сохранение в базе данных,
    d. интеграция со сторонними сервисами для анализа и мониторинга логов.
2. Уровни логирования (Log Levels): Monolog предоставляет несколько уровней логирования, таких как:
    a. DEBUG,
    b. INFO,
    c. WARNING,
    d. ERROR и другие.

    Это позволяет контролировать, какие сообщения будут записываться в лог в зависимости от уровня их значимости.
3. Каналы (Channels): Monolog поддерживает использование каналов, что позволяет организовать логирование по различным логическим категориям или компонентам приложения. Например, вы можете настроить отдельный

канал для логирования запросов к API и другой канал для логирования ошибок базы данных.

4. Форматирование (Formatting): Monolog предоставляет гибкую настройку формата записей в лог. Вы можете определить, какая информация будет включена в каждую запись, например, дату, уровень логирования, сообщение, контекст и другие метаданные.

5. Расширяемость: Monolog легко расширяется. Вы можете создавать собственные обработчики, форматтеры и прочие компоненты логирования, а также интегрировать Monolog с другими инструментами и сервисами.

Для того, чтобы подключить Monolog к нашему приложению, выполним команду:

```
docker run -it -v $(pwd)/code:/data/mysite.local/ myapp/php-cli composer require monolog/monolog
```

Она установит необходимые зависимости для применения библиотеки.

Теперь мы можем приступать непосредственно к программированию. Для начала определим в настройках приложения адреса хранения логов нашего приложения. Создадим в директории code новую поддиректорию log.

Сформируем конфигурацию наших логов:

```
[log]
LOGS_ON = "true"
LOGS_FILE = "debug.log"
```

Первая настройка – это **флаг**, который будет говорить нам, что логи можно включать. Это заготовка как раз на тот случай, когда мы захотим пособирать какую-то аналитическую информацию.

Второй параметр – это **имя файла лога**, в который мы будем собирать данные.

Сам модуль мы будем подключать в класс Application прямо в конструкторе ровно также, как мы делали это с конфигуратором:

```
public static Logger $logger;
public function __construct(){
        // остальной код конструктора

        Application::$logger = new Logger('application_logger');
        Application::$logger->pushHandler(
            new StreamHandler(
                                            $_SERVER['DOCUMENT_ROOT']   .   "/log/"
.Application::$config->get()['log']['LOGS_FILE'], Level::Debug)
            );
        Application::$logger->pushHandler(new FirePHPHandler());
    }
```

Не забудьте подключить нужные пространства имен:

```
use Monolog\Level;
use Monolog\Logger;
use Monolog\Handler\StreamHandler;
use Monolog\Handler\FirePHPHandler;
```

> 💡 Итак, в конструкторе мы создали экземпляр класса Logger – это основной элемент подключенной библиотеки. Каждый логгер нужно называть, так как в приложении может быть несколько логгеров.

Далее мы указываем, куда будем сохранять логи. Это делается через класс StreamHandler, указывая в параметрах его создания адрес нашего файла. И также мы указываем, что записывать будем через файловый обработчик.

Обратите здесь внимание на параметр уровня логов!

Мы указали уровень Debug. В целом запись в логи производится разными методами в зависимости от критичности. То есть, сообщение может быть информационным, а может рассказывать и о критической ошибке. Monolog поддерживает несколько уровней логирования.

Перечислим их по нарастанию приоритета:

- DEBUG: Детализированная информация, пишется всё.
- INFO: Интересующие аналитические события – например, SQL запросы
- NOTICE: Нормальные события, но имеющие важность (например, файл не был найден, но это не блокирует работу)

- WARNING: Исключительные события, но не ошибки (например, не найден элемент в массиве)
- ERROR: Ошибки в процессе исполнения программы
- CRITICAL: Критические сбои, неперехваченные исключения
- ALERT: Отказ части системы
- EMERGENCY: Полный отказ системы

Таким образом, если в системе установлен уровень логирования ERROR, то все попытки сохранить DEBUG события будут проигнорированы.

Мы установили в примере выше самый широкий уровень логирования, но в production-системе лучше применять ERROR и выше.

Теперь для примера давайте встроимся в код вызова методов в run:

```
else {
                $logMessage = "Метод " . $this->methodName . " не существует в
контроллере " . $this->controllerName . " | ";
                $logMessage .= "Попытка вызова адреса " .
$_SERVER['REQUEST_URI'];

                Application::$logger->error($logMessage);
                return "Метод не существует";
            }
```

Теперь, если мы вызовем несуществующий адрес, то в директории log появится нужный нам файл, где будет храниться сообщение:

```
[2023-07-06T15:32:24.969678+00:00] application_logger.ERROR: Метод actionAction
не существует в контроллере
Geekbrains\Application1\Domain\Controllers\UserController | Попытка вызова
адреса /user/action/ [] []
```

Эффективное использование логов может значительно облегчить отладку, мониторинг и анализ работы приложения.

- Определите цель и уровни важности:
    - Начните с определения целей логирования.
    - Разберитесь, какую информацию вы хотите получить из логов.
    - Определите уровни важности для различных видов сообщений (например, информационные, предупреждения, ошибки).

Это поможет вам классифицировать и фильтровать логи на основе их значимости.

- Используйте подходящий формат:
    - Выберите подходящий формат для ваших логов, чтобы обеспечить удобочитаемость и обработку данных.
    - Некоторые популярные форматы включают текстовые файлы, JSON или структурированные журналы.
    - Структурированные журналы позволяют легче анализировать данные с использованием инструментов обработки журналов или сценариев.
- Записывайте достаточно информации:
    - Убедитесь, что ваша система логирования записывает достаточно информации для понимания контекста и диагностики проблемы.
    - Включайте полезные метаданные, такие как временная метка, идентификатор запроса, пользовательский идентификатор и другие соответствующие данные.
- Используйте правильные уровни логирования:
    - Используйте различные уровни логирования для разных типов сообщений. Например, информационные сообщения могут использоваться для отслеживания хода выполнения программы, предупреждения – для потенциальных проблем, а ошибки – для фиксации критических ошибок.
    - Такая структура позволяет легче фильтровать и анализировать логи.
- Установите механизм ротации логов:
    - Лог-файлы могут быстро расти в размере, поэтому важно установить механизм ротации логов.
    - Ротация позволяет сохранять только последние файлы логов или ограничивать их размер, чтобы не заполнять диск.
    - Это поможет сохранить производительность системы и обеспечить доступность новых лог-файлов.

Как видите, теперь у нас есть возможность встраиваться в любое место приложения с тем, чтобы анализировать работу приложения. Но это лишь базовый метод отладки приложения. Давайте посмотрим, как можно развить этот подход.

Профилирование

> 💡 Профилирование – это процесс анализа и измерения производительности программы или системы с целью оптимизации ее работы.

Оно позволяет идентифицировать узкие места в коде, определить, сколько времени занимают определенные операции или функции, и выявить возможности для улучшения производительности.

В процессе профилирования используются различные инструменты и техники, такие как счетчики производительности, трассировка стека вызовов, сборщики мусора, анализаторы памяти и другие. Эти инструменты помогают разработчикам и инженерам понять, как программа работает в реальном времени и где возможны улучшения производительности для оптимизации работы системы.

Профилировать приложения можно без дополнительных расширений. Например, если мы хотим посмотреть, сколько памяти потребляет наше приложение, мы можем в точке входа (index.php). Замерить объём потреблённой оперативной памяти в байтах можно при помощи встроенной функции memory_get_usage(). Нам надо замерить память в начале и в конце, а затем посчитать разницу. Так мы поймём, сколько же приложение израсходовало ресурсов:

```
$memory_start = memory_get_usage();

require_once('./vendor/autoload.php');

use Geekbrains\Application1\Application\Application;

try{
    $app = new Application();
    $result = $app->run();
    echo $result;
}
catch(Exception $e){
    echo $e->getMessage();
}

$memory_end = memory_get_usage();

echo "<h3>Потреблено " . ($memory_end - $memory_start) . " байт памяти</h3>";
```

Такмы уже можем делать выводы о том, стало ли со временем наше приложение «тяжелее». Но для того, чтобы понять, что именно стало причиной изменения потребления ресурсов, этой информации будет недостаточно.

> 💡 Для этого можно использовать стандартный для PHP профилировщик – **Xdebug**.

Давайте установим его в нашу систему.

В Dockerfile допишем строчку:

```
RUN pecl install -o -f xdebug && docker-php-ext-enable xdebug
```

Также в файл конфигурации php.ini, который лежит рядом с Dockerfile, мы добавим ключи конфигурации xdebug:

```
[xdebug]
zend_extension=xdebug
xdebug.mode=develop,debug
xdebug.start_with_request=yes
xdebug.remote_port=9003
xdebug.client_port=9003
xdebug.remote_host=host.docker.internal
```

Они будут говорить нам о том, что расширение будет запущено в режиме отладки, работать будет внутри Docker-контейнеров, а подключаться к отладчику мы будем по порту 9003.

В docker-compose для контейнеров с PHP мы укажем следующие дополнительные параметры (можно указать их после секции network в каждом контейнере), чтобы соблюсти совместимость:

```
extra_hosts:
      - "host.docker.internal:host-gateway"
    environment:
      XDEBUG_MODE: develop,debug
      XDEBUG_CONFIG:
        client_host=host.docker.internal
        start_with_request=yes
```

Далее мы запустим наши контейнеры с ключом --build, который принудительно пересоберет образы:

```
docker-compose up -d --build
```

Но как нам проверить, что все установилось корректно?

Можно вызвать функцию xdebug_info. Поскольку мы используем шаблонизатор, то надо как-то передать её вывод в шаблон Twig, ведь просто так мы её не сможем вызвать.

Для проверки давайте напишем в классе Render код, который впоследствии удалим – он нам нужен только для проверки корректности сборки:

```
public function renderPage(string $contentTemplateName = 'page-index.tpl', array
$templateVariables = []) {
        $template = $this->environment->load('main.tpl');

        $templateVariables['content_template_name'] = $contentTemplateName;

        // временный код
        ob_start();
        \xdebug_info();
        $xdebug = ob_get_clean();

       $templateVariables['xdebug'] = $xdebug;
       // ------
        if(isset($_SESSION['user_name'])){
            $templateVariables['user_authorized'] = true;
        }

        return $template->render($templateVariables);
    }
```

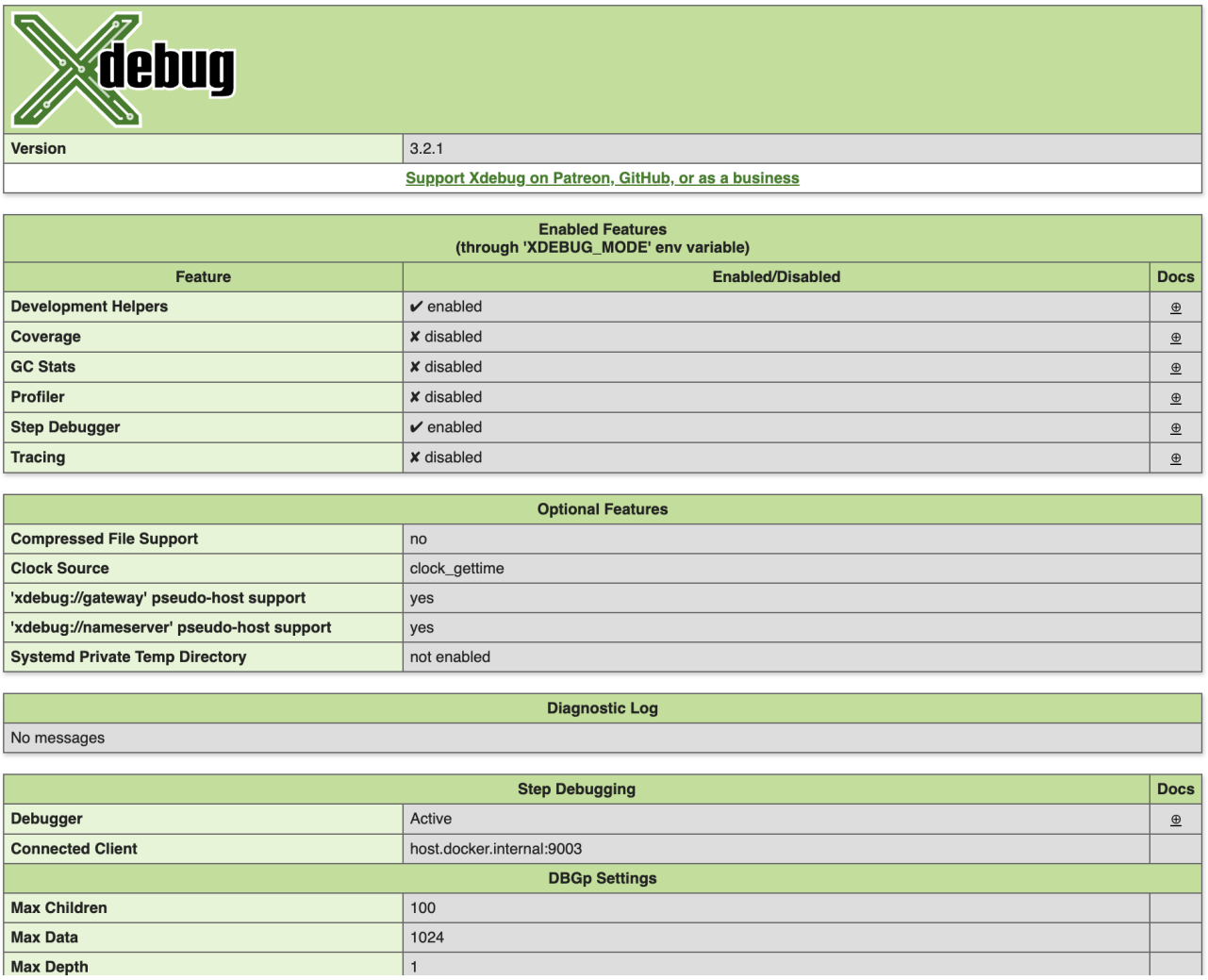
🔥 По сути, функция xdebug_info сразу же выводит результат на экран. Но нам этого не нужно. Поэтому при помощи output буферизации мы сохраним вывод в переменную и передадим его в шаблон.

Буферизация включается при помощи функции ob_start. Всё, что идёт после неё и пытается вывести что-то на экран, будет сохранено в буфер – это некая область перехвата данных, которая не пускает их дальше. При помощи ob_get_clean мы выгружаем всё, что накопилось в буфере, и чистим его.

Таким образом, в шаблон будет передана переменная xdebug. В шаблоне же mail.tpl мы выведем её привычным образом:

```
{{ xdebug | raw }}
```

Если вы всё сделали правильно, то вы увидите информацию о модуле XDebug:



Version	3.2.1
Support Xdebug on Patreon, GitHub, or as a business	

Enabled Features (through 'XDEBUG_MODE' env variable)		
Feature	**Enabled/Disabled**	**Docs**
Development Helpers	✔ enabled	⊕
Coverage	✘ disabled	⊕
GC Stats	✘ disabled	⊕
Profiler	✘ disabled	⊕
Step Debugger	✔ enabled	⊕
Tracing	✘ disabled	⊕

Optional Features	
Compressed File Support	no
Clock Source	clock_gettime
'xdebug://gateway' pseudo-host support	yes
'xdebug://nameserver' pseudo-host support	yes
Systemd Private Temp Directory	not enabled

Diagnostic Log
No messages

Step Debugging		Docs
Debugger	Active	⊕
Connected Client	host.docker.internal:9003	
DBGp Settings		
Max Children	100	
Max Data	1024	
Max Depth	1	

Точки отладки

Но пока это только начало.

В программировании есть мощный инструмент отладки, который позволяет программисту временно приостановить выполнение программы в определенной точке, чтобы исследовать состояние программы и проверить значения переменных.

> 🔥 Этот инструмент – breakpoint (точка останова).

При установке breakpoint'a в коде PHP интерпретатор останавливает выполнение программы в указанной точке. Это дает возможность программисту пошагово исследовать код, выполнять его пошагово, отслеживать значения переменных, проверять условия и т. д. Во время остановки на breakpoint'e можно проводить отладочные операции, такие как просмотр стека вызовов, изменение значений переменных, анализ выполнения условных операторов и циклов.

Существует несколько способов установки breakpoint'ов в PHP:

- Использование отладчика (debugger): отладчики, такие как Xdebug, предоставляют возможность установить breakpoint'ы в коде и контролировать выполнение программы с помощью специальных команд. Они обычно интегрируются с различными IDE (интегрированными средами разработки) и предоставляют расширенные возможности для отладки.
- Использование функции xdebug_break(): функция xdebug_break() является частью расширения Xdebug и позволяет установить breakpoint непосредственно в коде PHP. При выполнении этой функции интерпретатор останавливается на указанном месте.
- Вручную добавление инструкции die() или exit(): если вы хотите быстро установить временный breakpoint в определенной части кода, вы можете добавить инструкцию die() или exit() в соответствующем месте. Это остановит выполнение программы и позволит вам изучить состояние программы.

> 🔥 Важно помнить, что использование breakpoint'ов и отладки является инструментом для разработки и отладки программы, и в рабочей среде они обычно не должны быть оставлены в коде.

Итак, нам нужно научить нашу среду разработки работать с точками останова.

VS Code предоставляет ряд расширений, которые помогут в работе. Нам потребуются:

- PHP Intelephense
- PHP Debug
- PHP Profiler

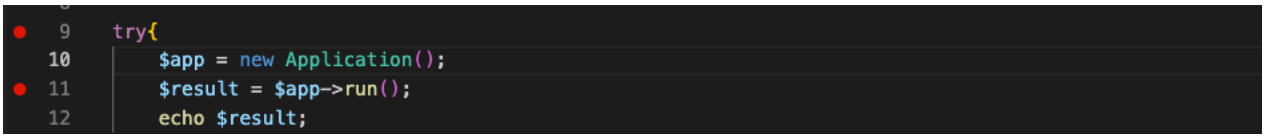
Установите их в своей IDE. Для других IDE настройка, разумеется, будет отличаться.

После установки в меню «Выполнить», затем «Открыть конфигурации». У Вас на экране появится содержимое файла launch.json. Отредактируем его следующим образом:

```
{
    // Используйте IntelliSense, чтобы узнать о возможных атрибутах.
        // Наведите указатель мыши, чтобы просмотреть описания существующих
атрибутов.
                // Для получения дополнительной информации посетите:
https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Listen for Xdebug",
            "type": "php",
            "request": "launch",
            "port": 9003,
            "pathMappings": {
                "/data/mysite.local": "${workspaceFolder}/code/"
            }
        },
        {
            "name": "Listen on Docker for Xdebug",
            "type": "php",
            "request": "launch",
            "port": 9003,
            "pathMappings": {
                "/data/mysite.local": "${workspaceFolder}/code/"
            }
        },
    ]
}
```

Обратите внимание на параметр pathMappings, который ставит в соответствие корневую директорию нашего проекта в контейнере и локальную среду.

Теперь у нас становится доступной возможность простановки точек останова:



Рядом с нужной строкой поставьте красную точку.

Давайте запустим режим дебага, выбрав конфигурацию Listen on Docker for Xdebug. Вызовем любую страницу в браузере. Интерпретатор остановится на первой встретившейся точке останова.

В левой части IDE вам станут доступны данные о том, какие есть переменные в данном вызове, суперглобальные массивы и константы.

> 💡 Также ниже вам будет доступен стек вызовов (или call stack) – это структура данных, используемая в компьютерных программах для отслеживания последовательности вызовов функций или методов во время выполнения программы.

Когда программа вызывает функцию или метод, информация о вызове и переданных аргументах сохраняется в стеке вызовов. Каждый вызов функции или метода добавляется в верхнюю часть стека, образуя "стековый кадр" или "фрейм вызова".

Этот фрейм содержит информацию, такую как адрес возврата – место, куда вернется выполнение программы после завершения вызываемой функции, а также локальные переменные и другие связанные данные.

При выполнении функции или метода, если внутри него происходит вызов другой функции или метода, новый фрейм вызова добавляется на вершину стека, и процесс продолжается. Когда вызываемая функция или метод завершает свою работу, фрейм вызова удаляется из стека, и управление передается обратно в вызывающую функцию или метод.

> 🔥 Стек вызовов работает по принципу "последним пришел – первым вышел" (Last-In-First-Out, LIFO), что означает, что последний добавленный фрейм вызова будет первым удаленным при завершении вызовов.

В меню профилировщика мы можем переходить между точками останова и опускаться или подниматься по стеку вызовов, чтобы видеть, что будет происходить на каждом шаге.

Такой механизм даст вам предельно подробное понимание того, что именно происходит с данными в вашей программе. А значит, вы сможете корректировать её поведение, чтобы получать ожидаемый результат.

Крайне важно понимать, что такой механизм как XDebug, а тем более – точки остановки, не должны размещаться на production серверах. Поэтому лучше всего убрать из php.ini настройку:

```
xdebug.mode
```

Перенесем ее в docker-compose:

```
environment:
      XDEBUG_MODE: develop,debug
      XDEBUG_CONFIG:
        client_host=host.docker.internal
        start_with_request=yes
        xdebug.mode= ${XDEBUG_MODE}
```

Как вы наверняка уже догадались, мы будем размещать флаг режима работы xdebug в файле .env. В production окружении он должен быть установлен в значение off.

Процесс дебага

Мы уже изучили возможности логирования и профилирования, но нам надо поговорить об аспектах практического применения изученных процессов.

> 🔥 Процесс отлова и исправления проблем (иными словами – дебаг) в приложениях является неотъемлемым процессом разработки.

Редкий программист может написать идеально работающий код с первого раза. Поэтому нужно уметь отыскивать проблемы быстро и эффективно.

На ранних этапах карьеры поиск причины ошибки часто сводится в лучшем случае к поиску по последней фразе ошибки в сети Интернет. А чаще – просто к слепой попытке угадать причину. Однако это совсем не инженерный подход к задаче.

Самый очевидный случай, который побуждает нас к отладке приложения – это текст ошибки на экране. Давайте постараемся воспроизвести процесс дебага так, как он проходит в реальной работе программиста.

Возьмём адрес http://mysite.local/user/hash/

Если вызвать его без GET-параметров, то в текущем варианте логики мы увидим ответ примерно следующего содержания:

Warning: Undefined array key "pass_string" in /data/mysite.local/src/Domain/Controllers/UserController.php on line *81*

Call Stack

#	Time	Memory	Function	Location
1	0.0029	398616	{main}( )	.../index.php:0
2	0.0541	821344	Geekbrains\Application1\Application\Application->run( )	.../index.php:11
3	0.0592	841880	call_user_func_array:{/data/mysite.local/src/Application/Application.php:73}($callback = [0 => class Geekbrains\Application1\Domain\Controllers\UserController { protected array $actionsPermissions = [...] }, 1 => 'actionHash'], $args = [])	.../Application.php:73
4	0.0592	841952	Geekbrains\Application1\Domain\Controllers\UserController->actionHash( )	.../Application.php:73

Fatal error: Uncaught TypeError: Geekbrains\Application1\Application\Auth::getPasswordHash(): Argument #1 ($rawPassword) must be of type string, null given, called in /data/mysite.local/src/Domain/Controllers/UserController.php on line 81 and defined in /data/mysite.local/src/Application/Auth.php on line *6*

TypeError: Geekbrains\Application1\Application\Auth::getPasswordHash(): Argument #1 ($rawPassword) must be of type string, null given, called in /data/mysite.local/src/Domain/Controllers/UserController.php on line 81 in /data/mysite.local/src/Application/Auth.php on line *6*

Call Stack

#	Time	Memory	Function	Location
1	0.0029	398616	{main}( )	.../index.php:0
2	0.0541	821344	Geekbrains\Application1\Application\Application->run( )	.../index.php:11
3	0.0592	841880	call_user_func_array:{/data/mysite.local/src/Application/Application.php:73}($callback = [0 => class Geekbrains\Application1\Domain\Controllers\UserController { protected array $actionsPermissions = [...] }, 1 => 'actionHash'], $args = [])	.../Application.php:73
4	0.0592	841952	Geekbrains\Application1\Domain\Controllers\UserController->actionHash( )	.../Application.php:73
5	0.0593	842504	Geekbrains\Application1\Application\Auth::getPasswordHash($rawPassword = NULL)	.../UserController.php:81

Итак, что же мы видим здесь? Это три ошибки.

Кстати, благодаря наличию XDebug, мы получаем хорошо читаемое сообщение с порядком вызова – так называемым **трейсом**.

Самым простым способом (но не факт, что рабочим) будет взять текст ошибки, открыть Google или StackOverflow и вбить в поиск этот текст. Но этот метод не гарантирует ни ответа, ни понимания причин.

Так что мы будем разбираться с тем, как читать такие ответы. Разумеется, здесь понадобится минимальное знание английского языка или хотя бы онлайн переводчик.

Начнем с расшифровки сообщений. В оранжевых блоках заголовков мы видим:

1. Предупреждение – обращение к отсутствующему ключу в файле UserController на строке 81.
2. Фатальная ошибка – аргумент в методе getPasswordHash должен иметь тип «строка», но передано значение null.
3. Ошибка типа – она говорит то же самое.

Теперь посмотрим на трейс вызова – это серые блоки call stack.

Мы видим, что в первом случае вызывается главный поток (по сути – вызов index.php), далее метод run в классе Application. Затем происходит вызов функции через call_user_func_array – это наша логика **роутинга**.

Поэтому этого в UserController вызывается actionHash, где и происходит ошибка.

Второй трейс говорит примерно то же, но на последнем шаге еще происходит вызов генерации хэша пароля.

> 🔥 Начнем с фатальной ошибки – это самый серьёзный тип, при наличии которого работа скрипта будет совершенно невозможна.

Можно поставить точку остановки в коде на строке 81 в UserController, но текст ошибки уже говорит нам, что туда приходит NULL.

Давайте посмотрим на код.

```
public function actionHash(): string {
        return Auth::getPasswordHash($_GET['pass_string']);
}
```

На 81 строке мы уже понимаем, что проблема в отсутствии параметра pass_string. Метод getPasswordHash у нас требует строго строку. Поэтому давайте исправим здесь работу нашей логики.

Очевидно, что отсутствие параметра должно быть обработано:

```
public function actionHash(): string {
        if(isset($_GET['pass_string']) && !empty($_GET['pass_string'])){
            return Auth::getPasswordHash($_GET['pass_string']);
        }
        else {
                throw new Exception("Невозможно сгенерировать хэш. Не передан
пароль");
        }
    }
```

Мы проверяем, задан ли параметр и не пуст ли он. В противном случае мы будем выбрасывать исключение о том, что хэш не может быть сгенерирован.

Разберем еще один пример. При создании учетной записи admin (поскольку мы делали это вручную) можно забыть указать временную метку дня рождения:

id_user	user_name	user_lastname	user_birthday_timestamp	login	
1	Admin	Admin	null	admin	$2y$10$z
2	Иван	Петров	794102400		null
3	Петр	Иванов	602380800		null
4	awd	awd	989020800		null
5		123	696988800		null
6		123	696988800		null
7		123	696988800		null
8	Test	Test	791683200		null
9	Test	Test	791683200		null

Попробуем теперь вызвать метод вывода всех пользователей на экран.

http://mysite.local/user/index/

В ответ мы получим ошибку уровня Fatal. Зная об отсутствии даты рождения, можно смело сказать: «ОК, мы сейчас её заполним, и проблема уйдёт». Но так мы только уберём симптомы, но не вылечим проблему.

Поэтому давайте приступим к анализу:

Fatal error: Uncaught TypeError: Geekbrains\Application1\Domain\Models\User::getUserBirthday(): Return value must be of type int, null returned in /data/mysite.local/src/Domain/Models/User.php on line 40

TypeError: Geekbrains\Application1\Domain\Models\User::getUserBirthday(): Return value must be of type int, null returned in /data/mysite.local/src/Domain/Models/User.php on line 40

Call Stack

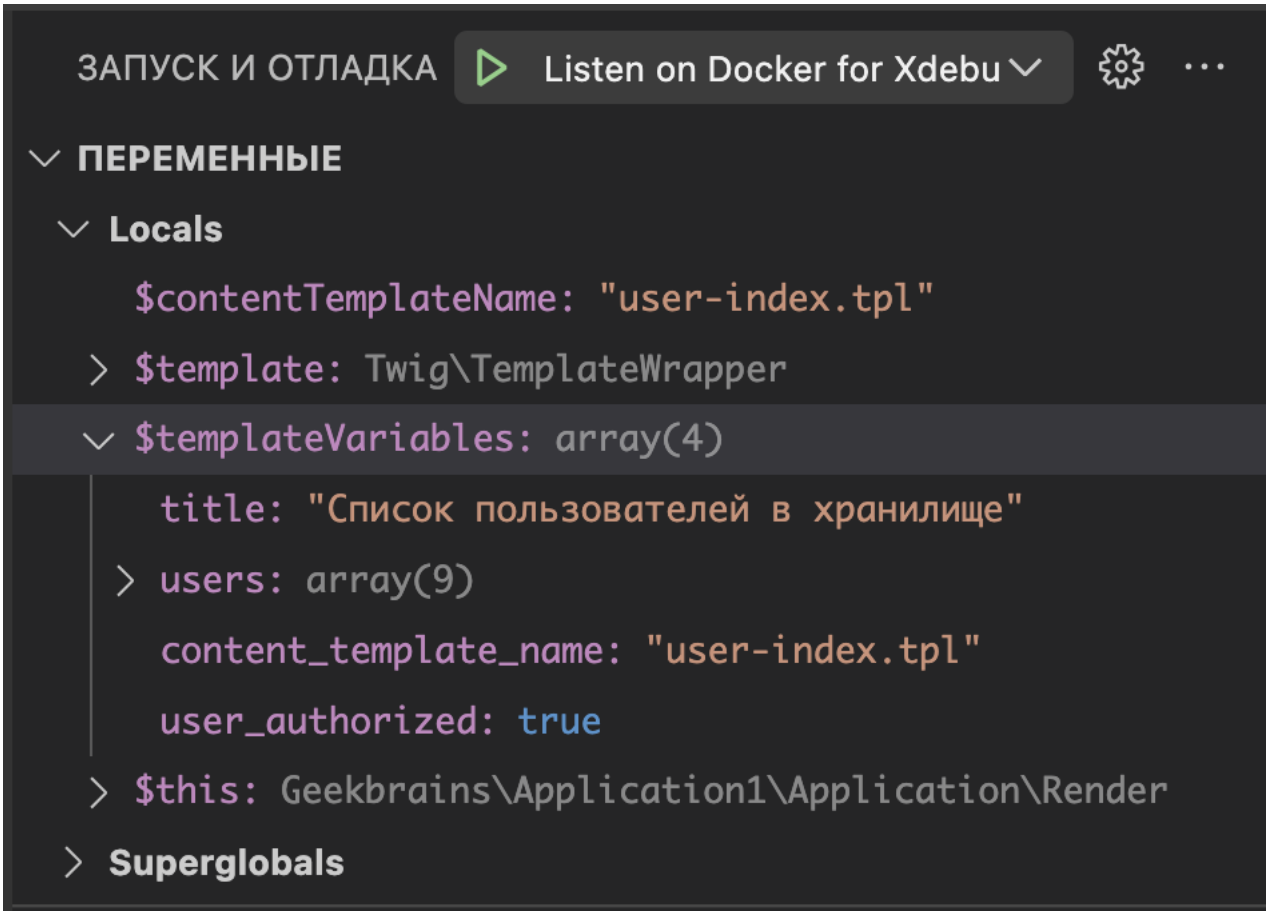
#	Time	Memory	Function	Location
1	0.0028	398488	{main}( )	.../index.php:0
2	0.0545	821216	Geekbrains\Application1\Application\Application->run( )	.../index.php:11
3	0.0609	842656	call_user_func_array:{/data/mysite.local/src/Application/Application.php:73}($callback = [0 => class Geekbrains\Application1\Domain\Controllers\UserController { protected array $actionsPermissions = [...] }, 1 => 'actionIndex'], $args = [])	.../Application.php:73
4	0.0609	842728	Geekbrains\Application1\Domain\Controllers\UserController->actionIndex( )	.../Application.php:73
5	0.0889	1451568	Geekbrains\Application1\Application\Render->renderPage($contentTemplateName = 'user-index.tpl', $templateVariables = ['title' => 'Список пользователей в хранилище', 'users' => [0 => class Geekbrains\Application1\Domain\Models\User { ... }, 1 => class Geekbrains\Application1\Domain\Models\User { ... }, 2 => class Geekbrains\Application1\Domain\Models\User { ... }, 3 => class Geekbrains\Application1\Domain\Models\User { ... }, 4 => class Geekbrains\Application1\Domain\Models\User { ... }, 5 => class Geekbrains\Application1\Domain\Models\User { ... }, 6 => class Geekbrains\Application1\Domain\Models\User { ... }, 7 => class Geekbrains\Application1\Domain\Models\User { ... }, 8 => class Geekbrains\Application1\Domain\Models\User { ... }]])	.../UserController.php:32
6	0.0952	1517496	Twig\TemplateWrapper->render($context = ['title' => 'Список пользователей в хранилище', 'users' => [0 => class Geekbrains\Application1\Domain\Models\User { ... }, 1 => class Geekbrains\Application1\Domain\Models\User { ... }, 2 => class Geekbrains\Application1\Domain\Models\User { ... }, 3 => class Geekbrains\Application1\Domain\Models\User { ... }, 4 => class Geekbrains\Application1\Domain\Models\User { ... }, 5 => class Geekbrains\Application1\Domain\Models\User { ... }, 6 => class Geekbrains\Application1\Domain\Models\User { ... }, 7 => class Geekbrains\Application1\Domain\Models\User { ... }, 8 => class Geekbrains\Application1\Domain\Models\User { ... }], 'content_template_name' => 'user-index.tpl', 'user_authorized' => TRUE])	.../Render.php:37
7	0.0952	1517496	Twig\Template->render($context = ['title' => 'Список пользователей в хранилище', 'users' => [0 => class Geekbrains\Application1\Domain\Models\User { ... }, 1 => class Geekbrains\Application1\Domain\Models\User { ... }, 2 => class Geekbrains\Application1\Domain\Models\User { ... }, 3 => class Geekbrains\Application1\Domain\Models\User { ... }, 4 => class Geekbrains\Application1\Domain\Models\User { ... }, 5 => class Geekbrains\Application1\Domain\Models\User { ... }, 6 => class Geekbrains\Application1\Domain\Models\User { ... }, 7 => class Geekbrains\Application1\Domain\Models\User { ... }, 8 => class Geekbrains\Application1\Domain\Models\User { ... }], 'content_template_name' => 'user-index.tpl', 'user_authorized' => TRUE], [])	.../TemplateWrapper.php:40

Что же происходит здесь?

Ошибка говорит нам о том, что метод getUserBirthday должен возвращать тип integer, но возвращает null. Трейс при этом доходит до генерации шаблона. Поэтому давайте начнем с него.

Начать стоит с того, какой же шаблон у нас будет вызываться. Мы знаем, что шаблоны вызываются в методе renderPage класса Render.

Создадим точку остановки прямо в return нашего метода. В момент повторного вызова страницы мы увидим содержимое массива переменных шаблона:



Теперь мы видим, что вызывается шаблон user-index.tpl. Посмотрим на него.

В нем как раз есть вызов функции, которая выдает ошибку!

```
<li>{{ user.getUserName() }} {{ user.getUserLastName() }}. День рождения: {{ user.getUserBirthday() | date('d.m.Y') }}</li>
```

То есть, при попытке вызова этой функции, она пытается вернуть null. Посмотрим на определение этой функции:

```
public function getUserBirthday(): int {
        return $this->userBirthday;
    }
```

Это обычный getter, который даёт доступ к приватному свойству класса. Но каким же образом можно его исправить?

Конечно, можно задать некий день рождения по умолчанию. Но это будет иметь не очень хорошие последствия для пользователей. В случае рассылки поздравлений будет много ложных срабатываний. Значит, для нас null является нормальным значением свойства.

Более того, в самом определении свойства также есть этот факт:

```
private ?int $userBirthday;
```

Что же – давайте поправим возвращаемый тип у метода:

```
public function getUserBirthday(): ?int {
        return $this->userBirthday;
    }
```

Теперь при обновлении страницы фатальной ошибки не будет. Но появляется другая проблема.

Мы применяем фильтр в Twig, который преобразует timestamp в дату. А поскольку дата пустая, Twig будет выводить просто сегодняшнее число. Нам это тоже не подходит, так как не отображает реального состояния данных.

Нам нужно доработать шаблон таким образом, что при получении значения null мы будем выводить информацию о том, что день рождения не установлен.

И здесь нам поможет встроенное в Twig ветвление:

```
<p>Список пользователей в хранилище</p>

<ul id="navigation">
    {% for user in users %}
        <li>
        {{ user.getUserName() }}
        {{ user.getUserLastName() }}.
        День рождения:
            {% if user.getUserBirthday() is empty %}
                <b>не установлен</b>
            {% else %}
                {{ user.getUserBirthday() | date('d.m.Y') }}
            {% endif %}
         </li>
    {% endfor %}
</ul>
```

Не забудем обновить кэш шаблонов. И теперь при отсутствии дня рождения в хранилище мы будем выводить корректную информацию.

Заключение

Теперь мы умеем не только дорабатывать наше приложение, но еще и заниматься его отладкой. Полученные знания дают нам возможность анализировать производительность приложения, его состояние, а также быстро находить ошибки.

Домашнее задание

В уже созданных маршрутах попробуйте вызывать их с некорректными данными. Что будет происходить? Будут ли появляться ошибки?

При появлении ошибок, произведите их анализ. Обязательно зафиксируйте шаги своих размышлений.

На основании анализа произведите устранение.

Что можно почитать еще?

1. https://phptherightway.com/
2. https://php.net
3. PHP 8 | Котеров Дмитрий Владимирович
4. Объектно-ориентированное мышление | Мэтт Вайсфельд

5. https://stackoverflow.com/questions/34034730/how-to-enable-color-for-php-cli - как покрасить вывод консоли

6. https://phptoday.ru/post/gotovim-lokalnuyu-sredu-docker-dlya-razrabotki-na-php - варианты образа Composer

7. https://anton-pribora.ru/articles/php/mvc/