

Файлы, подключение кода, Composer

ОСНОВЫ PHP



Оглавление

Введение	3
Словарь терминов	4
Хранение данных в файлах	5
Чтение из файлов	5
Запись в файлы	9
Форматы файлов	11
Файлы с кодом	13
Точка входа	14
Подключение файлов с кодом	15
Автоматизация подключения файлов	23
Настройка приложения	26
Работа с файловой системой	28
Заключение	33
Домашнее задание	33
Что можно почитать еще?	34

Введение

Как вы помните, PHP – это интерпретируемый язык программирования, который запускается на сервере. Очень важным аспектом является то, что в большинстве случаев скрипт на PHP в конце всегда «умирает».

Это означает, что скрипт работает по следующему алгоритму:

1. Запуск интерпретатора и загрузка модулей.
2. Инициализация переменных.
3. Запуск функций и выполнение логики.
4. Завершение работы и уничтожение хранимых в памяти данных.

Таким образом, между вызовами, например, разных страниц в браузере сохранение каких-либо данных является задачей, которую нам только предстоит решить. Например, если мы хотим запомнить данные пользователя (имя, дата рождения) между вызовами, чтобы раз в году поздравлять его с днем рождения, нам придется разместить их где-то на сервере в долгосрочном хранилище (на жестком диске).

Самым простым решением для сохранения состояния между запусками скрипта является сохранение данных в файле на сервере. Это может быть простой текстовый файл или файл, хранящий данные в каком-то структурированном формате (например, CSV – comma separated values). В таком файле данные хранятся построчно, где каждая строка – это кортеж данных. Кортежем в программировании называют набор элементов разных типов, принадлежащих к одной сущности.

Например, так:

```
Иван Иванов; 01-02-1990  
Петр Петров; 02-03-1991
```

Что будет на уроке сегодня:

- Долгосрочное хранение данных
- Подключение файлов с кодом
- Автозагрузка
- Конфигурация
- Первое приложение

Словарь терминов

Точка входа в приложение PHP – это файл, который является первым файлом, который запускается при обработке запросов на сервере.

Формат JSON (JavaScript Object Notation) – это формат обмена данными, основанный на синтаксисе объектов JavaScript. Он используется для представления данных в виде текста, который может быть легко считан и понят другими программами.

Рефакторинг кода – это процесс изменения внутренней структуры программного кода, без изменения его внешнего поведения, с целью улучшения его качества, читаемости, понимаемости, поддерживаемости и расширяемости.

Абсолютный путь – это полный путь к файлу или директории от корневого каталога.

Относительный путь – это путь к файлу или директории относительно текущей рабочей директории.

Суперглобальные массивы в PHP – это специальные массивы, которые доступны в любом месте скрипта и содержат различную информацию, связанную с сервером, запросом пользователя и другими аспектами выполнения скрипта.

Composer – это менеджер зависимостей для PHP, который позволяет управлять зависимостями PHP-приложения. Он упрощает процесс установки и обновления библиотек и фреймворков, используемых в проекте.

CSV (comma separated values) – файл, в котором данные хранятся построчно, где каждая строка – это кортеж данных.

Кортеж – набор элементов разных типов, принадлежащих к одной сущности.

file_get_contents() – функция, которая позволяет прочитать содержимое файла в виде строки.

fread() – функция, которая используется для чтения данных из открытого файла в бинарном режиме.

fopen() – функция, которая открывает файл, сохраняя результат открытия в переменную с типом Resource.

fclose() – функция, которая закрывает файл.

Resource – тип данных, который используется для обозначения внешних источников данных, таких как файлы, базы данных, соединения с серверами и т.д.

file_exists () – функция, которая проверяет, существует ли искомый нами файл.

is_readable () – функция, которая поможет убедиться, что PHP сможет прочесть файл.

file_put_contents() – функция, которая записывает данные в файл, создавая его при необходимости.

fwrite () – функция, которая записывает данные в открытый файл.

is_writable() – функция, которая проверять права на запись в файл.

Функции include и include_once – они работают так же, как require и require_once, за исключением того, что, если файл не найден, то будет выдано предупреждение, но выполнение скрипта не будет прервано.

Функции require и require_once – функции, которые загружают указанный файл и вставляют его содержимое в текущий файл, как бы собирая его наподобие конструктора. Require_once загружает файл только один раз, а require может загрузить файл несколько раз.

Функция scandir – это функция, которая используется для сканирования содержимого директории и возврата массива элементов, представляющих содержимое этой директории.

Хранение данных в файлах

Чтение из файлов

Начнем с более простой задачи – чтения данных из файла

PHP предоставляет несколько способов чтения файла. Самый простой и наиболее распространённый способ – использование функции **file_get_contents()**. Она позволяет прочитать содержимое файла в виде строки.

Пример:

```
$fileContents = file_get_contents('file.txt');  
echo $fileContents;
```

Предположим, что у нас есть файл, в котором хранятся данные о студентах:

Если мы применим чтение из примера выше, то мы получим вывод ровно в том же формате, который будет лежать в файле.

🔥 Но здесь есть проблема! Она состоит в том, что файл может оказаться очень большим. При чтении такого файла всё его содержимое PHP будет вынужден загрузить в оперативную память системы.

Это очень дорогая операция, да и памяти может банально не хватить. Поэтому в случае, когда размер файла неизвестен заранее, лучше использовать более низкоуровневые функции чтения.

Основная идея здесь будет состоять в том, что в память выгружается небольшой кусок данных, обрабатывается, затем на его место загружается следующий кусок.

Это функции:

- `fopen()`,
- `fread()`,
- `fclose()`.

💡 Функция `fopen()` открывает файл, сохраняя результат открытия в переменную с типом `Resource`.

Тип «`Resource`» в PHP представляет собой специальный тип данных, который используется для обозначения внешних источников данных, таких как файлы, базы данных, соединения с серверами и т.д. Этот тип данных не является примитивным типом в PHP, он хранит ссылку на внешний ресурс, который управляется внутри расширения PHP.

Когда вы создаете соединение с файлом, PHP возвращает ссылку на этот ресурс в виде типа «`Resource`». Это означает, что этот ресурс не является простым значением, которое можно скопировать или присвоить другой переменной, но является внешним ресурсом, который управляется внутри расширения PHP.

Когда вы закончили работу с ресурсом, вы должны освободить его, чтобы предотвратить утечку памяти и другие проблемы (например, превышение лимита соединений с внешним источником данных). Обычно для этого используется

функция расширения PHP, например, **fclose()** для закрытия файла или **mysqli_close()** для закрытия соединения с базой данных.

Далее надо прочитать данные из файла, ссылку на который мы сохранили. Это делает функция **fread()** – она используется для чтения данных из открытого файла в бинарном режиме. Эта функция имеет следующий синтаксис:

```
fread ( resource $handle , int $length ) : string
```

Где:

- `$handle` – это открытый файловый указатель, возвращаемый функцией `fopen()`.
- `$length` – это количество байтов, которое нужно прочитать из файла.


Функция **fread()** читает количество байтов, указанное в `$length`, из файла, на который указывает `$handle` – это как раз тот самый Resource, который мы получили при открытии через `fopen`. В итоге функция возвращает полученные данные в виде строки. Если до конца файла осталось меньше `$length` байтов, то функция прочитает только оставшуюся часть файла.

Например, чтобы прочитать первые 100 байтов из файла "example.txt", вы можете использовать следующий код:

```
$file = fopen("example.txt", "rb");  
$data = fread($file, 100);  
fclose($file);
```

В этом примере мы открываем файл "example.txt" в бинарном режиме (режим "rb"), читаем первые 100 байтов с помощью функции `fread()`, сохраняем их в переменную `$data`, а затем закрываем файл с помощью функции `fclose()`.

Режимов для работы у `fopen` может быть несколько. Например, файл может быть открыт на запись, что мы рассмотрим чуть позже.

 Обратите внимание, что функция `fread()` в данном примере читает данные из файла в бинарном режиме, то есть она не обрабатывает переносы строк или другие символы в специальном режиме.

Функция `fclose()` закрывает файл. Это нужно обязательно делать, так как при незакрытии файла, он может оказаться недоступным для чтения другими пользователями.

🔥 Но 100 байтов нам может не хватить – файл может оказаться больше!

Поэтому мы должны читать файл до тех пор, пока не дойдем до его конца.

И здесь нам пригодится уже знакомый цикл **while**.

Модифицируем наш код:

```
<?php
$file = fopen("/code/example.txt", "rb");
if ($file === false) {
    echo("Файл невозможно открыть или он не существует");
}
else {
    $contents = '';
    while (!feof($file)) {
        $contents .= fread($file, 100);
    }
    fclose($file);
    echo $contents;
}
```

🔥 Обратите внимание на то, что здесь мы начинаем применять обработку нестандартных ситуаций в работе скриптов. Это нужно для того, чтобы, в случае ошибок, наши пользователи могли получать более информативные сообщения.

В нашем случае мы обработаем ситуацию, когда файл не существует (например, адрес неверный).

В таком случае наша программа выдаст **Warning** о том, что файл не удалось открыть, так как он не существует. А дальше на экране появится наша ошибка. Выглядеть это будет так:

```
Warning: fopen(/code/example1.txt): Failed to open stream: No such file or
directory in /code/fopen-simple.php on line 4
Файл невозможно открыть или он не существует
```

Но Warning для пользователя может содержать излишнюю информацию.

Поэтому мы можем оставить только наше сообщение. В самом простом варианте нам нужно проверять, существует ли вообще искомый нами файл. Это можно сделать при помощи **file_exists**. Также нам надо убедиться, что PHP сможет прочесть файл – за это отвечает функция **is_readable**.

Теперь мы можем модифицировать наш код так, чтобы он выводил только желаемое нами сообщение об ошибке:

```
$address = "/code/example1.txt";

if (file_exists($address) && is_readable($address)) {
    $file = fopen($address, "rb");

    $contents = '';

    while (!feof($file)) {
        $contents .= fread($file, 100);
    }

    fclose($file);
    echo $contents;
}
else {
    echo("Файл не существует или недоступен для чтения");
}
```

Запись в файлы

Теперь нам надо научиться писать в файл для того, чтобы пополнять наши данные.

Наравне с `file_get_contents` есть функция-обертка **`file_put_contents()`**. Она записывает данные в файл, создавая его при необходимости:

```
$file = 'people.txt';
file_put_contents($file, 'Иван Иванов');
```

Но, как вы помните, размер записываемых данных не всегда известен заранее. Поэтому можно использовать аналог `fread` – функцию **`fwrite`**. Она записывает данные в открытый файл. И здесь важно обращать внимание на несколько аспектов.

Мы уже умеем проверять существование файла и его читаемость. Наравне с этим есть возможность проверять права на запись в файл. Это делает функция **`is_writable`**.

Помимо этого, есть разные режимы записи в файл. Читали мы из файла, открыв его в режиме «**r**» – чтение. Для записи же нам нужно будет выбрать один из режимов дополнения информации:

- 'w' – открыть только для записи, начать запись с начала файла и удалить его содержимое. Если файл не существует, попытаться его создать.
- 'w+' – аналогично w, но из файла можно будет и читать.

- 'a' – открыть только для записи, дописывать данные в конец файла, не удаляя его содержимое. Если файл не существует, попытаться его создать.
- 'a+' – аналогично 'a', но из файла можно будет и читать.
- 'x' – создать и открыть файл только для записи, начав запись с чистого файла. Если файл уже существует, вызов `fopen()` завершится ошибкой, вернув `false`.
- 'x+' – аналогично 'x', но из файла можно будет и читать.
- 'c' – открыть файл только для записи. Если файл не существует, он создается. Если он существует, он не усекается (в отличие от 'w'), и вызов этой функции не завершается ошибкой (как в случае с 'x'). Указатель файла располагается в начале файла.
- 'c+' – аналогично 'c', но из файла можно будет и читать.

Если мы собираем данные от пользователей для дальнейшего хранения, чтобы поздравлять их с днем рождения, нам надо будет дописывать данные в конец файла, чтобы не уничтожать имеющиеся данные. Поэтому нам подойдет режим – 'a'.

```
$address = '/code/birthdays.txt';
$data = "Василий Васильев, 05-06-1992";
$fileHandler = fopen($address, 'a');
fwrite($fileHandler, $data);
fclose($fileHandler);
```

В этом примере мы открываем файл для записи, размещаем нужную нам строку в файл и закрываем файл с помощью функции `fclose()`.

Однако, наш скрипт всего лишь размещает одну и ту же строку. Давайте научим его запрашивать данные у пользователя.

Нам понадобится функция чтения пользовательского ввода в консоль – **readline**. Она выводит на экран переданный ей текст, а в ответ возвращает введенные пользователем данные.

```
$address = '/code/birthdays.txt';

$name = readline("Введите имя: ");
$date = readline("Введите дату рождения в формате ДД-ММ-ГГГГ: ");
$data = $name . ", " . $date . . "\r\n";

$fileHandler = fopen($address, 'a');

if(fwrite($fileHandler, $data)){
    echo "Запись $data добавлена в файл $address";
}
else {
```

```
        echo "Произошла ошибка записи. Данные не сохранены";  
    }  
  
    fclose($fileHandler);
```

🔥 Обратите внимание на то, что в конце каждой строки добавляются спецсимволы переноса строк – `\r\n`. Это сделано для того, чтобы одной строке соответствовал один пользователь.

Форматы файлов

Мы уже научились хранить данные в определенной структуре:

- Одна строка – один пользователь;
- Имя и дата рождения разделены запятой.

По сути, мы применили формат **CSV**, который хранит данные ровно в таком же виде.

Помимо него PHP поддерживает множество форматов файлов:

- CSV-файлы
- XML-файлы
- JSON-файлы

В зависимости от того, какой формат используется, существуют различные способы чтения и записи файлов.

Например, если вы работаете с CSV-файлом, вы можете использовать функцию **fgetcsv()** для чтения из файла в виде массива.

Пример:

```
$address = '/code/birthdays.txt';  
$fileHandle = fopen($address, 'r');  
while ($data = fgetcsv($fileHandle)) {  
    print_r($data);  
}
```

XML формат – более сложная структура. Она схожа с HTML. Расшифровывается как extended markup language. В ней можно сохранять данные с тегами.

Например, наш файл дней рождений в XML будет выглядеть так:

```
<?xml version="1.0" encoding="UTF-8"?>
<people>
  <person>
    <name>Василий Васильев</name>
    <birthdate>05-06-1992</birthdate>
  </person>
  <person>
    <name>Иван Иванов</name>
    <birthdate>05-12-1993</birthdate>
  </person>
</people>
```

Если вы работаете с XML-файлом, вы можете использовать функции SimpleXML или DOM для чтения и записи файлов.

Пример использования SimpleXML:

```
$xml = simplexml_load_file('file.xml');
print_r($xml);
```

В этом примере мы используем функцию **simplexml_load_file()** для загрузки содержимого файла в объект SimpleXMLElement. Затем мы выводим содержимое объекта SimpleXMLElement с помощью функции **print_r()**.

Здесь мы используем объекты, с которыми познакомимся чуть позже в этом курсе, чтобы вы могли полноценно продолжить работу с этой библиотекой функций.



Формат JSON (JavaScript Object Notation) – это легкий формат обмена данными, основанный на синтаксисе объектов JavaScript. Он используется для представления данных в виде текста, который может быть легко считан и понят другими программами.

JSON состоит из пар «ключ-значение», которые группируются в объекты или массивы. Ключи должны быть строками, а значения могут быть любым допустимым типом данных, включая другие объекты или массивы. Значения разделяются запятой, а пары «ключ-значение» – двоеточием.

Наш файл превратится вот в такой JSON документ:

```
{
  "name": "Василий Васильев",
  "birthday": "05-06-1992"
}
```

Пользователей у нас много, поэтому JSON также может быть представлен в виде массивов.

💡 Массивы начинаются с открывающейся скобки "[", а заканчиваются — закрывающейся скобкой "]".

💡 Значения в массиве разделяются запятой.

```
[
{ "name": "Василий Васильев", "birthday": "05-06-1992"}
{ "name": "Иван Иванов", "birthday": "05-12-1993"}
]
```

JSON часто используется для передачи данных между клиентской и серверной сторонами веб-приложений, а также для обмена данными между различными приложениями и сервисами. Он легкий, читабельный и легко парсится, что делает его очень популярным среди разработчиков. Так что мы еще не раз вернемся к нему.

Если вы работаете с JSON-файлом, вы можете использовать функции **json_encode()** и **json_decode()** для преобразования данных в формат JSON и обратно.

Пример:

```
$data = array('name' => 'Петр Петров', 'birthday' => '06-11-1998');
$json = json_encode($data);
// Раскодирование JSON
$decodedData = json_decode($json);
print_r($decodedData);
```

Поскольку это текстовый формат, его запись непосредственно в файл никак не отличается от обычной работы со строками.

Файлы с кодом

До этого момента мы работали в самой примитивной парадигме, где весь необходимый код хранится в одном файле. Но в реальных системах объём кода измеряется сотнями тысяч строк. И хранить такой код в одном файле и без системы разделения сложно и неудобно. Тем более, что код будут поддерживать множество инженеров.

Поэтому общепринятой практикой является раскладывание кода по файлам в соответствии со смыслом кода.

Пока мы работали только с элементарными конструкциями – функциями. Но уже с ними мы можем разделять наш код. Чуть позже мы познакомимся с более сложными концепциями организации кода.

Для примера давайте объединим функционал нашего хранилища пользователей в единое приложение, которое уже можно будет полноценно использовать.

Профессиональные программисты никогда не решают задачу «в лоб» – сначала они проектируют решение. Поэтому давайте и мы сначала поймем, как будет реализовано наше приложение.

Точка входа

У нас есть ряд разрозненных файлов, по которым разбросана логика. И это не очень здорово с архитектурной точки зрения.



Хорошей практикой является создание точки входа. Точка входа в приложение PHP – это файл, который является первым файлом, который запускается при обработке запросов на сервере. В то же время это единственный файл, который можно вызывать, если мы хотим запустить приложение.

Этот файл определяет конфигурацию и настройки приложения. Обычно точка входа имеет название "index.php" или "app.php" и находится в корневой директории приложения.

Точка входа является центральной точкой всего приложения и определяет, какие файлы будут загружены для обработки запросов. Все запросы к приложению направляются к точке входа, которая затем обрабатывает запросы, вызывает нужные функции и возвращает результат клиенту.

Очень важно обеспечить безопасность точки входа, так как она является уязвимым местом в приложении и может быть использована для атак на систему. Для этого можно использовать различные методы, такие как:

- проверка наличия корректных параметров запроса,
- фильтрация входных данных,
- проверка прав доступа.


Таким образом, вызов нашего приложения будет иметь следующий вид:

```
php app.php %command_name% %parameters%
```

То есть, в консоли мы будем передавать в качестве **command_name** желаемое действие (например, добавление пользователя в хранилище), а в **parameters** – данные (имя пользователя, день его рождения).

Нашей точкой входа будет файл `app.php`. Что мы уже можем в нем разместить?

- указание файла хранилища (но не обращение к нему);
- вызов корневой функции.

 Корневая функция в нашем случае – это та функция, вызов которой будет всегда стартовать приложение.

То есть, наш `app.php` на старте будет выглядеть примерно так:

```
// наш файл-хранилище
$storageFile = '/code/birthdays.txt';
// вызов корневой функции
$result = main($storageFile);
// вывод результата
echo $result;
```

Теперь нам нужно описать логику работы функции `main`.

Но мы уже условились, что хранить в файле вызова логику нельзя – в нём может находиться только:

- конфигурирование;
- вызов корневой функции;
- побочные эффекты.

Таким образом, функция `main` должна лежать в отдельном файле.

Но как тогда `app.php` будет знать о том, что делает функция `main`?

Для этого надо подключить к `app.php` наш файл с логикой!

Подключение файлов с кодом

В PHP есть несколько функций, которые позволяют подключать файлы из других файлов.



Функции **require** и **require_once** загружают указанный файл и вставляют его содержимое в текущий файл, как бы собирая его наподобие конструктора.

Если файл не найден, будет вызвана фатальная ошибка и выполнение скрипта будет прервано!



Разница между **require** и **require_once** заключается в том, что **require_once** загружает файл только один раз, в то время как **require** может загрузить файл несколько раз.

Для файлов с кодом **рекомендуется** использовать **require_once**, так как это гарантирует наличие файла, а также не позволяет загружать одну и ту же функцию несколько раз.

Также есть функции **include** и **include_once** – они работают так же, как **require** и **require_once**, за исключением того, что, если файл не найден, будет выдано предупреждение, но выполнение скрипта не будет прервано. Такое подключение для файлов с кодом не очень удобно, так как не гарантирует наличие вызываемой логики.

Независимо от того, какая функция используется для подключения файлов в PHP, необходимо быть осторожным при работе с подключаемыми файлами, особенно если файлы содержат конфиденциальную информацию или код, который может изменять состояние приложения.

Давайте создадим файл `main.function.php`, в котором будем хранить корневую функцию и функции ядра нашего приложения.



Корневая функция и функция ядра нашего приложения – это те функции, которые будут всегда выполняться вне зависимости от того, какое действие выполняется.

Например, нам всегда надо будет уметь разобрать команду пользователя. Но пока у нас там будет только функция `main`.

Для хранения функций создадим директорию `src`:

```
function main(string $ storageFileAddress) : string {  
  
}
```


Наша корневая функция будет принимать на вход строковый параметр адреса нашего хранилища, а в ответ будет отдавать строку с ответом для пользователя.

Подключим наш файл с кодом в `app.php`, добавив в начало строку:

```
// подключение файлов логики  
require_once('src/main.function.php');
```

Обратите внимание на то, как описывается здесь путь к файлу.

В Linux есть возможность указывать абсолютные и относительные пути. Это способы указать расположение файла или директории в файловой системе компьютера.



Абсолютный путь – это полный путь к файлу или директории от корневого каталога.

Он начинается с символа `'/'`, который указывает на корневой каталог, а затем следуют имена директорий, разделенные символом `'/'`.

Например, абсолютный путь к файлу `'birthdays.txt'` будет выглядеть так: **`"/code/birthdays.txt"`**.

Абсолютный путь всегда указывает на одну и ту же точку в файловой системе, независимо от того, где он вызывается.



Относительный путь – это путь к файлу или директории относительно текущей рабочей директории.

Он **НЕ** начинается с символа `'/'` и может содержать ссылки на родительские директории `'..'`.

Например, если мы захотим обратиться к `birthdays` из `src`, то относительный путь будет выглядеть так: **`"../birthdays.txt"`**.

Относительный путь всегда зависит от текущей рабочей директории, в которой вызывается команда или программа.

Наша функция должна сделать следующие шаги:

1. Проверить, что файл доступен для работы.
2. Подключить файл.


3. Считать команду и провалидировать её.
4. Выполнить действие и сформировать ответ.


Глядя на эти действия, можно предположить, что нам также потребуется:

- иметь набор функций для работы с хранилищем (file.function.php),
- обработчик команды (его можно разместить в main),
- компоновщик человеко-читаемого ответа (template.function.php).

Можем сразу создать эти функции и подключить их к главному файлу.

Теперь нам надо научиться обрабатывать команды от пользователя. В консольных вызовах PHP все данные, переданные при вызове команды будут храниться в суперглобальном массиве **`$_SERVER`**.


 Суперглобальные массивы в PHP – это специальные массивы, которые доступны в любом месте скрипта и содержат различную информацию, связанную с сервером, запросом пользователя и другими аспектами выполнения скрипта.

 Их не надо специально создавать – они формируются автоматически!

В PHP есть несколько суперглобальных массивов:

- `$_GET`: содержит данные, переданные в запросе HTTP методом GET через URL-параметры.
- `$_POST`: содержит данные, переданные в запросе HTTP методом POST через тело запроса.
- `$_REQUEST`: объединяет данные из `$_GET`, `$_POST` и `$_COOKIE`.
- `$_COOKIE`: содержит данные, переданные в запросе через cookie.
- `$_SERVER`: содержит информацию о сервере и текущем запросе, включая заголовки HTTP, пути к файлам и переменные окружения.
- `$_FILES`: содержит информацию о файлах, переданных на сервер через форму загрузки файлов.
- `$_ENV`: содержит переменные окружения.

Каждый элемент в этих массивах представляет собой переменную, которая хранит значение, переданное в запросе или сгенерированное сервером. Для доступа к элементам суперглобальных массивов можно использовать их имена, за которыми следует индекс элемента.

 Важно помнить, что данные в суперглобальных массивах необходимо всегда проверять и фильтровать, чтобы избежать возможных уязвимостей безопасности.

Нас будет интересовать переменная:

```
$_SERVER['argv']
```

В ней хранится массив переданных скрипту аргументов.

Например, если мы вызываем наш скрипт командой:

```
php app.php read-all
```

то в переменной сформируется следующий массив данных:

```
array(2) {
  [0]=>
    string(14) "/code/app.php"
  [1]=>
    string(8) "read-all"
}
```

То есть, в этом массиве первым элементом всегда идет имя скрипта, а затем уже все параметры вызова.

Как же будет работать наше приложение?

Для начала сделаем в нем несколько команд:

- **read-all** – эта команда будет выводить на экран все записи хранилища.
- **add** – эта команда будет добавлять в хранилище пользователя.
- **clear** – эта команда будет очищать хранилище.
- **help** – эта команда будет выводить список доступных в приложении команд.

Для обработки команд создадим функцию `parseCommand` в `main.function.php`. Поскольку массив `$_SERVER` суперглобальный, нам не нужно передавать в неё какие-то параметры.

В ответ она будет возвращать имя функции, которую надо вызвать:

```
function parseCommand() : string {  
  
}
```

Мы уже знаем, что имя команды всегда будет приходить элементом с индексом 1. Если его нет, то команда не передана, а значит, пользователь ошибся.

Также при попытке обращения к нему скрипт сгенерирует Warning, что увеличит время выполнения. Так что нам надо уметь обрабатывать такую ситуацию. В подобном случае в консольных вызовах принято возвращать правила использования команды, то есть по сути вызывать help.

Для валидации вызова применим оператор match. Функция будет иметь следующий вид:

```
function parseCommand() : ?string {  
    $functionName = 'helpFunction';  
  
    if(isset($_SERVER['argv'][1])) {  
        $functionName = match($_SERVER['argv'][1]) {  
            'read-all' => 'readAllFunction',  
            'add' => 'addFunction',  
            'clear' => 'clearFunction',  
            'help' => 'helpFunction',  
            default => 'helpFunction'  
        }  
    }  
  
    return $functionName;  
}
```

Теперь модифицируем нашу корневую функцию так, чтобы она могла начать выполнять действия:

```
function main(string $storageFileAddress) : string {  
    $functionName = parseCommand();  
  
    if(function_exists($functionName)) {  
        $result = $functionName($storageFileAddress);  
    }  
    else {  
        $result = handleError("Вызываемая функция не существует");  
    }  
  
    return $result;  
}
```

Как видите, к функции можно обращаться не только по явному имени, но и брать его из переменной. Но в таком случае важно проверять фактическое наличие функции в системе.

Пока у нас нет ни одной функции обработки, так что нам надо создать функцию `handleError` в файле `template.function.php`, чтобы выводить ошибку для пользователя:

```
function handleError(string $errorText) : string {
    return "\033[31m" . $errorText . " \r\n \033[97m";
}
```

Здесь мы добавляем два консольных кода:

- **\033[31m** – этот код переключает вывод консоли на красный цвет.
- **\033[97m** – этот код возвращает вывод консоли в белый цвет.

Остается только описать функции обработки команд.

Логика чтения из файла практически остается прежней.

Мы только обернем её в функцию:

```
function readAllFunction(string $address) : string {
    if (file_exists($address) && is_readable($address)) {
        $file = fopen($address, "rb");

        $contents = '';

        while (!feof($file)) {
            $contents .= fread($file, 100);
        }

        fclose($file);
        return $contents;
    }
    else {
        return handleError("Файл не существует");
    }
}
```

Логика добавления – аналогична:

```
function addFunction(string $address) : string {
    $name = readline("Введите имя: ");
    $date = readline("Введите дату рождения в формате ДД-ММ-ГГГГ: ");
    $data = $name . ", " . $date . "\r\n";

    $fileHandler = fopen($address, 'a');

    if(fwrite($fileHandler, $data)){
        return "Запись $data добавлена в файл $address";
    }
    else {
        return handleError("Произошла ошибка записи. Данные не сохранены");
    }
}
```

```
    fclose($fileHandler);  
}
```

Логика удаления данных из файла будет новой, но можно вспомнить про режимы чтения и применить режим «w», записав в файл пустой символ:

```
function clearFunction(string $address) : string {  
    if (file_exists($address) && is_readable($address)) {  
        $file = fopen($address, "w");  
  
        fwrite($file, '');  
  
        fclose($file);  
        return "Файл очищен";  
    }  
    else {  
        return handleError("Файл не существует");  
    }  
}
```

Выполнение же функции help мы полностью пробросим в template.function.php:

```
function help() : string {  
    return handleHelp();  
}
```

Само тело текста помощи:

```
function handleHelp() : string {  
    $help = "Программа работы с файловым хранилищем \r\n";  
  
    $help .= "Порядок вызова\r\n\r\n";  
  
    $help .= "php /code/app.php [COMMAND] \r\n\r\n";  
  
    $help .= "Доступные команды: \r\n";  
    $help .= "read-all - чтение всего файла \r\n";  
    $help .= "add - добавление записи \r\n";  
    $help .= "clear - очистка файла \r\n";  
    $help .= "help - помощь \r\n";  
  
    return $help;  
}
```

Итак, наше первое приложение полностью готово. Но остаётся ещё один важный аспект, который надо учесть.

Автоматизация подключения файлов

В самом начале работы над приложением мы вставили подключение файлов логики:

```
// подключение файлов логики
require_once('src/main.function.php');
require_once('src/template.function.php');
require_once('src/file.function.php');
```

Но как быть, если файлов будет гораздо больше? Контролировать подключение будет неудобно.



Здесь нам на помощь приходит **Composer** – это менеджер зависимостей для PHP, который позволяет управлять зависимостями PHP-приложения. Он упрощает процесс установки и обновления библиотек и фреймворков, используемых в проекте.

Composer использует файл конфигурации "composer.json", в котором определяются зависимости приложения, а также дополнительные настройки, такие как автозагрузка классов и настройки репозитория. При установке зависимостей Composer анализирует файл "composer.json" и загружает необходимые пакеты из зарегистрированных репозитория.

Composer позволяет управлять версиями зависимостей и их обновлением, а также решать конфликты между зависимостями. Он также позволяет создавать и публиковать свои пакеты на Packagist, который является репозиторием для пакетов, управляемых Composer.

Composer является широко используемым инструментом в экосистеме PHP, который значительно упрощает управление зависимостями и помогает сэкономить время при разработке приложений.

Давайте научимся применять его в нашем приложении! Для начала нам надо установить его внутри контейнера.

Опишем соответствующий Dockerfile, который мы разместим уровнем выше директории code:

```
FROM php:8.2
VOLUME /code
```

```
RUN curl -sS https://getcomposer.org/installer | php --  
--install-dir=/usr/local/bin --filename=composer  
  
ENV COMPOSER_ALLOW_SUPERUSER 1  
  
WORKDIR /code
```

Наш образ будет базироваться на версии PHP 8.2. Кроме наличия у него установленного Composer, он ничем не будет отличаться от базового образа.

Давайте построим образ php-cli-gb:

```
docker build -t "php-cli-gb"
```

Теперь нам нужно описать файл composer.json для нашего проекта. Если вы вспомните раздел о форматах файлов в данной лекции, то увидите, что этот файл имеет структуру JSON:

```
{  
  "name": "example/project",  
  "description": "Приложение работы с файловым хранилищем",  
  "type": "project",  
  "require": {  
    "php": "^8.0"  
  },  
  "autoload": {  
    "files": [  
      "src/main.function.php",  
      "src/template.function.php",  
      "src/file.function.php"  
    ]  
  }  
}
```

В начале мы указываем имя, описание и тип нашего приложения. Это необязательные поля, но они делают наш проект понятнее.

В блоке require мы указываем требования к нашему проекту. Здесь указана только версия PHP 8.0 и выше, так как оператор match появился именно в версии языка 8.

Теперь переходим к блоку автозагрузки. Сейчас мы делаем простую файловую автозагрузку, но с ходом курса будем её усложнять.

В подблоке files в массиве мы перечисляем адреса файлов, которые надо подключить.

После того, как наш composer.json готов в контейнере надо запустить установку зависимостей:

```
docker container run -it -v ${pwd}/code:/code/ php-cli-gb composer install
```


Команда `composer install` посмотрит на `composer.json` и сформирует код автозагрузчика в директории `vendor`. Если вы используете системы версионирования, то имейте в виду, что эта директория никогда не коммитится, так как должна собираться на конечной машине. Поэтому стоит сразу добавить её, например, в `.gitignore`.



Обратите внимание на то, что помимо команды `install` есть команда `update`.

Основные отличия между **`composer install`** и **`composer update`** следующие:

- `composer install` устанавливает все зависимости проекта, указанные в файле `composer.json`. Если файл `composer.lock` присутствует (а он создается после первого `install`), то устанавливаются версии зависимостей, указанные в этом файле. Если же файл `composer.lock` отсутствует, то Composer устанавливает последние доступные версии зависимостей.
- `composer update` обновляет зависимости проекта до последних доступных версий. Если файл `composer.lock` присутствует, то обновляются только те зависимости, которые были добавлены или изменены с момента последнего обновления. Если же файл `composer.lock` отсутствует, то обновляются все зависимости.



При использовании `composer update` следует быть осторожным, так как это может привести к конфликтам между зависимостями, если новые версии имеют несовместимые изменения.



Рекомендуется использовать `composer update` только при необходимости обновления зависимостей до новых версий, а `composer install` – для установки зависимостей на новой или чистой машине.

Теперь нам остаётся только заменить в точке входа (`app.php`) подключение файлов с громоздкого:

```
require_once('src/main.function.php');  
require_once('src/template.function.php');  
require_once('src/file.function.php');
```

на лаконичное:

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

Если же у нас появляются новые файлы логики, то мы добавляем их в `composer.json` и перед работой с приложением запускаем команду `composer install`, чтобы автозагрузчик обновил свой код.

Настройка приложения

Ещё одним аспектом, который мы должны добавить, зная правила подключения файлов, является настройка нашего приложения.

💡 Настройка приложения – это процесс определения параметров и конфигурации, необходимых для запуска приложения и его корректной работы.

Настройка может включать в себя такие параметры, как база данных, настройки безопасности, параметры среды и многое другое.

В большинстве случаев, настройка приложения осуществляется через файлы конфигурации, которые определяют значения параметров и настроек, используемых в приложении. В разных языках программирования и фреймворках для этого могут использоваться различные форматы файлов конфигурации, такие как XML, JSON, YAML и т.д.

Настройки приложения могут также задаваться через переменные окружения, которые позволяют задать значения параметров внутри операционной системы, а не в самом приложении. Это позволяет упростить управление настройками и обеспечить большую гибкость в настройке приложений.

💡 Хорошая настройка приложения может значительно повысить его производительность, безопасность и надежность, а также упростить его сопровождение и развертывание.

В нашем приложении есть точка хранения файла с данными. Хранить его адрес прямо в коде – дурной тон, ведь адрес может меняться, но это не должно влиять на логику. Поэтому подключим к нашему приложению файл конфигурации.

🔥 Это можно сделать, например, с помощью встроенной в PHP функции `parse_ini_file` – она будет читать файл определенного формата и сохранять его данные в массив настроек.

Создадим такой файл – config.ini:

```
[storage]
address=/code/birthdays.txt
```

Теперь для его чтения в file.function.php добавим функцию:

```
function readConfig(string $configAddress): array|false{
    return parse_ini_file($configAddress, true);
}
```

Обратите внимание, что она может возвращать как массив, так и булевское значение ЛОЖЬ.

Теперь, чтобы наше приложение могло работать с конфигурацией, чуть модифицируем его.

В корневую функцию main добавим блок чтения настроек:

```
function main(string $configFileAddress) : string {
    $config = readConfig($configFileAddress);

    if(!$config){
        return handleError("Невозможно подключить файл настроек");
    }

    $storageFileAddress = $config['storage']['address'];

    $functionName = parseCommand();

    if(function_exists($functionName)) {
        $result = $functionName($storageFileAddress);
    }
    else {
        $result = handleError("Вызываемая функция не существует");
    }

    return $result;
}
```

Теперь наша функция не зависит от жёсткого адреса хранилища.

Остаётся только поменять файл app.php:

```
<?php

require_once('vendor/autoload.php');

// вызов корневой функции
$result = main("/code/config.ini");
// вывод результата
echo $result;
```

Прямо в момент вызова мы говорим программе, где взять настройки. Теперь наш код завершен.

Работа с файловой системой

Часто мы можем не знать, какие именно файлы нам доступны. Поэтому нам может быть необходимо работать не с существующими файлами, а для начала создавать нужные директории, читать присутствующие в них файлы.

Предположим, что мы хотим написать логику, которая будет выводить на экран содержимое директории, в которой лежат нужные нам файлы – например, это будут разные файлы с информацией о пользователях.

Для того, чтобы создать такое приложение, нам нужно для начала создать папку, где будут лежать такие файлы. Назовём её `profiles`. Мы можем создать её вручную, но можем сразу же встроиться в логику так, чтобы при обращении к директории система сама создавала её в случае отсутствия.

Для начала опишем в конфигурации адрес нашей предполагаемой директории.

Добавим в `config.ini` строку:

```
[profiles]
address = /code/profiles/
```

Также нам надо будет добавить в точку входа обработчик.

Поэтому в `parseCommand` в файле `main.function.php` мы изменим код:

```
function parseCommand() : string {
    $functionName = 'helpFunction';

    if(isset($_SERVER['argv'][1])) {
        $functionName = match($_SERVER['argv'][1]) {
            'read-all' => 'readAllFunction',
            'add' => 'addFunction',
            'clear' => 'clearFunction',
            'read-profiles' => 'readProfilesDirectory',
            'help' => 'helpFunction',
            default => 'helpFunction'
        };
    }

    return $functionName;
}
```

Как видите, обрабатывать новую команду будет функция `readProfilesDirectory`.

Добавим её (пока без описания логики) в `file.function.php`:

```
function readProfilesDirectory(): string {  
}
```



Внутри функции мы должны проверять, что директория существует.

За это отвечает встроенная в PHP функция `is_dir` – она проверяет, является ли переданный адрес директорией, и возвращает `true` или `false` в зависимости от результата.

Но, как мы помним, пока наши функции принимали на вход адрес хранилища дней рождений. Здесь он нам не пригодится.

Но нужно будет как-то получить доступ к конфигурации приложения. Значит, пора сделать рефакторинг нашего кода.



Рефакторинг кода – это процесс изменения внутренней структуры программного кода, без изменения его внешнего поведения, с целью улучшения его качества, читаемости, понимаемости, поддерживаемости и расширяемости.

Во время рефакторинга код переписывается таким образом, чтобы устранить недочеты, улучшить его организацию и сделать его более оптимальным, но при этом его функциональность остается неизменной.

Процесс рефакторинга может включать следующие действия:

1. Переименование переменных, функций, сущностей, чтобы сделать их имена более понятными и описательными.
2. Выделение повторяющегося кода в отдельные функции или классы для уменьшения дублирования кода.
3. Разбиение больших функций или классов на более мелкие и управляемые куски, чтобы повысить читаемость и понимаемость кода.
4. Изменение структуры данных и алгоритмов для повышения эффективности и производительности программы.
5. Упрощение сложных условных конструкций и циклов, чтобы уменьшить сложность кода.

6. Добавление комментариев и документации для повышения понимаемости кода другими разработчиками.

Рефакторинг является важным инструментом для поддержания и улучшения качества кодовой базы. Он позволяет делать программное обеспечение более надежным, легким в сопровождении и улучшает процесс разработки, так как облегчает работу с кодом для разработчиков.

Переделаем код так, чтобы при вызове команды передавалась конфигурация, а не конкретный адрес.

В функции `main` в `main.function.php` мы заменим вызов с:

```
$result = $functionName($storageFileAddress);
```

на:

```
$result = $functionName($config);
```

Это также позволит нам избавиться от строки:

```
$storageFileAddress = $config['storage']['address'];
```

Так как теперь адрес будет определяться на уровне функций работы с хранилищем. Поэтому их тоже нужно переписать.

Например, функция `readAllFunction` теперь будет выглядеть так:

```
function readAllFunction(array $config) : string {  
    $address = $config['storage']['address'];
```

Как видите, изменение обошлось нам не очень дорого – всего одна строка. Но нужно также модифицировать остальные функции.

Вернувшись к нашей функции чтения директории, мы можем наконец-то сформировать обращение к директории:

```
function readProfilesDirectory(array $config): string {  
    $profilesDirectoryAddress = $config['profiles']['address'];
```

```
if(!is_dir($profilesDirectoryAddress)){
    mkdir($profilesDirectoryAddress);
}
return "";
}
```

Итак, мы проверяем, существует ли наша директория.

Если нет, то создаём её. Давайте для теста запустим наш скрипт:


```
php /code/app.php read-profiles
```


Как видите, директория создалась.

Давайте наполним её файлами с профилями примерно следующего содержания:

```
{
    "name": "Иван",
    "lastname": "Иванов"
}
```

Вы наверняка заметили, что это формат JSON. Нам надо вывести список файлов в директории. А при вводе конкретного имени файла выводить содержимое.

 Список файлов можно сформировать при помощи встроенной в PHP функции `scandir` – это функция, которая используется для сканирования содержимого директории и возврата массива элементов, представляющих содержимое этой директории. Она предоставляет информацию о файлах и поддиректориях, находящихся в указанной директории, в виде массива.

 Обратите внимание на то, что первыми двумя элементами в этом массиве всегда будут две ссылки – на текущую директорию и на директорию выше.

Поэтому эти два элемента получаемого массива можно сразу отсекать:

```
function readProfilesDirectory(array $config): string {
    $profilesDirectoryAddress = $config['profiles']['address'];

    if(!is_dir($profilesDirectoryAddress)){
        mkdir($profilesDirectoryAddress);
    }

    $files = scandir($profilesDirectoryAddress);
```

```

$result = "";

if(count($files) > 2){
    foreach($files as $file){
        if(in_array($file, ['.', '..']))
            continue;

        $result .= $file . "\r\n";
    }
}
else {
    $result .= "Директория пуста \r\n";
}

return $result;
}

```

Разберем полученную функцию.

Итак, мы считываем содержимое директории в массив `files`. Если количество элементов в нем больше 2, то мы считаем, что директория не пуста. Два элемента, которые всегда будут в нем – это как раз те самые ссылки.

Далее мы создаем пустую переменную `result`, в которую будем формировать ответ от скрипта. Сам массив мы можем обойти при помощи цикла `foreach`, проверяя, что нам не встретились ссылки, которые не представляют для нас ценности. В итоге мы выводим имена `json` файлов, содержащихся в директории.

Остаётся только вывести содержимое на экран.

Для этого создадим новые функции:

```
'read-profile' => 'readProfile'
```

Команда будет выглядеть следующим образом:

```
php /code/app.php read-profile IvanovIvan
```

Внутри функции нам нужно будет проверить, что команда сформирована корректно:

```

if(!isset($_SERVER['argv'][2])){
    return handleError("Не указан файл профиля");
}

```

Параметр вызова легко достать из суперглобального массива `SERVER`.

Далее мы формируем адрес файла, который будем пытаться считать:


```
$profileFileName = $profilesDirectoryAddress . $_SERVER['argv'][2] . ".json";

if(!file_exists($profileFileName)){
    return handleError("Файл $profileFileName не существует");
}
```

Даже если команда передана верно, может быть указано ошибочное имя.

Эту ситуацию также надо обработать:

```
$contentJson = file_get_contents($profileFileName);
$contentArray = json_decode($contentJson, true);
```

Затем мы читаем файл в переменную и при помощи [json_decode](#) превращаем полученные данные в формате json в массив.

Затем формируем строку вывода:

```
$info = "Имя: " . $contentArray['name'] . "\r\n";
$info .= "Фамилия: " . $contentArray['lastname'] . "\r\n";

return $info;
```

Заключение

В этой лекции мы не только научились сохранять данные между вызовами скриптов, но и овладели крайне важными навыками:

- научились структурировать свой код;
- автоматизировали подключение файлов;
- собрали своё первое полноценное приложение.

Домашнее задание

1. Обработка ошибок. Посмотрите на реализацию функции в файле fwrite-cli.php в исходниках. Может ли пользователь ввести некорректную информацию (например, дату в виде 12-50-1548)? Какие еще некорректные данные могут быть введены? Исправьте это, добавив соответствующие обработки ошибок.

2. Поиск по файлу. Когда мы научились сохранять в файле данные, нам может быть интересно не только чтение, но и поиск по нему. Например, нам надо проверить,

кого нужно поздравить сегодня с днем рождения среди пользователей, хранящихся в формате

Василий Васильев, 05-06-1992

И здесь нам на помощь снова приходят циклы. Понадобится цикл, который будет построчно читать файл и искать совпадения в дате. Для обработки строки пригодится функция `explode`, а для получения текущей даты – `date`.

3. Удаление строки. Когда мы научились искать, надо научиться удалять конкретную строку. Запросите у пользователя имя или дату для удаляемой строки. После ввода либо удалите строку, оповестив пользователя, либо сообщите о том, что строка не найдена.

4. Добавьте новые функции в итоговое приложение работы с файловым хранилищем.

Что можно почитать еще?

1. <https://phptherightway.com/>
2. <https://php.net>
3. PHP 8 | Котеров Дмитрий Владимирович
4. <https://stackoverflow.com/questions/34034730/how-to-enable-color-for-php-cli> - как покрасить вывод консоли
5. <https://phptoday.ru/post/gotovim-lokalnuyu-sredu-docker-dlya-razrabotki-na-php> - варианты образа Composer