Base

1. Что такое ссылки?

Ссылки в **PHP** позволяют ссылаться на область памяти, где расположено значение переменной или параметра. Для создания **ссылки** перед переменной указывается символ амперсанда-&.

2. Каковы основные операции с использованием ссылок?

Есть три основных операции с использованием ссылок: присвоение по ссылке, передача по ссылке и возврат по ссылке.

Первая из них - ссылки PHP позволяют создать две переменные указывающие на одно и то же значение.

Второе, что делают ссылки - передача параметров по ссылке. При этом локальная переменная в функции и переменная в вызывающей области видимости ссылаются на одно и то же содержимое.

Третье, что могут делать ссылки - это возврат по ссылке. Возврат по ссылке используется в тех случаях, когда вы хотите использовать функцию для выбора переменной, с которой должна быть связана данная ссылка. *Не* используйте возврат по ссылке для увеличения производительности. Ядро PHP само занимается оптимизацией. Применяйте возврат по ссылке только имея технические причины на это. Для возврата по ссылке используйте такой синтаксис:

```
class foo {
  public $value = 42;
  public function &getValue() {
    return $this->value;
  }
}

$obj = new foo;
$myValue = &$obj->getValue(); // $myValue указывает на $obj->value,
равное 42.
$obj->value = 2;
echo $myValue;                // отобразит новое значение $obj->value, то
есть 2.
```

3. Назовите простые типы данных, поддерживаемые в PHP.

- String
- Integer
- Float (числа с плавающей запятой, также называемые Double)
- Boolean
- Array (составной)

- Object (составной)
- NULL (специальный)
- Resource (специальный)

дополнительные типы:

- callable (составной). Функции обратного вызова (callback-функции)
- iterable (составной). Может использоваться как тип параметра для указания, что функция принимает набор значений, но ей не важна форма этого набора, пока он будет использоваться с foreach.

4. Что такое инкремент и декремент, в чем разница между префиксным и постфиксным инкрементом и декрементом?

Инкремент

Оператор инкремента, обозначается знаком ++ и может быть расположен с любой стороны от операнда, с которым он работает. Он увеличивает это значение на единицу, точно также, как при прибавлении единицы к значению. Фактический результат зависит от того, где был применен оператор, до или после операнда, с которым он применялся. Данный оператор часто используется с переменными, и зачастую это происходит внутри циклов (про циклы будет рассказано далее).

Префиксная форма инкремента

Префиксная форма - это когда оператор инкремента расположен перед операндом, такая форма записи означает то, что инкремент будет выполнен первым: он увеличивает значение операнда на единицу и только потом уже выполняется вся остальная часть инструкции:

$num1 = 2;

$result = ++$num1;

echo $result;

Постфиксная форма инкремента

Постфиксная форма записывается немного по другому - инкремент располагается в этом случае после операнда. При постфиксной форме записи первое использование операнда возвращает его текущее значение, только после этого значение будет увеличено на единицу:

```
$num1 = 2;
echo $num1++;  // операнд вернет 2, затем увеличит значение
echo $num1; // значение операнда 3
```

Декремент

Оператор декремента, обозначается знаком --, и в отличие от оператора инкремента, уменьшает, а не увеличивает, на единицу значение своего операнда. Декремент также допускает префиксную и постфиксную форму записи:

```
// префиксная форма декремента
$num1 = 5;
echo --$num1;  // сначала выполняется декремент, затем выводится 4
echo $num1; // значение операнда 4
// постфиксная форма декремента
$num1 = 5;
echo $num1--;  // операнд вернет 5, затем уменьшит значение
echo $num1; // значение операнда 4
```

5. Что такое рекурсия?

Нерекурсивные функции (другими словами, функции, которые вы использовали в прошлом), запускались единожды при каждом вызове и возвращали результат благодаря инструкции возврата return.

А вот рекурсивная функция, вызванная единожды, может затем вызывать сама себя неопределенное число раз, прежде чем скомбинирует результат всех вызовов и вернет его по инструкции return.

6. В чем разница между =, == и ===?

Равным (=) является оператор присваивания, который устанавливает переменную слева от = в значение выражения, которое находится справа.

Двойное равенство (==) — это оператор сравнения, который преобразует операнды одного типа перед сравнением (https://www.php.net/manual/ru/types.comparisons.php).

=== (Triple equals) — это оператор сравнения строгого равенства, который возвращает false для значений, которые не относятся к аналогичному типу.

7. Какие знаете принципы ООП?

Объектно-ориентированное программирование базируется на принципах:

- инкапсуляция
- наследование
- абстракция
- полиморфизм

Инкапсуляция — объединение полей и методов в классе, с целью закрыть прямой доступ к полям и открыть его для методов, которые этими полями управляют.

Наследование — позволяет создавать классы на основе уже существующих. Тем самым облегчая задачу по созданию новых классов с точки зрения использования уже существующего программного кода. Класс, от которого произошло наследование, называется базовым или родительским. Классы, которые произошли от базового, называются потомками, наследниками или производными классами. В PHP также используются абстрактные классы.

Абстрагирование – это способ выделить набор значимых характеристик объекта, исключая из рассмотрения незначимые. Соответственно, **абстракция** – это набор всех таких характеристик.

Абстрактный класс — это класс, содержащий хотя бы один абстрактный метод. Он описан в программе, имеет поля, методы и не может использоваться для непосредственного создания объекта. То есть от абстрактного класса можно только наследовать. Объекты создаются только на основе производных классов, наследованных от абстрактного.

Полиморфизм (греч. «многообразие форм») — в программировании означает возможность использования одного имени для методов разных классов находящихся в одной иерархии наследования (т.е. в родственных классах) с целью выполнения схожих действий.

8. Какая система типов используется в PHP? Опишите плюсы и минусы.

Объявления типов могут использоваться для аргументов функций, возвращаемых значений и, начиная с PHP 7.4.0, для свойств класса. Они используются во время исполнения для проверки, что значение имеет точно тот тип, который для них указан. В противном случае будет выброшено исключение TypeError.

mixed эквивалентен объединению типов object|resource|array|string|int|float|bool|null. Доступно с PHP 8.0.0.

Можно комбинировать простые типы в составные типы. PHP позволяет комбинировать типы следующими способами:

- Объединение простых типов. Начиная с PHP 8.0.0.
- Пересечение типов классов (интерфейсы и имена классов). Начиная с PHP 8.1.0.

По умолчанию, PHP будет преобразовывать значения неправильного типа в ожидаемые. К примеру, если в функцию передать параметр типа int в аргумент, объявленный как string, то он преобразуется в string.

Можно включить режим строгой типизации на уровне файла. В этом режиме, тип значения должен строго соответствовать объявленному, иначе будет выброшено исключение TypeError. Единственным исключением из этого правила является передача значения типа int туда, где ожидается float.

9. Чем отличаются ключевые слова: include и require, mysql_connect и mysql_pconnect?

Выражение include включает и выполняет указанный файл.

require аналогично include, за исключением того, что в случае возникновения ошибки он также выдаст фатальную ошибку уровня **E_COMPILE_ERROR**. Другими словами, он остановит выполнение скрипта, тогда как include только выдал бы предупреждение **E_WARNING**, которое позволило бы скрипту продолжить выполнение.

mysql_connect — Открывает соединение с сервером MySQL

mysql_pconnect — Устанавливает постоянное соединение с сервером MySQL

Данный модуль устарел, начиная с версии PHP 5.5.0, и удалён в PHP 7.0.0. Используйте вместо него MySQLi или PDO_MySQL. Смотрите также инструкцию MySQL: выбор API. Альтернативы для данной функции:

- mysqli_connect() с p: префиксом хоста
- PDO::__construct() с опцией драйвера **PDO::ATTR_PERSISTENT**

10. Что такое интерфейсы? Используете ли вы их? Если да — расскажите об этом.

Интерфейсы объектов позволяют создавать код, который указывает, какие методы должен реализовать класс, без необходимости определять, как именно они должны быть реализованы. Интерфейсы разделяют пространство имён с классами и трейтами, поэтому они не могут называться одинаково.

Интерфейсы объявляются так же, как и обычные классы, но с использованием ключевого слова interface вместо class. Тела методов интерфейсов должны быть пустыми.

Все методы, определённые в интерфейсах, должны быть общедоступными, что следует из самой природы интерфейса.

На практике интерфейсы используются в двух взаимодополняющих случаях:

- Чтобы позволить разработчикам создавать объекты разных классов, которые могут использоваться взаимозаменяемо, поскольку они реализуют один и тот же интерфейс или интерфейсы. Типичный пример - несколько служб доступа к базе данных, несколько платёжных шлюзов или разных стратегий кеширования. Различные реализации могут быть заменены без каких-либо изменений в коде, который их использует.
- Чтобы разрешить функции или методу принимать и оперировать параметром, который соответствует интерфейсу, не заботясь о том, что ещё может делать

объект или как он реализован. Эти интерфейсы часто называют Iterable, Cacheable, Renderable и так далее, чтобы описать их поведение.

Интерфейсы могут определять магические методы, требуя от реализующих классов реализации этих методов.

Для реализации интерфейса используется оператор implements. Класс должен реализовать все методы, описанные в интерфейсе, иначе произойдёт фатальная ошибка. При желании классы могут реализовывать более одного интерфейса, разделяя каждый интерфейс запятой.

11. Что такое абстрактный класс и чем он отличается от интерфейса?

Абстрактные классы представляют собой классы, предназначенные для наследования от них. При этом объекты таких классов нельзя создать.

PHP поддерживает определение абстрактных классов и методов. На основе абстрактного класса нельзя создавать объекты, и любой класс, содержащий хотя бы один абстрактный метод, должен быть определён как абстрактный. Методы, объявленные абстрактными, несут, по существу, лишь описательный смысл и не могут включать реализацию.

При наследовании от абстрактного класса, все методы, помеченные абстрактными в родительском классе, должны быть определены в дочернем классе и следовать обычным правилам наследования и совместимости сигнатуры.

12. Может ли абстрактный класс содержать частный метод?

да

13. Какие модификаторы видимости есть в PHP?

С помощью специальных модификаторов можно задать область видимости для свойств и методов класса. В PHP есть три таких модификатора: public : к свойствам и методам, объявленным с данным модификатором, можно обращаться из внешнего кода и из любой части программы protected : свойства и методы с данным модификатором доступны из текущего класса, а также из классов-наследников private : свойства и методы с данным модификатором доступны только из текущего класса

14. Какие магические методы вы знаете, и как их применяют?

Магические методы - это специальные методы, которые переопределяют действие PHP по умолчанию, когда над объектом выполняются определённые действия.

Следующие названия методов считаются магическими: __construct(), __destruct(), __call(), __callStatic(), __get(), __set(), __isset(), __unset(), __sleep(), __wakeup(),

__serialize(), __unserialize(), __toString(), __invoke(), __set_state(), __clone() и __debugInfo()

Все магические методы, за исключением __construct(), __destruct() и __clone(), *ДОЛЖНЫ* быть объявлены как public, в противном случае будет вызвана ошибка уровня **E_WARNING**. До PHP 8.0.0 для магических методов __sleep(), __wakeup(), __serialize(), __unserialize() и __set_state() не выполнялась проверка.

__call() запускается при вызове недоступных методов в контексте объект.

__callStatic() запускается при вызове недоступных методов в статическом контексте.

Метод __set() будет выполнен при записи данных в недоступные (защищённые или приватные) или несуществующие свойства.

Метод __get() будет выполнен при чтении данных из недоступных (защищённых или приватных) или несуществующих свойств.

Метод __isset() будет выполнен при использовании isset() или empty() на недоступных (защищённых или приватных) или несуществующих свойствах.

Метод __unset() будет выполнен при вызове unset() на недоступном (защищённом или приватном) или несуществующем свойстве.

Функция serialize() проверяет, присутствует ли в классе метод с магическим именем __sleep(). Если это так, то этот метод выполняется до любой операции сериализации. Он может очистить объект и должен возвращать массив с именами всех переменных этого объекта, которые должны быть сериализованы. Если метод ничего не возвращает, то сериализуется **null** и выдаётся предупреждение **E_NOTICE**.

Предполагаемое использование __sleep() состоит в завершении работы над данными, ждущими обработки или других подобных задач очистки. Кроме того, этот метод может быть полезен, когда есть очень большие объекты, которые нет необходимости полностью сохранять.

С другой стороны, функция unserialize() проверяет наличие метода с магическим именем __wakeup(). Если она имеется, эта функция может восстанавливать любые ресурсы, которые может иметь объект.

Предполагаемое использование __wakeup() заключается в восстановлении любых соединений с базой данных, которые могли быть потеряны во время операции сериализации и выполнения других операций повторной инициализации.

serialize() проверяет, есть ли в классе функция с магическим именем __serialize(). Если да, функция выполняется перед любой сериализацией. Она должна создать и вернуть ассоциативный массив пар ключ/значение, которые представляют сериализованную форму объекта. Если массив не возвращён, будет выдано TypeError.\

Если и __serialize() и __sleep() определены в одном и том же объекте, будет вызван только метод __serialize(). __sleep() будет игнорироваться. Если объект реализует

интерфейс Serializable, метод serialize() интерфейса будет игнорироваться, а вместо него будет использован __serialize().

Предполагаемое использование __serialize() заключается в определении удобного для сериализации произвольного представления объекта. Элементы массива могут соответствовать свойствам объекта, но это не обязательно.

И наоборот, unserialize() проверяет наличие магической функции __unserialize(). Если функция присутствует, ей будет передан восстановленный массив, который был возвращён из __serialize(). Затем он может восстановить свойства объекта из этого массива соответствующим образом.

Если и __unserialize() и __wakeup() определены в одном и том же объекте, будет вызван только метод __unserialize(). __wakeup() будет игнорироваться.

Функция доступна с PHP 7.4.0.

Метод __toString() позволяет классу решать, как он должен реагировать при преобразовании в строку. Например, что вывести при выполнении echo $obj;.

Начиная с PHP 8.0.0, возвращаемое значение следует стандартной семантике типа PHP, что означает, что оно будет преобразовано в строку (string), если возможно, и если strict typing отключён.

Начиная с PHP 8.0.0, любой класс, содержащий метод __toString(), также будет неявно реализовывать интерфейс Stringable и, таким образом, будет проходить проверку типа для этого интерфейса В любом случае рекомендуется явно реализовать интерфейс.

В PHP 7.4 возвращаемое значение *ДОЛЖНО* быть строкой (string), иначе выдаётся Error.

До PHP 7.4.0 возвращаемое значение *должно* быть строкой (string), в противном случае выдаётся фатальная ошибка **E_RECOVERABLE_ERROR**. is emitted.

Нельзя выбросить исключение из метода __toString() до PHP 7.4.0. Это приведёт к фатальной ошибке.

Метод __invoke() вызывается, когда скрипт пытается выполнить объект как функцию.

__set_state() вызывается для тех классов, которые экспортируются функцией var_export().

Единственным параметром этого метода является массив, содержащий экспортируемые свойства в виде ['property' => value, ...].

__debugInfo() вызывается функцией var_dump(), когда необходимо вывести список свойств объекта. Если этот метод не определён, тогда будут выведены все свойства объекта с модификаторами public, protected и private.

15. Что такое генераторы и как их использовать?

Генераторы предоставляют лёгкий способ реализации простых итераторов без использования дополнительных ресурсов или сложностей, связанных с реализацией класса, реализующего интерфейс Iterator.

Генератор позволяет вам писать код, использующий foreach для перебора набора данных без необходимости создания массива в памяти, что может привести к превышению лимита памяти, либо потребует довольно много времени для его создания. Вместо этого, вы можете написать функцию-генератор, которая, по сути, является обычной функцией, за исключением того, что вместо возврата единственного значения, генератор может возвращать (yield) столько раз, сколько необходимо для генерации значений, позволяющих перебрать исходный набор данных.

Наглядным примером вышесказанного может послужить использование функции range() как генератора. Стандартная функция range() генерирует массив, состоящий из значений, и возвращает его, что может привести к генерации огромных массивов данных. Например, вызов **range(0, 1000000)** приведёт к использованию более 100 МБ оперативной памяти.

В качестве альтернативы мы можем создать генератор xrange(), который использует память только для создания объекта Iterator и сохранения текущего состояния, что потребует не больше 1 килобайта памяти.

```
function xrange($start, $limit, $step = 1) {
  if ($start <= $limit) {
    if ($step <= 0) {
      throw new LogicException('Шаг должен быть положительным');
    }
    for ($i = $start; $i <= $limit; $i += $step) {
      yield $i;
    }
  } else {
    if ($step >= 0) {
      throw new LogicException('Шаг должен быть отрицательным');
    }
    for ($i = $start; $i >= $limit; $i += $step) {
      yield $i;
    }
  }
}
/* Обратите внимание, что и range() и xrange() дадут один и тот же вывод
*/
echo 'Нечётные однозначные числа с помощью range():  ';
foreach (range(1, 9, 2) as $number) {
  echo "$number ";
}
echo "\n";
echo 'Нечётные однозначные числа с помощью xrange(): ';
```

```php
foreach (xrange(1, 9, 2) as $number) {
  echo "$number ";
}
```

Когда функция генератор вызывается, она вернёт объект встроенного класса Generator. Этот объект реализует интерфейс Iterator, станет однонаправленным объектом итератора и предоставит методы, с помощью которых можно управлять его состоянием, включая передачу в него и возвращения из него значений.

16. Что делает оператор yield?

Генератор в целом выглядит как обычная функция, за исключением того, что вместо возвращения одного значения, генератор будет перебирать столько значений, сколько необходимо. Любая функция, содержащая yield, является функцией генератора.

Когда вызывается генератор, он возвращает объект, который можно итерировать. Когда вы итерируете этот объект (например, в цикле foreach), PHP вызывает методы итерации объекта каждый раз, когда вам нужно новое значение, после чего сохраняет состояние генератора и при следующем вызове возвращает следующее значение.

Когда все значения в генераторе закончились, генератор просто завершит работу, ничего не вернув. После этого основной код продолжит работу, как если бы в массиве закончились элементы для перебора.

Вся суть генератора заключается в ключевом слове **yield**. В самом простом варианте оператор "yield" можно рассматривать как оператор "return", за исключением того, что вместо прекращения работы функции, "yield" только приостанавливает её выполнение и возвращает текущее значение, и при следующем вызове функции она возобновит выполнения с места, на котором прервалась.

17. Что такое traits? Альтернативное решение? Приведите пример.

Трейт - это механизм обеспечения повторного использования кода в языках с поддержкой только одиночного наследования, таких как PHP. Трейт предназначен для уменьшения некоторых ограничений одиночного наследования, позволяя разработчику повторно использовать наборы методов свободно, в нескольких независимых классах и реализованных с использованием разных архитектур построения классов. Семантика комбинации трейтов и классов определена таким образом, чтобы снизить уровень сложности, а также избежать типичных проблем, связанных с множественным наследованием и смешиванием (mixins).

Трейт очень похож на класс, но предназначен для группирования функционала хорошо структурированным и последовательным образом. Невозможно создать самостоятельный экземпляр трейта. Это дополнение к обычному наследованию и позволяет сделать горизонтальную композицию поведения, то есть применение членов класса без необходимости наследования.

Если два трейта добавляют метод с одним и тем же именем, это приводит к фатальной ошибке в случае, если конфликт явно не разрешён.

Для разрешения конфликтов именования между трейтами, используемыми в одном и том же классе, необходимо использовать оператор insteadof для того, чтобы точно выбрать один из конфликтующих методов.

Так как предыдущий оператор позволяет только исключать методы, оператор as может быть использован для включения одного из конфликтующих методов под другим именем. Обратите внимание, что оператор as не переименовывает метод и не влияет на какой-либо другой метод.

Трейты могут использоваться как в классах, так и в других трейтах. Используя один или более трейтов в определении другого трейта, он может частично или полностью состоять из членов, определённых в этих трейтах.

Трейты поддерживают использование абстрактных методов для того, чтобы установить требования к использующему классу. Поддерживаются общедоступные, защищённые и закрытые методы. До PHP 8.0.0 поддерживались только общедоступные и защищённые абстрактные методы.

В трейтах можно определять статические переменные, статические методы и статические свойства.

Трейты могут также определять свойства.

18. Опишите поведение при использовании traits с одинаковыми именами полей и / или методов.

Если два трейта добавляют метод с одним и тем же именем, это приводит к фатальной ошибке в случае, если конфликт явно не разрешён.

Для разрешения конфликтов именования между трейтами, используемыми в одном и том же классе, необходимо использовать оператор insteadof для того, чтобы точно выбрать один из конфликтующих методов.

Так как предыдущий оператор позволяет только исключать методы, оператор as может быть использован для включения одного из конфликтующих методов под другим именем. Обратите внимание, что оператор as не переименовывает метод и не влияет на какой-либо другой метод.

19. Будут ли доступны частные методы trait в классе?

да

20. Можно ли компоновать traits в trait?

да

21. Расскажите об обработке ошибок и исключения (try catch, finaly и throw).

Модель исключений (exceptions) в PHP похожа с используемыми в других языках программирования. Исключение можно сгенерировать (выбросить) при помощи оператора [throw], и можно перехватить (поймать) оператором [catch]. Код генерирующий исключение, должен быть окружён блоком [try], для того, чтобы можно было перехватить исключение. Каждый блок [try] должен иметь как минимум один соответствующий ему блок [catch] или [finally].

В случае, если выброшено исключение, для которого нет блока [catch] в текущей функции, это исключение будет "всплывать" по стеку вызова, пока не будет найден подходящий блок [catch]. При этом, все встреченные блоки [finally] будут исполнены. Если стек вызовов раскрутится до глобальной области видимости, не встретив подходящего блока [catch], программа завершит работу с фатальной ошибкой, если только у вас не настроен глобальный обработчик исключений.

Генерируемый объект должен принадлежать классу Exception или наследоваться от Exception. Попытка сгенерировать исключение другого класса приведёт к фатальной ошибке PHP.

Начиная с PHP 8.0.0, ключевое слово [throw] является выражением и может использоваться в контексте других выражений. В более ранних версиях оно являлось оператором и требовало размещения в отдельной строке.

Блок [catch] определяет то, как следует реагировать на выброшенное исключение. В блоке [catch] указывается один или более типов исключений или ошибок(Error), которые он будет обрабатывать. Также указывается и переменная, которой будет присвоено пойманное исключение (начиная с PHP 8.0.0 задавать эту переменную не обязательно). Выброшенное исключение или ошибка будут обработаны первым подходящим блоком [catch].

Можно использовать несколько блоков [catch], перехватывающих различные классы исключений. Нормальное выполнение (когда не генерируются исключения в блоках [try]) будет продолжено за последним блоком [catch]. Исключения могут быть сгенерированы (или вызваны ещё раз) оператором [throw] внутри блока [catch]. Если нет, то исполнение будет продолжено после отработки блока [catch].

При генерации исключения код, следующий после описываемого выражения, не будет выполнен, а PHP попытается найти первый блок [catch], перехватывающий исключение данного класса. Если исключение не будет перехвачено, PHP выдаст фатальную ошибку: "Uncaught Exception ..." (Неперехваченное исключение), если не был определён обработчик ошибок при помощи функции set_exception_handler().

Начиная с PHP 7.1.0, блок [catch] может принимать несколько типов исключений с помощью символа (|). Это полезно, когда разные исключения из разных иерархий классов обрабатываются одинаково.

Начиная с PHP 8.0.0, задание переменной для пойманного исключения опционально. Если она не задана, блок [catch](<https://www.php.net/manual/ru/language.exceptions.php#language.exceptions.catch>) будет исполняться, но не будет иметь доступа к объекту исключения.

Блок [finally](<https://www.php.net/manual/ru/language.exceptions.php#language.exceptions.finally>) также можно использовать после или вместо блока [catch](<https://www.php.net/manual/ru/language.exceptions.php#language.exceptions.catch>). Код в блоке [finally](<https://www.php.net/manual/ru/language.exceptions.php#language.exceptions.finally>) всегда будет выполняться после кода в блоках [try](<https://www.php.net/manual/ru/language.exceptions.php>) и [catch](<https://www.php.net/manual/ru/language.exceptions.php#language.exceptions.catch>), независимо от того, было ли выброшено исключение, перед тем как продолжится нормальное выполнение кода.

Одно важное взаимодействие происходит между блоком [finally] и оператором [return]. Если оператор [return] встречается внутри блоков [try] или [catch], блок [finally] всё равно будет выполнен. Кроме того, оператор [return] выполняется, когда встречается, но результат будет возвращён после выполнения блока [finally]. Если блок [finally] также содержит оператор [return], возвращается значение, указанное в блоке [finally].

22. Что такое type hinting, как работает, зачем нужен?

Type Hinting — это механизм, который позволяет явно указывать типы параметров. Интерпретатор использует их и применяет исключение в тех ситуациях, когда тип не соответствует ожидаемому. Так работает в большинстве языков, где присутствует этот механизм.

В PHP используется слабая типизация, поэтому здесь все чуть сложнее. Можно жестко указать тип int, но это не значит, что интерпретатор начнет ругаться на все остальное. Он выполнит автоматическое приведение и спокойно обработает и логический, и строковый тип данных.

23. Что такое namespace-ы и зачем они нужны?

В широком смысле - это один из способов инкапсуляции элементов.

В PHP пространства имён используются для решения двух проблем, с которыми сталкиваются авторы библиотек и приложений при создании повторно используемых элементов кода, таких как классы и функции:

- Конфликт имён между вашим кодом и внутренними классами/функциями/константами PHP или сторонними.
- Возможность создавать псевдонимы (или сокращения) для Ну_Очень_Длинных_Имён, чтобы облегчить первую проблему и улучшить читаемость исходного кода.

Пространства имён в PHP предоставляют возможность группировать логически связанные классы, интерфейсы, функции и константы.

24. Сравнение значений переменных в PHP и подводные камни? Приведение типов. Что изменилось в PHP 8 в этом контексте?

https://nooby-games.ru/php-тонкости-работы-с-типами-данных/

Когда Вы конвертируете что-то в тип boolean, значения типа пустой строки или пустого массива, а также число 0 (будь то 0 или 0.00) рассматривается как FALSE.

До PHP 7, если в восьмеричном числе встречаются неразрешенные цифры (например 8 и 9), остальная часть числа начиная с этих цифр игнорируется! Конечно же, без каких-либо предупреждений. С PHP 7 интерпретатор выдает ошибку парсинга.

Если вы попытаетесь записать в переменную с типом integer число большее, чем верхняя числовая граница для этого типа, то это переменная автоматически приводится к типу float. Не такой уж и страшный косяк, но следует иметь это ввиду учитывая особенности типа float (например, сравнение переменных типа float и т.п., об этом подробнее написано ниже).

Если float выходит за границы integer (обычно +/- 2.15e+9 = 2^31 на 32-битных платформах и +/-9.22e+18 = 2^63 на 64-битных платформах, отличных от Windows), **результат не определен, так как float не имеет достаточной точности, чтобы дать точный целочисленный результат.** Когда это произойдет не будет выдано никакое предупреждение!

Но стоит отметить: начиная с PHP 7.0.0, вместо того, чтобы быть неопределенным и зависящим от платформы результатом, NaN и Infinity всегда будут равны нулю при приведении к целому числу.

Начальный ноль в числовом литерале означает: «это восьмеричное число». Но **ведущий ноль в строке не делает число, полученное при приведении данной строки к типу integer восьмеричным.**

Приведение строки к типу integer работает только если строка начинается с цифр, и работает до первого не числового символа

25. Как работает session в PHP, где хранится, как инициализируется?

Сессии являются простым способом хранения информации для отдельных пользователей с уникальным идентификатором сессии. Это может использоваться для сохранения состояния между запросами страниц. Идентификаторы сессий обычно отправляются браузеру через сессионный cookie и используются для получения имеющихся данных сессии. Отсутствие идентификатора сессии или сессионного cookie сообщает PHP о том, что необходимо создать новую сессию и сгенерировать новый идентификатор сессии.

Сессии используют простую технологию. Когда сессия создана, PHP будет либо получать существующую сессию, используя переданный идентификатор (обычно из сессионного cookie) или, если ничего не передавалось, будет создана новая сессия. PHP заполнит суперглобальную переменную $_SESSION сессионной информацией после того, как будет запущена сессия. Когда PHP завершает работу, он автоматически сериализует содержимое суперглобальной переменной $_SESSION и отправит для сохранения, используя сессионный обработчик для записи сессии.

По умолчанию PHP использует внутренний обработчик files для сохранения сессий, который установлен в INI-переменной session.save_handler. Этот обработчик сохраняет данные на сервере в директории, указанной в конфигурационной директиве session.save_path.

Сессии могут запускаться вручную с помощью функции session_start(). Если директива session.auto_start установлена в 1, сессия автоматически запустится, в начале запроса.

Сессия обычно завершает свою работу, когда PHP заканчивает исполнять скрипт, но может быть завершена и вручную с помощью функции session_write_close().

26. Суперглобальные массивы. Какие знаете? Как использовали?

Суперглобальными массивами (англ. Superglobal arrays) в PHP называются предопределённые массивы, имеющие глобальную область видимости без использования директивы global. Большая часть этих массивов содержит входные данные запроса пользователя (параметры GET-запроса, поля форм при посылке методом POST, куки и т. п.).

Все суперглобальные массивы, кроме $GLOBALS и $_REQUEST, имеют устаревшие аналоги с длинными именами, которые доступны вплоть до версии 5.3.x (начиная с 5.4.0 были удалены). Таким образом, обращения $_GET['year'] и $HTTP_GET_VARS['year'] идентичны (за исключением области видимости: массивы с «длинными» именами не являются суперглобальными).

$GLOBALS

Массив всех глобальных переменных (в том числе и пользовательских).

$_SERVER (аналог для устаревшего — $HTTP_SERVER_VARS)

Содержит переменные окружения, которые операционная система передаёт серверу.

$_ENV (уст. $HTTP_ENV_VARS)

Текущие переменные среды (англ. Environment variables). Их набор специфичен для платформы, на которой выполняется скрипт.

$_GET (уст. $HTTP_GET_VARS)

Содержит параметры GET-запроса, переданные в URI после знака вопроса «?».

$_POST (уст. $HTTP_POST_VARS)

Ассоциативный массив значений полей HTML-формы при отправке методом POST. Индексы элементов соответствуют значению свойства name объектов(кнопки, формы, радио-кнопки, флажки и т. д.) HTML-формы.

$_FILES (уст. $HTTP_POST_FILES)

Ассоциативный массив со сведениями об отправленных методом POST файлах. Каждый элемент имеет индекс, идентичный значению атрибута «name» в форме, и, в свою очередь, также является массивом со следующими элементами:

['name'] — исходное имя файла на компьютере пользователя.

['type'] — указанный агентом пользователя MIME-тип файла. PHP не проверяет его, и поэтому нет никаких гарантий, что указанный тип соответствует действительности.

['size'] — размер файла в байтах.

['tmp_name'] — полный путь к файлу во временной папке. Файл необходимо переместить оттуда функцией move_uploaded_file. Загруженные файлы из временной папки PHP удаляет самостоятельно.

['error'] — код ошибки. Если файл удачно загрузился, то этот элемент будет равен 0 (UPLOAD_ERR_OK).

$_COOKIE (уст. $HTTP_COOKIE_VARS)

Ассоциативный массив с переданными агентом пользователя значениями куки.

$_REQUEST

Содержит элементы из массивов $_GET, $_POST, $_COOKIE. С версии PHP 4.1 включает $_FILES.

$_SESSION (уст. $HTTP_SESSION_VARS)

Содержит данные сессии.

27. Сравните include vs required, include_once vs required_once.

Выражение include включает и выполняет указанный файл.

require аналогично include, за исключением того, что в случае возникновения ошибки он также выдаст фатальную ошибку уровня **E_COMPILE_ERROR**. Другими словами, он остановит выполнение скрипта, тогда как include только выдал бы предупреждение **E_WARNING**, которое позволило бы скрипту продолжить выполнение.

Выражение include_once включает и выполняет указанный файл во время выполнения скрипта. Его поведение идентично выражению include, с той лишь разницей, что если

код из файла уже один раз был включён, он не будет включён и выполнен повторно и вернёт **true**. Как видно из имени, он включит файл только один раз.

include_once может использоваться в тех случаях, когда один и тот же файл может быть включён и выполнен более одного раза во время выполнения скрипта, в данном случае это поможет избежать проблем с переопределением функций, переменных и т.д.

Выражение require_once аналогично require за исключением того, что PHP проверит, включался ли уже данный файл, и если да, не будет включать его ещё раз.

28. Что означает сложность алгоритма?

https://techrocks.ru/2021/04/02/big-o-notation-examples/

Анализ временной сложности помогает нам определять, насколько больше времени потребуется нашему алгоритму для решения более объемной задачи.

Нотация «O» большое описывает, как возрастает предположительное время работы алгоритма по мере увеличения размера решаемой задачи.

29. Что такое замыкание в PHP? Приведите пример.

Анонимные функции, также известные как замыкания (closures), позволяют создавать функции, не имеющие определённых имён. Они наиболее полезны в качестве значений callable-параметров, но также могут иметь и множество других применений.

Анонимные функции реализуются с использованием класса Closure.

Замыкания также могут быть использованы в качестве значений переменных; PHP автоматически преобразует такие выражения в экземпляры внутреннего класса Closure. Присвоение замыкания переменной использует тот же синтаксис, что и для любого другого присвоения, включая завершающую точку с запятой

30. В чем разница между замыканием в PHP и JavaScript?

В JavaScript замыкание можно рассматривать как область видимости: когда вы определяете функцию, она молча наследует область, в которой она определена, которая называется замыканием, и сохраняет ее независимо от того, где она используется. Несколько функций могут использовать одно и то же замыкание, и они могут иметь доступ к нескольким замыканиям, если они находятся в пределах доступной области.

В PHP замыкание — это вызываемый класс, к которому вы вручную привязываете свои параметры.

Одно из отличий заключается в том, как оба они справляются с хранением контекста, в котором выполняется анонимная функция:

```
// JavaScript:
var a = 1;
var f = function() {
   console.log(a);
};
a = 2;
f();
// will echo 2;
```

```
// PHP
$a = 1;
$f = function() {
    echo $a;
};
$a = 2;
$f();
// will result in a "PHP Notice:  Undefined variable: a in Untitled.php
on line 5"
```

Чтобы исправить это уведомление, вам придется использовать use синтаксис:

```
$a = 1;
$f = function() use ($a) {
    echo $a;
};
$a = 2;
$f();
// but this will echo 1 instead of 2 (like JavaScript)
```

Чтобы анонимная функция вела себя как-то похоже на аналог JavaScript, вам придется использовать ссылки:

```
$a = 1;
$f = function() use (&$a) {
    echo $a;
};
$a = 2;
$f();
// will echo 2
```

31. Что такое позднее связывание? Расскажите о поведении и применения static.

PHP реализует функцию, называемую позднее статическое связывание, которая может быть использована для того, чтобы получить ссылку на вызываемый класс в контексте статического наследования.

Если говорить более точно, позднее статическое связывание сохраняет имя класса указанного в последнем "неперенаправленном вызове". В случае статических вызовов это явно указанный класс (обычно слева от оператора [::]); в случае не статических вызовов это класс объекта. "Перенаправленный вызов" - это статический вызов, начинающийся с self::, parent::, static::, или, если двигаться вверх по иерархии классов, forward_static_call(). Функция get_called_class() может быть использована для получения строки с именем вызванного класса, а static:: представляет её область действия.

Само название "позднее статическое связывание" отражает в себе внутреннюю реализацию этой особенности. "Позднее связывание" отражает тот факт, что обращения через static:: не будут вычисляться по отношению к классу, в котором вызываемый метод определён, а будут вычисляться на основе информации в ходе исполнения. Также эта особенность была названа "статическое связывание" потому, что она может быть использована (но не обязательно) в статических методах.

Статические ссылки на текущий класс, такие как self:: или __CLASS__, вычисляются используя класс, к которому эта функция принадлежит, как и в том месте, где она была определена

Позднее статическое связывание пытается устранить это ограничение, предоставляя ключевое слово, которое ссылается на класс, вызванный непосредственно в ходе выполнения. Попросту говоря, ключевое слово, которое позволит вам ссылаться на B из test() в предыдущем примере. Было решено не вводить новое ключевое слово, а использовать static, которое уже зарезервировано.

32. Как переопределить хранение сессий?

то бы взять под контроль функционал сессий в php, нужно написать свои обработчики событий для сессии, т.е. определить обработчики следующих событий:

- Открытия сессии (происходит при вызове session_start())
- Закрытия сессии (когда мы уходим со страницы)
- Чтения данных сессии (происходит после вызова session_start())
- Записи данных сессии (происходит когда мы что то пишем в сессию)
- Уничтожения сессии (происходит при вызове session_destroy())
- Сборщик мусора (происходит время от времени)

Для обработки этих событий потребуются создать **шесть callback функций**, и передать их имена в функцию session_set_save_handler().

33. Расскажите о SPL-библиотеке (Reflection, autoload, структуры данных).

Стандартная библиотека PHP (SPL) - это набор интерфейсов и классов, предназначенных для решения стандартных задач.

SPL предоставляет ряд стандартных структур данных, итераторов для оббегания объектов, интерфейсов, стандартных исключений, некоторое количество классов для работы с файлами и предоставляет ряд функций, например spl_autoload_register().

- Предопределённые константы
- Структуры данных
    - SplDoublyLinkedList — Класс SplDoublyLinkedList
    - SplStack — Класс SplStack
    - SplQueue — Класс SplQueue
    - SplHeap — Класс SplHeap
    - SplMaxHeap — Класс SplMaxHeap
    - SplMinHeap — Класс SplMinHeap
    - SplPriorityQueue — Класс SplPriorityQueue
    - SplFixedArray — Класс SplFixedArray
    - SplObjectStorage — Класс SplObjectStorage
- Итераторы
    - AppendIterator — Класс AppendIterator
    - ArrayIterator — Класс ArrayIterator
    - CachingIterator — Класс CachingIterator
    - CallbackFilterIterator — Класс CallbackFilterIterator
    - DirectoryIterator — Класс DirectoryIterator
    - EmptyIterator — Класс EmptyIterator
    - FilesystemIterator — Класс FilesystemIterator
    - FilterIterator — Класс FilterIterator
    - GlobIterator — Класс GlobIterator
    - InfiniteIterator — Класс InfiniteIterator
    - IteratorIterator — Класс IteratorIterator
    - LimitIterator — Класс LimitIterator
    - MultipleIterator — Класс MultipleIterator
    - NoRewindIterator — Класс NoRewindIterator
    - ParentIterator — Класс ParentIterator
    - RecursiveArrayIterator — Класс RecursiveArrayIterator
    - RecursiveCachingIterator — Класс RecursiveCachingIterator
    - RecursiveCallbackFilterIterator — Класс RecursiveCallbackFilterIterator
    - RecursiveDirectoryIterator — Класс RecursiveDirectoryIterator
    - RecursiveFilterIterator — Класс RecursiveFilterIterator
    - RecursiveIteratorIterator — Класс RecursiveIteratorIterator
    - RecursiveRegexIterator — Класс RecursiveRegexIterator
    - RecursiveTreeIterator — Класс RecursiveTreeIterator
    - RegexIterator — Класс RegexIterator
- Интерфейсы
    - Countable — Интерфейс Countable
    - OuterIterator — Интерфейс OuterIterator
    - RecursiveIterator — Интерфейс RecursiveIterator
    - SeekableIterator — Интерфейс SeekableIterator

- Исключения
  - BadFunctionCallException — Класс BadFunctionCallException
  - BadMethodCallException — Класс BadMethodCallException
  - DomainException — Класс DomainException
  - InvalidArgumentException — Класс InvalidArgumentException
  - LengthException — Класс LengthException
  - LogicException — Класс LogicException
  - OutOfBoundsException — Класс OutOfBoundsException
  - OutOfRangeException — Класс OutOfRangeException
  - OverflowException — Класс OverflowException
  - RangeException — Класс RangeException
  - RuntimeException — Класс RuntimeException
  - UnderflowException — Класс UnderflowException
  - UnexpectedValueException — Класс UnexpectedValueException
- Функции SPL
  - class_implements — Возвращает список интерфейсов, реализованных в заданном классе или интерфейсе
  - class_parents — Возвращает список родительских классов заданного класса
  - class_uses — Возвращает список трейтов, используемых заданным классом
  - iterator_apply — Вызывает функцию для каждого элемента в итераторе
  - iterator_count — Подсчитывает количество элементов в итераторе
  - iterator_to_array — Копирует итератор в массив
  - spl_autoload_call — Попытка загрузить класс всеми зарегистрированными функциями __autoload()
  - spl_autoload_extensions — Регистрация и вывод расширений файлов для spl_autoload
  - spl_autoload_functions — Получение списка всех зарегистрированных функций __autoload()
  - spl_autoload_register — Регистрирует заданную функцию в качестве реализации метода __autoload()
  - spl_autoload_unregister — Отмена регистрации функции в качестве реализации метода __autoload()
  - spl_autoload — Реализация по умолчанию метода __autoload()
  - spl_classes — Возвращает доступные классы SPL
  - spl_object_hash — Возвращает хеш-идентификатор для объекта
  - spl_object_id — Получить целочисленный идентификатор объекта
- Обработка файлов
  - SplFileInfo — Класс SplFileInfo
  - SplFileObject — Класс SplFileObject
  - SplTempFileObject — Класс SplTempFileObject
- Различные классы и интерфейсы
  - ArrayObject — Класс ArrayObject
  - SplObserver — Интерфейс SplObserver
  - SplSubject — Интерфейс SplSubject

34. Расскажите о принципах SOLID.

https://techrocks.ru/2020/08/26/solid-principles-in-plain-russian/

- **S**ingle Responsibility Principle («Принцип единой ответственности», SRP)
- **O**pen-Closed Principle («Принцип открытости-закрытости», OCP)
- **L**iskov Substitution Principle («Принцип подстановки Барбары Лисков», LSP)
- **I**nterface Segregation Principle («Принцип разделения интерфейса», ISP)
- **D**ependency Inversion Principle («Принцип инверсии зависимостей», DIP)

Принцип единственной ответственности гласит, что класс должен делать какое-то одно действие и, соответственно, для его изменения должна быть только одна причина.

Принцип открытости-закрытости требует, чтобы классы были открыты для расширения, но закрыты для модификации.

Принцип подстановки Барбары Лисков гласит, что подклассы должны заменять свои базовые классы. Если у нас есть класс B, являющийся подклассом класса A, у нас должна быть возможность передать объект класса B любому методу, который ожидает объект класса A, причем этот метод не должен выдать в таком случае какой-то странный output.

Принцип разделения интерфейса: много клиентоориентированных интерфейсов лучше, чем один интерфейс общего назначения. Клиенты не должны принуждаться к реализации функций, которые им не нужны.

Принцип инверсии зависимостей гласит, что наши классы должны зависеть от интерфейсов или абстрактных классов, а не от конкретных классов и функций.

35. Расскажите о шаблонах GRASP.

GRASP (General Responsibility Assignment Software Patterns) — шаблоны проектирования, используемые для решения общих задач по назначению обязанностей классам и объектам.

Известно девять GRAPS шаблонов, изначально описанных в книге Крейга Лармана «Применение UML и шаблонов проектирования». В отличие от привычных читателю паттернов из Банды Четырех, GRAPS паттерны не имеют выраженной структуры, четкой области применения и конкретной решаемой проблемы, а лишь представляют собой обобщенные подходы/рекомендации/принципы, используемые при проектировании дизайна системы.

1. Информационный эксперт (Information Expert)

Шаблон определяет базовый принцип распределения ответственностей

- Повышает:
  - Инкапсуляцию;
  - Простоту восприятия;

    - Готовность компонентов к повторному использованию;
- Снижает:
    - степень зацепления.

2. Создатель (Creator) (*Абстрактная фабрика (шаблон проектирования)*)

Класс должен создавать экземпляры тех классов, которые он может:

- Содержать или агрегировать;
- Записывать;
- Использовать;
- Инициализировать, имея нужные данные.

3. Контроллер (Controller)

- Отвечает за операции, запросы на которые приходят от пользователя, и может выполнять сценарии одного или нескольких вариантов использования (например, создание и удаление);
- Не выполняет работу самостоятельно, а делегирует компетентным исполнителям;
- Может представлять собой:
    - Систему в целом;
    - Подсистему;
    - Корневой объект;
    - Устройство.

4. Слабое зацепление (Low Coupling)

«**Степень зацепления»** — мера *неотрывности* элемента от других элементов (либо мера данных, имеющихся у него о них).

«**Слабое» зацепление** является оценочной моделью, которая диктует, как распределить обязанности, которые необходимо поддерживать.

«**Слабое» зацепление** — распределение ответственностей и данных, обеспечивающее взаимную независимость классов. Класс со «слабым» зацеплением:

- Имеет слабую зависимость от других классов;
- Не зависит от внешних изменений (изменение в одном классе оказывает слабое влияние на другие классы);
- Прост для повторного использования.

5. Высокая связность (High Cohesion)

Высокая связность класса — это оценочная модель, направленная на удержание объектов должным образом сфокусированными, управляемыми и понятными. Высокая связность обычно используется для поддержания низкого зацепления. Высокая связность означает, что обязанности данного элемента тесно связаны и сфокусированы. Разбиение программ на классы и подсистемы является примером деятельности, которая увеличивает связность системы.

И наоборот, низкая связность — это ситуация, при которой данный элемент имеет слишком много несвязанных обязанностей. Элементы с низкой связностью часто страдают от того, что их трудно понять, трудно использовать, трудно поддерживать.

Связность класса — мера сфокусированности предметных областей его методов:

- «**Высокая»** связность — *сфокусированные* подсистемы (предметная область определена, управляема и понятна);
- «**Низкая»** связность — *абстрактные* подсистемы, затруднены:
    - Восприятие;
    - Повторное использование;
    - Поддержка;
    - Устойчивость к внешним изменениям.

6. Полиморфизм (Polymorphism)

Полиморфизм в языках программирования и теории типов — способность функции обрабатывать данные разных типов

Устройство и поведение системы:

- Определяется данными;
- Задано полиморфными операциями её интерфейса.

Пример: Адаптация коммерческой системы к *многообразию* систем учёта налогов может быть обеспечена через внешний интерфейс объектов-адаптеров (см. также: Шаблон «Адаптеры»).

7. Чистое изготовление (Pure Fabrication)

Не относится к предметной области, но:

- Уменьшает зацепление;
- Повышает связность;
- Упрощает повторное использование.

«*Pure Fabrication*» отражает концепцию сервисов в модели предметно-ориентированного проектирования.

Пример задачи: Не используя средства класса «А», внести его объекты в базу данных.

Решение: Создать класс «Б» для записи объектов класса «А» (см. также: ***«Data Access Object»***).

8. Перенаправление (Indirection) (*Посредник (шаблон проектирования)*)

Слабое зацепление между элементами системы (и возможность повторного использования) обеспечивается назначением промежуточного объекта их **посредником**.

Пример: В архитектуре Model-View-Controller, *контроллер (англ. controller)* ослабляет зацепление данных *(англ. model)* с их представлением *(англ. view)*.

9. Устойчивость к изменениям (Protected Variations)

Шаблон защищает элементы от изменения другими элементами (объектами или подсистемами) с помощью вынесения взаимодействия в фиксированный интерфейс, через который (и только через который) возможно взаимодействие между элементами. Поведение может варьироваться лишь через создание другой реализации интерфейса.

36. Расскажите о Dependency Injection: что такое DI-контейнеры? Какие есть варианты реализаций?

Внедрение зависимостей — это стиль настройки объекта, при котором поля объекта задаются внешней сущностью. Другими словами, объекты настраиваются внешними объектами. DI — это альтернатива самонастройке объектов.

Dependency Injection Container (DI Container) или контейнер внедрения зависимостей - это паттерн проектирования, смысл которого в том, чтобы разрешать все зависимости, существующие у объекта при его создании.

Например, для создания объекта профайлинга нужно создать объект настроек и передать его в конструктор.

```
class Profile {
    public function deactivateProfile(Setting $setting)
    {
        $setting->isActive = false;
    }
}
```

Поскольку нам нужно использовать объект $setting внутри функции, мы передаем его как параметр.

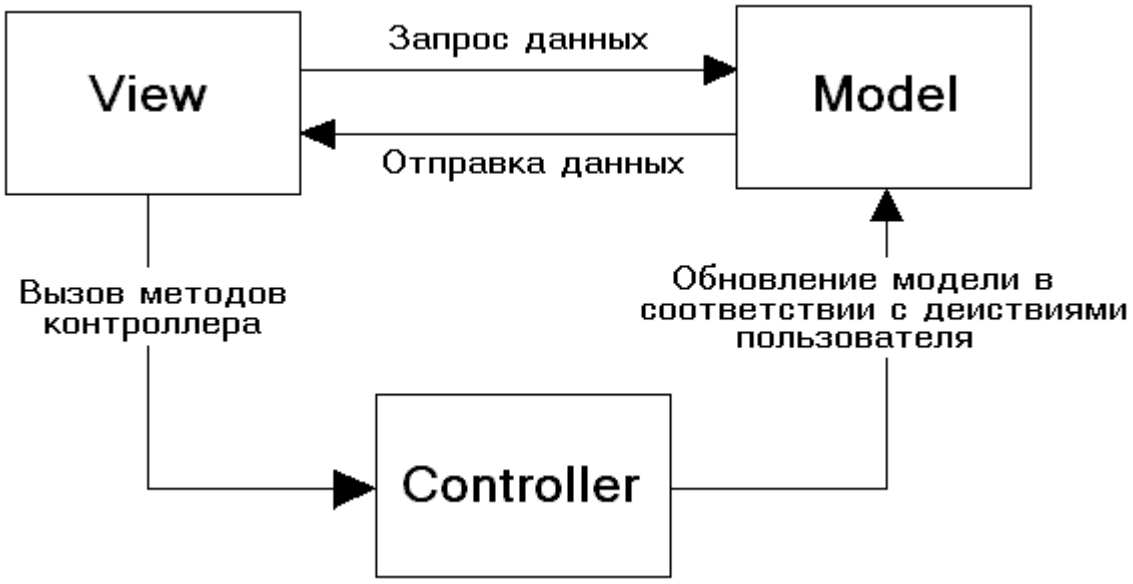
Передача объекта в качестве параметра для метода называется внедрение зависимости или Dependency Injection.

Существует много способов ввода объектов, вот несколько наиболее известных:

Инъекция объекта через конструктор Внедрение зависимости через метод setter
Использование DI Container

37. Что вам известно о MVC?

Шаблон MVC описывает простой способ построения структуры приложения, целью которого является отделение бизнес-логики от пользовательского интерфейса. В результате, приложение легче масштабируется, тестируется, сопровождается и конечно же реализуется.



Типичную последовательность работы MVC-приложения можно описать следующим образом:

- При заходе пользователя на веб-ресурс, скрипт инициализации создает экземпляр приложения и запускает его на выполнение.При этом отображается вид, скажем главной страницы сайта.
- Приложение получает запрос от пользователя и определяет запрошенные контроллер и действие. В случае главной страницы, выполняется действие по умолчанию (*index*).
- Приложение создает экземпляр контроллера и запускает метод действия,в котором, к примеру, содержаться вызовы модели, считывающие информацию из базы данных.
- После этого, действие формирует представление с данными, полученными из модели и выводит результат пользователю.

Модель — содержит бизнес-логику приложения и включает методы выборки (это могут быть методы ORM), обработки (например, правила валидации) и предоставления конкретных данных, что зачастую делает ее очень толстой, что вполне нормально.Модель не должна напрямую взаимодействовать с пользователем. Все переменные, относящиеся к запросу пользователя должны обрабатываться в контроллере.Модель не должна генерировать HTML или другой код отображения, который может изменяться в зависимости от нужд пользователя. Такой код должен обрабатываться в видах.Одна и та же модель, например: модель аутентификации пользователей может использоваться как в пользовательской, так и в

административной части приложения. В таком случае можно вынести общий код в отдельный класс и наследоваться от него, определяя в наследниках специфичные для подприложений методы.

Вид — используется для задания внешнего отображения данных, полученных из контроллера и модели.Виды содержат HTML-разметку и небольшие вставки PHP-кода для обхода, форматирования и отображения данных.Не должны напрямую обращаться к базе данных. Этим должны заниматься модели.Не должны работать с данными, полученными из запроса пользователя. Эту задачу должен выполнять контроллер.Может напрямую обращаться к свойствам и методам контроллера или моделей, для получения готовых к выводу данных.Виды обычно разделяют на общий шаблон, содержащий разметку, общую для всех страниц (например, шапку и подвал) и части шаблона, которые используют для отображения данных выводимых из модели или отображения форм ввода данных.

Контроллер — связующее звено, соединяющее модели, виды и другие компоненты в рабочее приложение. Контроллер отвечает за обработку запросов пользователя. Контроллер не должен содержать SQL-запросов. Их лучше держать в моделях. Контроллер не должен содержать HTML и другой разметки. Её стоит выносить в виды.В хорошо спроектированном MVC-приложении контроллеры обычно очень тонкие и содержат только несколько десятков строк кода. Чего, не скажешь о Stupid Fat Controllers (SFC) в CMS Joomla. Логика контроллера довольно типична и большая ее часть выносится в базовые классы.Модели, наоборот, очень толстые и содержат большую часть кода, связанную с обработкой данных, т.к. структура данных и бизнес-логика, содержащаяся в них, обычно довольно специфична для конкретного приложения.

38. Что вам известно о шаблонах GoF?

Паттернами проектирования (Design Patterns) называют решения часто встречающихся проблем в области разработки программного обеспечения. В данном случае предполагается, что есть некоторый набор общих формализованных проблем, которые довольно часто встречаются, и паттерны предоставляют ряд принципов для решения этих проблем.

Идея показалась привлекательной авторам Эриху Гамму, Ричарду Хелму, Ральфу Джонсону и Джону Влиссидесу, их принято называть «бандой четырёх» (Gang of Four).

Порождающие паттерны:

Порождающие паттерны — это паттерны, которые абстрагируют процесс инстанцирования или, иными словами, процесс порождения классов и объектов. Среди них выделяются следующие:

Абстрактная фабрика (Abstract Factory)

Строитель (Builder)

Фабричный метод (Factory Method)

Прототип (Prototype)

Одиночка (Singleton)

Структурные паттерны:

Структурные паттерны - рассматривает, как классы и объекты образуют более крупные структуры - более сложные по характеру классы и объекты. К таким шаблонам относятся:

Адаптер (Adapter)

Мост (Bridge)

Компоновщик (Composite)

Декоратор (Decorator)

Фасад (Facade)

Приспособленец (Flyweight)

Заместитель (Proxy)

Поведенческие паттерны:

Поведенческие паттерны - они определяют алгоритмы и взаимодействие между классами и объектами, то есть их поведение. Среди подобных шаблонов можно выделить следующие:

Цепочка обязанностей (Chain of responsibility)

Команда (Command)

Интерпретатор (Interpreter)

Итератор (Iterator)

Посредник (Mediator)

Хранитель (Memento)

Наблюдатель (Observer)

Состояние (State)

Стратегия (Strategy)

Шаблонный метод (Template method)

Посетитель (Visitor)

39. Что вам известно о шаблонах, которые применяются в ORM?

https://jeka.by/post/1094/dao-vs-active-record-vs-data-mapper/

DAO (Data Access Object), DataMapper (Doctrine) и ActiveRecord (Eloquent)

ORM или Object-Relational Mapping (объектно-реляционное отображение) — технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования.

- Doctrine
- Tryton
- ActiveRecord
- EclipseLink
- Hibernate
- Entity Framework

Active record (**AR**) — шаблон проектирования приложений, описанный Мартином Фаулером в книге *Patterns of Enterprise Application Architecture* («Шаблоны архитектуры корпоративных приложений»). AR является популярным способом доступа к данным реляционных баз данных в объектно-ориентированном программировании.

Схема Active Record — это подход к доступу к данным в базе данных. Таблица базы данных или представление обёрнуты в классы. Таким образом, объектный экземпляр привязан к единственной строке в таблице. После создания объекта новая строка будет добавляться к таблице на сохранение. Любой загруженный объект получает свою информацию от базы данных. Когда объект обновлён, соответствующая строка в таблице также будет обновлена. Класс обёртки реализует методы средства доступа или свойства для каждого столбца в таблице или представлении.

Этот образец обычно используется объектными инструментами персистентности и в объектно-реляционном отображении (ORM). Как правило, отношения внешнего ключа будут представлены как объектный экземпляр надлежащего типа через свойство.

Реализации данного шаблона часто нарушают принцип единственной ответственности (SRP), совмещая в одном объекте как представление и внутреннюю логику самого объекта, так и механизмы CRUD, поэтому Active Record может считаться антипаттерном[1]. В других случаях это утверждение спорно, так как сам по себе объект, реализующий ActiveRecord, не содержащий никакой бизнес-логики, а предоставляющий таблицу из базы данных, имеет лишь одну причину для изменения (изменение таблицы), что не противоречит определением принципа SRP[2].

Общий принцип работы Active Record

Пусть существует таблица в базе данных. Для данной таблицы создаётся специальный класс AR, являющийся отражением (представлением) таблицы, таким образом, что:

- каждый экземпляр данного класса соответствует одной записи таблицы;
- при создании нового экземпляра класса (и заполнении соответствующих полей) в таблицу добавляется новая запись;
- при чтении полей объекта считываются соответствующие значения записи таблицы баз данных;
- при изменении (удалении) какого-либо объекта изменяется (удаляется) соответствующая ему запись.

Как правило, Active Record Модель - это маппинг полей модели на поля в базе данных. В Active record сама модель отвечает за сохранение данных в базу данных. А это означает, что нарушется первый принцип из **S**OLID - принцип единственности ответственности. Класс отвечает не только за представление данных, но и за сохранение.

Doctrine — объектно-реляционный проектор (ORM) для PHP 7.1+, который базируется на слое абстракции доступа к БД (DBAL). Одной из ключевых возможностей Doctrine является запись запросов к БД на собственном объектно-ориентированном диалекте SQL, называемом DQL (Doctrine Query Language) и базирующемся на идеях HQL (Hibernate Query Language).

Data Mapper - это слой доступа к данным, который предоставляет двунаправленный маппинг данных между постоянным хранилищем данных (обычно, это sql база данных) и хранением данных в памяти (например, на время выполнениния php скрипта).

В отличие от Active Record, в Data Mapper появляется еще один слой или тип сущности такой как entityManager. Именно этот слой будет отвечать за перенос состояния модели в базу данных и обратно.

40. Напишите на PHP пример реализации паттерна Singleton.

```
class Singleton
{
    /**
     * The Singleton's instance is stored in a static field. This field
is an
     * array, because we'll allow our Singleton to have subclasses. Each
item in
     * this array will be an instance of a specific Singleton's
subclass. You'll
     * see how this works in a moment.
     */
private static $instances = [];

    /**
     * The Singleton's constructor should always be private to prevent
```

```
direct
     * construction calls with the `new` operator.
     */
protected function __construct() { }

    /**
     * Singletons should not be cloneable.
     */
protected function __clone() { }

    /**
     * Singletons should not be restorable from strings.
     */
public function __wakeup()
    {
throw new \Exception("Cannot unserialize a singleton.");
    }

    /**
     * This is the static method that controls the access to the
singleton
     * instance. On the first run, it creates a singleton object and
places it
     * into the static field. On subsequent runs, it returns the client
existing
     * object stored in the static field.
     *
     * This implementation lets you subclass the Singleton class while
keeping
     * just one instance of each subclass around.
     */
public static function getInstance(): Singleton
    {
        $cls = static::class;
if (!isset(self::$instances[$cls])) {
self::$instances[$cls] =new static();
        }

return self::$instances[$cls];
    }

    /**
     * Finally, any singleton should define some business logic, which
can be
     * executed on its instance.
     */
```

```
public function someBusinessLogic()
    {
        // ...
    }
}
```

41. Что такое Docker? Каков принцип его работы?

Docker — программное обеспечение для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации, контейнеризатор приложений. Позволяет «упаковать» приложение со всем его окружением[en] и зависимостями в контейнер, который может быть развёрнут на любой Linuxсистеме с поддержкой контрольных групп в ядре, а также предоставляет набор команд для управления этими контейнерами. Изначально использовал возможности LXC, с 2015 года начал использовать собственную библиотеку, абстрагирующую виртуализационные возможности ядра Linux — **libcontainer**. С появлением Open Container Initiative начался переход от монолитной к модульной архитектуре.

Программное обеспечение функционирует в среде Linux с ядром, поддерживающим контрольные группы и изоляцию пространств имён (*namespaces*); существуют сборки только для платформ x86-64 и ARM[14]. Начиная с версии 1.6 (апрель 2015 года) возможно использование в операционных системах семейства Windows[15].

Для экономии пространства хранения проект использует файловую систему Aufs с поддержкой технологии каскадно-объединённого монтирования: контейнеры используют образ базовой операционной системы, а изменения записываются в отдельную область. Также поддерживается размещение контейнеров в файловой системе Btrfs с включённым режимом копирования при записи.

В состав программных средств входит демон — сервер контейнеров (запускается командой docker -d), клиентские средства, позволяющие из интерфейса командной строки управлять образами и контейнерами, а также API, позволяющий в стиле REST управлять контейнерами программно.

Демон обеспечивает полную изоляцию запускаемых на узле контейнеров на уровне файловой системы (у каждого контейнера собственная корневая файловая система), на уровне процессов (процессы имеют доступ только к собственной файловой системе контейнера, а ресурсы разделены средствами libcontainer), на уровне сети (каждый контейнер имеет доступ только к привязанному к нему сетевому пространству имён и соответствующим виртуальным сетевым интерфейсам).

Набор клиентских средств позволяет запускать процессы в новых контейнерах (docker run), останавливать и запускать контейнеры (docker stop и docker start), приостанавливать и возобновлять процессы в контейнерах (docker pause и docker unpause). Серия команд позволяет осуществлять мониторинг запущенных процессов (docker ps по аналогии с ps в Unix-системах, docker top по аналогии с top и другие).

Новые образы возможно создавать из специального сценарного файла (docker build, файл сценария носит название Dockerfile), возможно записать все изменения, сделанные в контейнере, в новый образ (docker commit). Все команды могут работать как с docker-демоном локальной системы, так и с любым сервером Docker, доступным по сети. Кроме того, в интерфейсе командной строки встроены возможности по взаимодействию с публичным репозиторием Docker Hub, в котором размещены предварительно собранные образы приложений, например, команда docker search позволяет осуществить поиск образов среди размещённых в нём[16], образы можно скачивать в локальную систему (docker pull), возможно также отправить локально собранные образы в Docker Hub (docker push).

Также Docker имеет пакетный менеджер Docker Compose, позволяющий описывать и запускать многоконтейнерные приложения. Конфигурационные файлы Compose описываются на языке YAML[17].

42. Что такое LAMP / NAMP?

LAMP — акроним, обозначающий набор (комплекс) серверного программного обеспечения, широко используемый во Всемирной паутине. LAMP назван по первым буквам входящих в его состав компонентов:

- Linux — операционная система Linux;
- Apache — веб-сервер;
- MariaDB / MySQL — СУБД;
- PHP — язык программирования, используемый для создания веб-приложений (помимо PHP могут подразумеваться другие языки, такие как Perl и Python).

Существует множество вариантов термина, в частности:

- **LEMP** — Nginx вместо Apache (Nginx читается Engine-X)
    - **LNMP** — другой вариант названия 'Nginx вместо Apache'
- **LLSMP** - Linux, LiteSpeed, MySQL, PHP
- **BAMP** — BSD вместо Linux
- **MAMP** — Mac OS X вместо Linux.
- **SAMP** — Solaris вместо Linux
- **WAMP** — Microsoft Windows вместо Linux
- **WASP** — Windows, Apache, SQL Server и PHP
- **WIMP** — Windows, IIS, MySQL и PHP
- **PAMP** — Personal Apache MySQL PHP — набор серверов для платформы S60. Специфика платформы накладывает свой отпечаток на работу комплекса. Так, в частности, модули PHP получают и возвращают строки только в кодировке UTF-8.
- **FNMP** — FreeBSD и Nginx вместо Linux и Apache.
- **XAMPP** — кроссплатформенная сборка веб-сервера, **X** (любая из четырёх операционных систем), Apache, MySQL, PHP, Perl

43. Расскажите о regexp.

Регулярные выражения (обозначаемые в английском как *RegEx* или как *regex*) являются инструментальным средством, которое применяется для различных вариантов изучения и обработки текста: поиска, проверки, поиска и замены того или иного элемента, состоящего из букв или цифр (или любых других символов, в том числе специальных символов и символов пунктуации). Изначально регулярные выражения пришли в мир программирования из среды научных исследований, которые проводились в 50-е годы в области математики.

Спустя десятилетия принципы и идеи были перенесены в среду операционной системы UNIX (в частности вошли в утилиту grep) и были реализованы в языке программирования Perl, который на заре интернета широко использовался на бэкенде (и по сей день используется, но уже меньше) для такой задачи, как, например, валидация форм.

44. Расскажите о SSH-протоколе.

SSH — сокращение от «secure shell» (безопасная оболочка). Это протокол, который чаще всего используют для управления удалёнными компьютерами по сети.

Есть несколько шагов, которые нужно пройти, чтобы начать SSH-сеанс между компьютерами.

- Сначала нужно обеспечить безопасный способ обмена сообщениями между компьютерами, то есть настроить зашифрованный канал.
- Далее нужно проверить целостность данных, отправляемых клиентом.
- После этого проверяется подлинность клиента.

После этих трёх шагов мы можем безопасно общаться с удалённым компьютером, делиться секретными данными, а также проверить, есть ли у клиента разрешение на доступ к хосту. Каждый из разделов ниже будет более подробно описывать эти действия.

Вся информация, отправляемая с использованием SSH, зашифрована. Обе стороны должны знать и понимать способ шифрования.

Для шифрования передаваемых данных используется симметричное шифрование. Суть данного подхода заключается в том, что оба компьютера имеют одинаковый ключ шифрования, который называется «симметричный ключ». Симметричное шифрование работает очень хорошо, но только до тех пор, пока сторонние не имеют доступа к ключу.

Один компьютер может создать ключ и отправить в виде сообщения через интернет. Но сообщение ещё не будет зашифровано, поэтому любой, кто перехватит его, сразу же сможет расшифровать все следующие сообщения.

Решение этой проблемы состоит в использовании протокола обмена ключами Диффи-Хеллмана. Оба компьютера создают свой закрытый и открытый ключ. Вместе

они образуют пару ключей. Компьютеры делятся своими открытыми ключами друг с другом через интернет. Используя свой закрытый и чужой открытый ключ, стороны могут независимо сгенерировать одинаковый симметричный ключ.

45. Верификация

Следующий этап процесса установки сеанса SSH заключается в проверке того, что данные не были подделаны во время их передачи и что другой компьютер действительно является тем, за кого себя выдаёт.

Для верификации используют хеш-функцию. Это математическая функция, которая принимает входные данные и создаёт строку фиксированного размера.

Важной особенностью этой функции является то, что практически невозможно определить входные данные, зная лишь результат её работы.

После того как клиент и хост сгенерировали свои симметричные ключи, клиент использует хеш-функцию для генерации HMAC, что означает «код аутентификации сообщений, использующий хеширование». Клиент отправит этот HMAC на сервер для верификации.

Функция хеширования использует:

- симметричный ключ клиента,
- порядковый номер пакета,
- содержимое сообщения (зашифрованное).

Когда хост получает HMAC, он может использовать ту же самую хеш-функцию с этими тремя компонентами:

- собственный (идентичный клиентскому) симметричный ключ;
- порядковый номер пакета;
- зашифрованное сообщение.

Если сформированный хеш совпадает с HMAC, полученным от клиента, то мы можем быть уверены, что подключаемый компьютер — это компьютер с симметричным ключом, потому что только хост и клиент знают симметричный ключ, а другие компьютеры — нет.

Прелесть этого подхода в том, что мы не просто проверили личность клиента и убедились, что данные не были подделаны, но мы сделали это без передачи какой-либо секретной информации.

46. Аутентификация

Даже если мы используем симметричные ключи для безопасного общения, мы не знаем, имеет ли подключающийся компьютер разрешение на доступ к содержимому хоста. Для того чтобы проверить это, необходимо произвести аутентификацию.

Многие используют аутентификацию по паролю. Клиент отправляет хосту зашифрованное сообщение, содержащее пароль. Хост его расшифровывает и ищет пароль в базе данных, чтобы удостовериться, есть ли у клиента разрешение на доступ. Использование пароля для аутентификации допустимо, но имеет свои недостатки, так как необходимо хранить все пароли на сервере.

Более безопасной является аутентификация по сертификату. Сформировав сертификат, клиент единожды вводит пароль для доступа к серверу и отправляет ему открытую часть сертификата. В дальнейшем ввод пароля не требуется. Этот подход считается более безопасным, чем просто использование пароля, поскольку не подразумевает хранение секрета пользователя на хосте.

47. Что такое PDO?

Модуль Объекты данных PHP (PDO) определяет простой и согласованный интерфейс для доступа к базам данных в PHP. Каждый драйвер базы данных, в котором реализован этот интерфейс, может представить специфичную для базы данных функциональность в виде стандартных функций модуля. Но надо заметить, что сам по себе модуль PDO не позволяет манипулировать доступом к базе данных. Чтобы воспользоваться возможностями PDO, необходимо использовать соответствующий конкретной базе данных PDO драйвер.

PDO обеспечивает абстракцию *доступу к данным*, это значит, что вне зависимости от того, какая конкретная база данных используется, вы можете пользоваться одними и теми же функциями для выполнения запросов и выборки данных. PDO *не* абстрагирует саму *базу данных*, этот модуль не переписывает SQL-запросы и не эмулирует отсутствующий в СУБД функционал. Если нужно именно это, необходимо воспользоваться полноценной абстракцией базы данных.

Модуль PDO внедрён в PHP.

48. Что нового появилось в PHP 8?

Наиболее существенным изменением в версии PHP 8 стало использование JIT-компилятора. Он позволяет переводить код программы в машинный код в режиме реального времени непосредственно во время исполнения программы. Таким образом для некоторых случаев удается повысить скорость исполнения приложений. В случае с PHP 8 наибольший прирост скорости с помощью JIT-компилятора достигается при выполнении математических операций.

Еще одним практичным улучшением в PHP 8 стало добавление выражения match. В целом оно работает аналогично традиционному switch, однако, в отличие от последнего, использует строгое сравнение значений. При этом результат работы match может быть сохранен в переменную и использован в дальнейшем или возвращен, например, с помощью функции echo. В отличие от switch, выражение match работает с однострочными выражениями, не требующими конструкции break.

В PHP 8 также появились именованные аргументы для использования в библиотеках или ассоциативных массивах. Это нововведение, в отличие от PHP 7, использует

самодокументируемые аргументы, что позволяет использовать их в любом порядке, пропуская необязательные параметры. Это положительно сказывается на читаемости кода и его объеме, особенно если речь идет об использовании библиотек, работающих с булевыми параметрами.

Еще в PHP 8 появилась возможность использовать структурные метаданные с нативным синтаксисом PHP вместо аннотаций PHPDoc. Благодаря этому, например, при написании метаданных можно будет использовать подсказки среды разработки. Кроме того, в PHP 8 стали возможными одновременное объявление и инициализация свойств в конструкторе класса. В некоторые случаях это позволяет сократить объем кода втрое.

Наконец, еще одним существенным изменением в PHP 8 стало использование нового оператора Nullsafe, который записывается занком ? после обрабатываемого элемента. Это избавляет, например, от необходимости прописывать проверку на null для каждой переменной, используя вместо этого последовательность вызовов с оператором Nullsafe. В этом случае, если хотя бы один элемент возвращает значение null, вся последовательность вернет null.

49. Что такое PHP PEAR?

PEAR (акроним от английских слов PHP Extension and Application Repository) — это библиотека классов PHP с открытым исходным кодом, распространяемых через одноименный пакетный менеджер. В стандартную поставку PHP входит система управления классами PEAR, которая позволяет легко скачивать и обновлять их.

Чтобы класс вошёл в PEAR, он должен соответствовать очень жёстким правилам. Например, без особой необходимости нельзя создавать класс с такой же задачей, как у уже созданного.

В рамках PEAR был создан специальный стиль оформления PHP-кода, которого должны придерживаться все классы в библиотеке. Этот стиль стал наиболее распространённым стандартом стиля PHP-кода в интернете.

На английском языке «pear» означает «груша», которая и является логотипом проекта.

50. Какие версии PHP до сих пор поддерживаются?

В настоящий момент актуальными являются 2 **версии PHP** — 8.1 и 8.2, **версия** 8.0 находится пока в режиме **поддержки**, более старые **версии** уже не **поддерживаются**.

51. В чем разница между GET и POST?

Основное отличие метода GET от POST в способе передачи данных.

Запрос GET передает данные в URL в виде пар "имя-значение" (другими словами, через ссылку), а **запрос POST передает данные в теле запроса** (подробно показано в примерах ниже). Это различие определяет свойства методов и ситуации, подходящие для использования того или иного HTTP метода.

Страница, созданная методом GET, может быть открыта повторно множество раз. Такая страница может быть кэширована браузерами, проиндексирована поисковыми системами и добавлена в закладки пользователем. Из этого следует, что метод GET следует использовать для получения данных от сервера и не желательно в запросах, предполагающих внесений изменений в ресурс.

Например, можно использовать метод GET в HTML форме фильтра товаров: когда нужно, исходя из данных введенных пользователем, переправить его на страницу с отфильтрованными товарами, соответствующими его выбору.

Запрос, выполненный методом POST, напротив следует использовать в случаях, когда нужно вносить изменение в ресурс (выполнить авторизацию, отправить форму оформления заказа, форму обратной связи, форму онлайн заявки). Повторный переход по конечной ссылке не вызовет повторную обработку запроса, так как не будет содержать переданных ранее параметров. Метод POST имеет большую степень защиты данных, чем GET: параметры запроса не видны пользователю без использования специального ПО, что дает методу преимущество при пересылке конфиденциальных данных, например в формах авторизации.

HTTP метод POST поддерживает тип кодирования данных *multipart/form-data*, что позволяет передавать файлы.

Также следует заметить, что методы можно комбинировать. То есть, при необходимости вы можете отправить POST запрос на URL, имеющий GET параметры.

52. Чем отличаются операторы BREAK и CONTINUE?

Оператор break досрочно завершает весь цикл, тогда как **continue** вызывает следующую итерацию

53. Есть ли разница между одинарными и двойными кавычками?

Строку, заключенную в одинарные кавычки, интерпретатор php выводит как есть, заключенную же в двойные кавычки парсит на наличие в ней переменных и, найдя таковые, подставляет их значения.

54. Что такое Cookie и зачем они используются?

Ку́ки (англ. *cookie*, букв. — «печенье») — небольшой фрагмент данных, отправленный веб-сервером и хранимый на компьютере пользователя. Веб-клиент (обычно веб-браузер) всякий раз при попытке открыть страницу соответствующего сайта пересылает этот фрагмент данных веб-серверу в составе HTTP-запроса. Применяется для сохранения данных на стороне пользователя, на практике обычно используется для[1]:

- аутентификации пользователя;
- хранения персональных предпочтений и настроек пользователя;
- отслеживания состояния сеанса доступа пользователя;
- сведения статистики о пользователях.

Поддержки браузерами cookie (приём, сохранение и последующая пересылка серверу сохранённых cookie) требуют многие сайты с ограничениями доступа, большинство интернет-магазинов. Настройка оформления и поведения многих веб-сайтов по индивидуальным предпочтениям пользователя тоже основана на cookie.

Cookie легко перехватить и подменить (например, для получения доступа к учётной записи), если пользователь использует нешифрованное соединение с сервером. В группе риска пользователи, выходящие в интернет при помощи публичных точек доступа Wi-Fi и не использующие при этом таких механизмов, как SSL и TLS. Шифрование позволяет также решить и другие проблемы, связанные с безопасностью передаваемых данных.

55. Что нельзя хранить в Cookie и почему?

- персональная информация
- большие объемы данных

Во-первых, нужно проверить, не хранятся ли пароли от сайтов в cookie. Удивительно, но в 2021 году до сих пор существуют сайты, которые это делают. Информация о сессиях должна иметь ограничения — быть временной или заменяться с каждой новой сессией.

Во-вторых, нужно проверить, чтобы все конфиденциальные cookie были с флагом httpOnly и secure.

Cookie с флагом “secure” передаются на сервер только по протоколу HTTPS. Как правило, в этом случае есть сертификат SSL или TLS. Cookie с флагом “httpOnly” защищены от манипуляции JavaScript через документ, где хранятся cookie.

56. Какую среду разработки предпочитаете и почему?

Phpstorm

57. Git: Какой командой добавить изменения?

add

58. Git: Какой командой зафиксировать изменения?

commit

59. Git: Какой командой отправить изменения в удаленный репозиторий?

push

60. БД: Что такое транзакция?

Транзакция — это группа инструкций одной или нескольких баз данных, которые либо полностью фиксируются, либо полностью откатываются. Транзакции атомарны, согласованы, изолированы и устойчивы (atomic, consistent, isolated, durable — ACID). Если транзакция выполнена успешно, все инструкции в ней фиксируются. Если транзакция завершается ошибкой, то если хотя бы одна инструкция в группе завершается ошибкой, выполняется откат всей группы.

Начало и конец транзакции зависят от параметра AUTOCOMMIT и инструкций BEGIN TRANSACTION, COMMIT и ROLLBACK.

61. БД: Что такое нормализация?

Нормализация представляет процесс разделения данных по отдельным связанным таблицам. Нормализация устраняет избыточность данных (data redundancy) и тем самым избежать нарушения целостности данных при их изменении, то есть избежать аномалий изменения (update anomaly).

62. БД: Что такое денормализация? Для чего она нужна?

Денормализация (англ. *denormalization*) — намеренное приведение структуры базы данных в состояние, не соответствующее критериям нормализации, обычно проводимое с целью ускорения операций чтения из базы за счет добавления избыточных данных.

63. БД: Какие есть типы связей в базе данных?

Связи делятся на:

- Многие ко многим.
- Один ко многим.
    - с обязательной связью;
    - с необязательной связью;
- Один к одному.
    - с обязательной связью;
    - с необязательной связью;

64. БД: Что означает утверждение о том, что СУБД поддерживает контроль ссылочной целостности связей?

Ссылочная целостность – это ограничение базы данных, гарантирующее, что ссылки между данными являются действительно правомерными и неповрежденными. Ссылочная целостность является фундаментальным принципом теории баз данных и проистекает из той идеи, что база данных должна не только сохранять данные, но и активно содействовать обеспечению их качества.

Поддержка ссылочной целостности в базе данных обеспечивает много преимуществ.

- **Улучшенное качество данных.** Очевидным преимуществом является поддержка качества данных, хранимых в базе данных. Ошибки могут по-прежнему существовать, но, по крайней мере, ссылки будут подлинными и неповрежденными.
- **Убыстрение разработки.** Ссылочная целостность объявляется. Это гораздо продуктивнее (на один или два порядка), чем написание специального программного кода.
- **Меньшее число ошибок.** Объявления ссылочной целостности являются гораздо более лаконичными, чем эквивалентный программный код. По существу, такие объявления приводят к повторному использованию проверенного и оттестированного кода общего назначения в сервере баз данных, а не к новой реализации одной и той же логики от случая к случаю.
- **Согласованность между приложениями.** Ссылочная целостность обеспечивает качество данных для нескольких прикладных программ, которые могут обращаться к базе данных.

65. БД: Если используемая вами СУБД не поддерживает каскадные удаления для поддержки ссылочной целостности связей, что можно сделать для достижения аналогичного результата?

запрограммировать обработку всех связей

66. БД: Что такое первичный и внешний ключи?

Первичный ключ: набор определенных признаков, уникальных для каждой записи. Обозначается первичный ключ, как primary key.

Внешний ключ — это одно или несколько полей (атрибутов), которые являются первичными в другой таблице и значение которых заменяется значениями первичного ключа другой таблицы.

67. БД: В чем разница между первичным и уникальным ключами?

Ограничение первичного ключа — это столбец или группа столбцов в таблице, которые однозначно определяют каждую строку в этой таблице. Первичный ключ не может быть дубликатом, то есть одно и то же значение не может появляться в таблице более одного раза. Таблица должна иметь более одного первичного ключа. Первичный ключ может быть определен на уровне столбца или таблицы. Если вы создаете составной первичный ключ, он должен быть определен на уровне таблицы.

Уникальный ключ — это группа из одного или нескольких полей или столбцов таблицы, которые однозначно идентифицируют запись базы данных.

Уникальный ключ такой же, как первичный ключ, но он может принимать одно нулевое значение для столбца таблицы. Он также не может содержать одинаковые значения. На уникальные ограничения ссылается внешний ключ других таблиц.

Вот важные причины для использования первичного ключа:

- Основная цель первичного ключа — идентифицировать каждую запись в таблице базы данных.
- Вы можете использовать первичный ключ, если не разрешаете кому-либо вводить нулевые значения.
- Если вы удалите или обновите запись, то указанное вами действие будет предпринято для обеспечения целостности данных базы данных.
- Выполните операцию ограничения для отклонения операции удаления или обновления для родительской таблицы.
- Данные организуются в последовательности кластеризованного индекса всякий раз, когда вы физически организуете таблицу СУБД.

Вот важные причины для использования уникального ключа: • Цель уникального ключа — убедиться, что информация в столбце для каждой записи таблицы уникальна. • Когда вы позволяете пользователю ввести нулевое значение. • Уникальный ключ используется, потому что он создает некластеризованный индекс по умолчанию. • Уникальный ключ может быть использован, когда вам нужно сохранить нулевые значения в столбце. • Когда одно или несколько полей / столбцов таблицы, однозначно идентифицируют запись в таблице базы данных.

Вот важные особенности первичного ключа: • Первичный ключ реализует целостность объекта таблицы. • Вы можете сохранить только один основной элемент в таблице. • Первичный ключ содержит один или несколько столбцов таблицы. • Столбцы определены как не нулевые.

Вот важные особенности уникального ключа: • Вы можете определить более одного уникального ключа в таблице. • По умолчанию уникальные ключи находятся в некластеризованных уникальных индексах. • Он состоит из одного или нескольких столбцов таблицы. • Столбец таблицы может быть нулевым, но предпочтительным является только один нулевой на столбец. • На ограничение уникальности можно легко ссылаться с помощью ограничения внешнего ключа.

68. БД: Какие есть типы JOIN и чем они отличаются?

Самое простое объединение, которое мы можем сделать, это CROSS JOIN (перекрестное объединение) или «*декартово произведение*».

При этом объединении мы берем каждую строку одной таблицы и соединяем ее с каждой строкой другой таблицы.

FULL OUTER JOIN похоже на CROSS JOIN, но имеет важные отличия. Первое отличие состоит в том, что для FULL OUTER JOIN требуется условие объединения. Это условие определяет, как именно связаны строки разных таблиц и по какому критерию они должны объединяться.

Следующий вид объединения — внутреннее, INNER JOIN. Внутреннее объединение является наиболее используемым. INNER JOIN возвращает только те строки, где соблюдается условие объединения.

Следующие два вида объединений используют модификатор (LEFT или RIGHT), который определяет, какие данные таблицы включаются в результирующий набор.

Примечание: LEFT JOIN и RIGHT JOIN также называются LEFT OUTER JOIN и RIGHT OUTER JOIN (соответственно левое и правое внешнее объединение).

Эти виды объединений используются в запросах, когда мы хотим вернуть все данные из одной таблицы и добавить к ним связанные данные из другой таблицы (*если таковые есть*).

Если связанных данных во второй таблице нет, мы получаем в выводе все данные только из «первичной» таблицы.

То есть, это запрос информации о конкретной вещи с дополнительными сведениями, если они есть.

69. БД: Что такое курсоры в базах данных?

Курсор — это поименованная область памяти, содержащая результирующий набор select запроса. Второе определение — это механизм обработки результирующего набора select запроса.

Курсор — ссылка на контекстную область памяти. В некоторых реализациях языка программирования SQL (Oracle, Microsoft SQL Server) — получаемый при выполнении запроса результирующий набор и связанный с ним указатель текущей записи. Условно, курсор — это виртуальная таблица которая представляет собой альтернативное хранилище данных. При этом курсор, позволяет обращаться к своим данным, как к данным обычного массива.

Используются курсоры в хранимых процедурах.

70. БД: Что такое агрегатные функции SQL? Приведите несколько примеров.

Функции, возвращающие единственное значение для набора строк, называются агрегатными.

- avg(выражение) - арифметическое среднее;
- min(выражение) - минимальное значение выражения;
- max(выражение) - маскимальное значение выражения;
- sum(выражение) - сумма значений выражения;
- count(*) - количество строк в результате запроса;
- count(выражение) - количество значений выражения, не равных NULL.

оператор GROUP BY, который осуществляет группировку по какому либо полю, что иногда является необходимым.

оператор HAVING используется как дополнение к предыдущему. Он необходим для того, чтобы ставить условия для выборки данных при группировке. Если условие выполняется то выделяется группа, если нет — то ничего не произойдет.

71. БД: Что такое миграции?

обновление структуры базы данных от одной версии до другой (обычно более новой).

72. БД: Расскажите о связи один к одному, один ко многим, многие ко многим.

Связь **один к одному** образуется, когда ключевой столбец (идентификатор) присутствует в другой таблице, в которой тоже является ключом либо свойствами столбца задана его уникальность (одно и тоже значение не может повторяться в разных строках).

На практике связь «один к одному» наблюдается не часто. Например, она может возникнуть, когда требуется разделить данных одной таблицы на несколько отдельных таблиц с целью безопасности.

В типе связей **один ко многим** одной записи первой таблицы соответствует несколько записей в другой таблице.

Если нескольким записям из одной таблицы соответствует несколько записей из другой таблицы, то такая связь называется «**многие ко многим**» и организовывается посредством связывающей таблицы.

73. БД: Назовите и объясните три любых агрегирующих метода.

SUM. Возвращает сумму всех, либо только уникальных, значений в выражении. Функция SUM может быть использована только для числовых столбцов. Значения NULL пропускаются.

COUNT. Эта функция возвращает количество элементов, найденных в группе.

MIN. Возвращает минимальное значение выражения.

74. БД: Зачем используют оператор группировки GROUP BY?

Оператор **GROUP BY** определяет, как строки будут группироваться.

75. БД: В чем разница между WHERE и HAVING? Приведите примеры.

Предложение HAVING это как предложение WHERE, только для групп.

76. БД: В чем разница между операторами DISTINCT и GROUP BY?

Выражение **MySQL DISTINCT** используется для выборки уникальных значений из указанных столбцов.

В **MySQL DISTINCT** наследует поведение от **GROUP BY**. Если вы используете выражение **GROUP BY** без агрегатной функции, то оно будет выполнять роль ключевого слова **DISTINCT**.

Единственное отличие между ними заключается в следующем:

- **GROUP BY** сначала сортирует данные, а затем осуществляет группировку;
- Ключевое слово **DISTINCT**не выполняет сортировки.

Если вы используете ключевое слово **DISTINCT** вместе с выражением **ORDER BY**, то получите тот же результат, что и при применении **GROUP BY**.

77. БД: Для чего нужны операторы UNION, INTERSECT, EXCEPT?

Для объединения нескольких запросов используется служебное слово **UNION**.

SQL запрос Union служит для объединения выходных строк каждого запроса в один результирующий набор.

Если используется **параметр ALL**, то сохраняются все дубликаты выходных строк. Если параметр отсутствует, то в результирующем наборе остаются только уникальные строки.

Объединять вместе можно любое число запросов.

Использование оператора UNION требует выполнения нескольких условий:

- количество выходных столбцов каждого из запросов должно быть одинаковым;
- выходные столбцы каждого из запросов должны быть сравнимы между собой по типам данных (в порядке их очередности);
- в итоговом наборе используются имена столбцов, заданные в первом запросе;
- ORDER BY может быть использовано только в конце составного запроса, так как оно применяется к результату объединения.

В языке SQL есть средства для выполнения операций пересечения и разности запросов — предложение INTERSECT (пересечение) и предложение EXCEPT (разность). Эти предложения работают подобно тому, как работает UNION: в результирующий набор попадают только те строки, которые присутствуют в обоих запросах — INTERSECT, или только те строки первого запроса, которые отсутствуют во втором — EXCEPT. Но беда в том, что многие СУБД не поддерживают эти предложения. Но выход есть — использование предиката EXISTS.

Предикат EXISTS принимает значение TRUE (истина), если подзапрос возвращает хоть какое-нибудь количество строк, иначе EXISTS принимает значение FALSE.

Существует также предикат NOT EXISTS, который действует противоположным образом.

Обычно EXISTS используется в зависимых подзапросах (например, IN).

78. БД: Опишите разницу типов данных DATETIME и TIMESTAMP.

DATETIME

Хранит время в виде целого числа вида YYYYMMDDHHMMSS, используя для этого 8 байтов. Это время **не зависит от временной зоны**. Оно всегда отображается при выборке точно так же, как было сохранено, независимо от того какой часовой пояс установлен в MySQL.

TIMESTAMP

Хранит 4-байтное целое число, равное количеству секунд, прошедших с полуночи 1 января 1970 года по усреднённому времени Гринвича (т.е. нулевой часовой пояс, точка отсчёта часовых поясов).

При получении из базы отображается с учётом часового пояса.

Часовой пояс может быть задан в операционной системе, глобальных настройках MySQL или в конкретной сессии. Запомните, что сохраняется всегда количество секунд по UTC (универсальное координированное время, солнечное время на меридиане Гринвича), а не по локальному часовому поясу.

79. БД: Какие вы знаете движки таблиц и чем они отличаются?

InnoDB

В MySQL 8 подсистема хранения данных InnoDB используется по умолчанию и является наиболее широко применяемой из всех других доступных подсистем хранения. Подсистема InnoDB была выпущена вместе с MySQL 5.1 как плагин в 2008 году, и она рассматривается как подсистема хранения по умолчанию, начиная с версии 5.5 и выше. Поддержка подсистемы хранения InnoDB была перенята корпорацией Oracle в октябре 2005 года у финской компании Innobase Oy.

Таблицы InnoDB поддерживают ACID-совместимые фиксации транзакций, откат и возможности аварийного восстановления для защиты пользовательских данных. InnoDB также поддерживает блокировку на уровне строк, что помогает улучшить параллелизм и производительность. InnoDB хранит данные в кластеризованных индексах, чтобы уменьшить операции ввода-вывода для всех запросов SQL на выборку данных на основе первичного ключа. InnoDB также поддерживает ограничения внешнего ключа, которые обеспечивают лучшую целостность данных в базе данных. Максимальный размер таблицы InnoDB может масштабироваться до 256 Тб, что должно быть вполне достаточным во многих случаях использования больших данных.

Важные замечания по InnoDB

Выполнение простого запроса, такого как SELECT count(*) FROM [имя таблицы], без наличия индексов будет очень медленным, поскольку, для того чтобы получить данные, он выполняет полное сканирование таблицы. Если вы хотите часто применять запрос на количество данных к таблице InnoDB, предлагается создавать триггеры на операции вставки и удаления, после чего можно увеличивать или уменьшать счетчики, когда записи вставляются или удаляются, что может помочь вам достигнуть лучшей производительности.

> Дамп содержимого MySQL, который используется для создания резервной копии, работает с InnoDB слишком медленно. Во время выполнения команды mysqldump можно включить флаги --opt--compress, которые фактически сжимают данные, прежде чем делать дамп содержимого вашей базы данных/таблицы MySQL.

InnoDB - это подсистема хранения данных с **мультиверсионным управлением параллелизмом** (MVCC), которая хранит информацию о старых версиях измененных строк, чтобы поддержать функционал транзакций и отката, что оказывается весьма кстати в целях поддержания целостности данных или в случаях аварийного прекращения работы.

Для оптимизации производительности таблицы InnoDB ниже приведено несколько параметров, которые мы можем использовать в настройках my.cnf. Однако они зависят от вашей среды и баз данных.

- innodb_open_files = 300: определяет максимальное количество файлов, которые она может держать открытыми при работе с режимом innodb_file_per_table.
- innodb_buffer_pool_size = 128M: задает размер пула в памяти, который может использоваться для кеширования индексов и табличных данных. Это один из важных аспектов настройки таблиц InnoDB. Это значение можно увеличить в зависимости от размера оперативной памяти на сервере.
- innodb_thread_concurrency = 8: этот параметр используется для нескольких параллельных потоков, которые будут использоваться для обработки запроса и зависят от числа доступных ЦП.

MyISAM

Подсистема хранения данных MyISAM использовалась по умолчанию для MySQL вплоть до версии 5.5 1. В отличие от InnoDB, таблицы подсистемы хранения данных MyISAM не поддерживают ACID-совместитмость. Таблицы MyISAM поддерживают только блокировку уровня таблицы, поэтому таблицы MyISAM небезопасны для транзакций. Таблицы MyISAM оптимизированы для сжатия и скорости. MyISAM обычно используется, когда вам нужно иметь в основном операции чтения с минимальными транзакционными данными. Максимальный размер таблицы My- ISAM может достигать 256 Тб, что помогает в таких случаях, как анализ данных.

Важные примечания относительно таблиц MyISAM

Подсистема хранения данных MyISAM поддерживает полнотекстовое индексирование, которое может помочь в сложных операциях поиска. С помощью полнотекстовых

индексов можно индексировать данные, хранящиеся в типах данных BLOB и TEXT. Мы подробно рассмотрим полнотекстовое индексирование в следующих статьях.

Из-за низких накладных расходов MyISAM использует более простую структуру, которая обеспечивает хорошую производительность; однако это не сильно помогает для получения хорошей производительности, когда есть потребность в лучшем параллелизме и случаях использования, которые не нуждаются в тяжелых операциях чтения. Наиболее распространенной проблемой производительности MyISAM является блокировка таблицы, которая может задерживать ваши параллельные запросы в очереди. Это происходит, когда она блокирует таблицу для любой другой операции до тех пор, пока более ранняя операция не будет выполнена.

Таблица MyISAM не поддерживает транзакции и внешние ключи. Судя по всему, из-за этих ограничений вместо таблиц MyISAM теперь системные таблицы схемы MySQL 8 используют таблицы InnoDB.

Memory

Подсистема хранения в памяти (подсистема оперативного хранения данных) обычно называется подсистемой хранения данных на основе кучи. Она используется для чрезвычайно быстрого доступа к данным. Эта подсистема хранения содержит данные в оперативной памяти, поэтому ей не нужны операции ввода- вывода. Поскольку она хранит данные в оперативной памяти, все данные теряются при перезапуске сервера. Такая подсистема в основном используется для временных таблиц или таблицы подстановки. Эта подсистема поддерживает блокировку на уровне таблицы, которая ограничивает параллелизм с высокой частотой записи.

Ниже приведены важные примечания об оперативных таблицах Memory.

- Оперативная таблица хранит данные в оперативной памяти, которая имеет очень ограниченный объем; если вы попытаетесь записать слишком много данных в оперативную таблицу, она начнет свопить данные на диск, и тогда вы потеряете преимущества подсистемы хранения данных в памяти.
- Оперативные таблицы не поддерживают типы данных TEXT и BLOB; такие типы данных могут не потребоваться, так как таблицы имеют ограниченную емкость.
- Эта подсистема хранения может использоваться для кеширования результатов; таблицы поиска, например, или почтовые индексы и названия штатов.
- Оперативные таблицы поддерживают индексы на основе B-дерева и хеш-индексы.

Archive

Эта подсистема хранения данных используется для хранения больших объемов исторических данных без каких-либо индексов. Архивные таблицы не имеют ограничений по объему хранимых данных. Архивная подсистема хранения данных оптимизирована для операций с высокой частотой вставки, а также поддерживает блокировку на уровне строк. Такие таблицы хранят данные в сжатом и малом форматах. Архивная подсистема не поддерживает операции DELETE или UPDATE; она разрешает только операции INSERT, REPLACE и SELECT.

Blackhole

Эта подсистема хранения данных принимает данные, но их не сохраняет. Вместо сохранения данных она отбрасывает (уничтожает) их после каждой вставки.

В следующем ниже примере показана работа таблицы BLACKHOLE:

```
mysql> CREATE TABLE user(id INT, name CHAR(10)) ENGINE = BLACKHOLE;
Query OK, 0 rows affected (0.07 sec)
```

```
mysql> INSERT INTO USER VALUES(1,'Kandarp'),(2,'Chintan');
Query OK, 2 rows affected (0.00 sec)
Records: 2 Duplicates: 0 Warnings: 0
```

```
mysql> SELECT * FROM USER;
Empty set (0.00 sec)
```

И какой тогда прок от такой подсистемы хранения данных? Зачем кому-то ее использовать? Зачем запускать запрос INSERT, который ничего в таблицу не вставляет?

Эта подсистема хранения полезна для репликации с большим количеством серверов. Подсистема хранения данных Blackhole работает в качестве фильтрующего сервера между ведущим и ведомым серверами, который не хранит никаких данных, но который применяет только правила replicate-do-* и replicate-ignore-* и пишет двоичные журналы. Эти двоичные журналы используются для выполнения репликации на ведомых серверах. Мы обсудим это подробно в главе 6 «*Репликация для построения высокодоступных решений*».

CSV

Подсистема хранения данных CSV хранит данные в файлах в формате .csv, используя формат с разделением значений запятыми. Эта подсистема извлекает данные из базы данных и копирует их в .csv. Если вы создаете CSV-файл из электронной таблицы и копируете его на сервер папок данных MYSQL, она может читать данные с помощью запроса SELECT на выборку данных. Аналогичным образом, если вы записываете данные в таблицу, внешняя программа может их прочитать из CSV-файла. Эта подсистема хранения данных используется для обмена данными между программным обеспечением или приложениями. Таблица CSV не поддерживает индексирование и разделение. Чтобы избежать ошибок при создании таблицы, все столбцы в подсистеме хранения данных CSV должны быть определены с атрибутом NOT NULL.

Merge

Эта подсистема хранения данных также называется подсистемой хранения MRG_Myisam. Эта подсистема хранения объединяет все данные в одну таблицу MyISAM и использует ее для ссылки на единственное представление. В таблицах объединения/слияния все столбцы перечисляются в том же порядке. Эти таблицы хороши для сред объединения баз данных.

В следующем ниже примере показано, как создавать таблицы MERGE:

```
mysql> CREATE TABLE user1 (id INT NOT NULL AUTO_INCREMENT PRIMARY
KEY,name
CHAR(20)) ENGINE=MyISAM;
```

```
mysql> CREATE TABLE user2 (id INT NOT NULL AUTO_INCREMENT PRIMARY
KEY,name
CHAR(20)) ENGINE=MyISAM;
```

```
mysql> INSERT INTO user1 (name) VALUES ('abc'),('xyz');
```

```
mysql> INSERT INTO user2 (name) VALUES ('def'),('pqr');
```

```
mysql> CREATE TABLE user (id INT NOT NULL AUTO_INCREMENT,name CHAR(20),
INDEX(id))ENGINE=MERGE UNION=(user1,user2);
```

Как правило, эта подсистема используется для управления таблицами, связанными с журналом регистрации событий. В отдельных таблицах MyISAM можно задавать различные месяцы журналов и объединять эти таблицы с помощью подсистемы хранения данных MERGE.

Таблицы MyISAM имеют ограничение по объему хранения для операционной системы, но коллекция таблиц MyISAM (MERGE) не имеет таких ограничений. Таким образом, использование подсистемы MERGE позволит вам разделять данные на многочисленные таблицы MyISAM, что может помочь в преодолении ограничений по объему хранения.

С помощью подсистемы MERGE трудно выполнять разделение, следовательно, таблицами MERGE оно не поддерживается, и мы не можем реализовать раздел на таблице MERGE или любой таблице MyISAM.

Federated

Подсистема интегрированного хранения данных FEDERATED позволяет создавать одну базу данных на нескольких физических серверах. Она открывает клиентское соединение с другим сервером и выполняет запросы к таблице, получая и отправляя строки по мере необходимости. Первоначально она рекламировалась как конкурентная функциональность, которая поддерживала многие корпоративные проприетарные серверы баз данных, такие как Microsoft SQL Server и Oracle, но это всегда было натяжкой, мягко говоря. Хотя казалось, что в ней задействовалось много гибкости и хитрых приемов, она оказалась источником многих проблем и по умолчанию деактивирована. Однако мы можем ее активировать, запустив двоичный файл сервера MySQL с параметром --federated.

Давайте создадим таблицу FEDERATED.

```
CREATE TABLE user_federated (
   id INT(20) NOT NULL AUTO_INCREMENT,
   name VARCHAR(32) NOT NULL DEFAULT '',
   PRIMARY KEY (id),
   INDEX name (name))
ENGINE=FEDERATED DEFAULT CHARSET=latin1
CONNECTION='mysql://remote_user:[password]@remote_host:port/federated/ta
ble';
```

В поле CONNECTION содержится следующая ниже информация для вашей справки:

- remote_user: имя пользователя удаленного сервера MySQL;
- password: пароль удаленного сервера MySQL;
- remote_host: имя хоста удаленного сервера;
- port: номер порта удаленного сервера;
- federated: имя базы данных удаленного сервера;
- table: имя таблицы базы данных удаленного сервера.

NDB Cluster

NDB Cluster (или просто NDB) - это подсистема хранения прямо в оперативной памяти, обеспечивающая высокую доступность и сохраняемость данных.

Кластерная подсистема хранения данных NDB Cluster может конфигурироваться с помощью ряда параметров аварийного переключения и балансировки нагрузки, но проще всего начать с подсистемы хранения на уровне кластера. NDB Cluster

использует подсистему хранения NDB и содержит полный набор данных, который зависит только от других наборов данных, доступных в кластере.

Кластерная часть NDB Cluster настроена независимо от серверов MySQL. В NDB Cluster каждая часть кластера считается узлом.

80. БД: Какие способы оптимизации производительности баз данных знаете?

Индексирование - один из наиболее эффективных способов наращивания производительности БД, входит в число основных механизмов БД. Как правило, строки БД хранятся в том порядке, в каком создаются. Для извлечения из записи БД некоторой произвольной величины требуется последовательное сканирование соответствующих строк БД. Индекс создает отдельное множество строк, упорядоченных в соответствии с выбранным индексом и содержащих указатели на исходные строки. Индексированные таблицы просматриваются значительно быстрее, чем неиндексированные таблицы. Однако индексирование "съедает" дополнительное дисковое пространство. Кроме того, на модификацию индексированной таблицы требуется больше времени, поскольку все применяемые индексы тоже приходится корректировать. СУБД проводят индексацию автоматически; в крупных СУБД - индексация автоматическая или выборочная.

Кластеризация — это попытка разместить рядом в одном физическом блоке данные тех строк, доступ к которым осуществляется при помощи одинаковых значений ключа. Индексные кластеры, например, удобно использовать для хранения родительской и дочерних строк таблиц, связанных ссылочной целостностью. Кластеры удобно определять для тех наборов атрибутов, соединение по которым проводится наиболее часто, поскольку это увеличивает скорость поиска. Следует отметить, что в реализациях СУБД используется, как правило, один кластер.

Оптимизация кода запросов

Оптимизация работы СУБД. Для оптимизации работы СУБД существует несколько способов, это: блокировка доступа к данным при наличии конфликтующих одновременных обращений; использование серверов приложений; эффективное использование оперативной памяти и памяти на дисках; правильный выбор размера буфера ввода/вывода; кэширование данных; повышение эффективности работы сети; работа с объектными файлами.

Оптимизация структур данных

81. БД: Что такое партиционирование, репликация и шардинг?

Секционирование (англ. *partitioning*) — разделение хранимых объектов баз данных (таких как таблиц, индексов, материализованных представлений) на отдельные части с раздельными параметрами физического хранения. Используется в целях повышения управляемости, производительности и доступности для больших баз данных.

Возможные критерии разделения данных, используемые при секционировании — по предопределённым диапазонам значений, по спискам значений, при помощи значений хеш-функций; в некоторых случаях используются другие варианты. Под *композитными* (составными) критериями разделения понимают последовательно применённые критерии разных типов.

В отличие от сегментирования, где каждый сегмент управляется отдельным экземпляром СУБД, и используются средства координации между ними (что позволяет распределить базу данных на несколько вычислительных узлов), при секционировании доступ ко всем секциям осуществляется из единого экземпляра СУБД (или симметрично из любого экземпляра кластерной СУБД, такого, как Oracle RAC).

Репликация – это поддержание двух и более идентичных копий (реплик) данных на разных узлах РБД. Реплика может включать всю базу данных (полная репликация), одно или несколько взаимосвязанных отношений или фрагмент отношения.

Шардинг — метод разделения и хранения единого логического набора данных в виде множества баз данных. Другое определение шардинга — горизонтальное разделение данных.

Идея шардинга состоит в том, чтобы отказаться от модели, в которой каждая нода должна вычислять каждую операцию, в пользу модели параллельного выполнения, в которой ноды обрабатывают только определенные вычисления. Это позволяет параллельно обрабатывать множество транзакций.

82. БД: Чем отличаются SQL от NoSQL базы данных?

Основное **различие между реляционной и нереляционной базой данных** состоит в том, что **реляционная база данных** хранит данные в таблицах, в то время как **нереляционная база данных** хранит данные в формате ключ-значение, в документах или каким-либо другим способом без использования таблиц, таких как **реляционная база данных**.

SQL основаны на таблицах, а NoSQL — на документах, парах ключ-значение, графовых БД, хранилищах с широкими столбцами.

SQL используют универсальный язык структурированных запросов для определения и обработки данных. Это накладывает определенные ограничения: прежде чем начать обработку, данные надо разместить внутри таблиц и описать.

NoSQL таких ограничений не имеет. Динамические схемы для неструктурированных данных позволяют:

- ориентировать информацию на столбцы или документы;
- основывать ее на графике;
- организовывать в виде хранилища KeyValue;
- создавать документы без предварительного определения их структуры, использовать разный синтаксис;
- добавлять поля непосредственно в процессе обработки.

NoSQL (от англ. *not only SQL — не только SQL*) — обозначение широкого класса разнородных систем управления базами данных, появившихся в конце 2000-х — начале 2010-х годов и существенно отличающихся от традиционных реляционных СУБД с доступом к данным средствами языка SQL. Применяется к системам, в которых делается попытка решить проблемы масштабируемости и доступности за счёт полного или частичного отказа от требований атомарности и согласованности данных.

Масштабируемость

В большинстве случаев базы данных SQL можно масштабировать по вертикали. Что это значит? Можно увеличить нагрузку на один сервер, увеличив таким образом ЦП, ОЗУ или объем накопителя.

В отличие от SQL базы данных NoSQL масштабируются по горизонтали. Это означает, что больший трафик обрабатывается путем разделения или добавления большего количества серверов. Это делает NoSQL удобнее при работе с большими или меняющимися наборами данных.

В КАКИХ СЛУЧАЯХ ИСПОЛЬЗУЮТ SQL?

SQL подойдет, если нужна обработка большого количества сложных запросов, или рутинного анализа данных. Выбирайте реляционную БД, если нужна надежная обработка транзакций и ссылочная целостность.

А В КАКИХ NOSQL?

Если объем данных большой, лучше использовать NoSQL. Отсутствие явных структурированных механизмов ускорит процесс обработки Big Data. А еще это безопаснее — такие БД сложнее взломать.

Выбирайте NoSQL, если:

- необходимо хранить массивы в объектах JSON;
- записи хранятся в коллекции с разными полями или атрибутами;
- необходимо горизонтальное масштабирование.

83. БД: Какие бывают NoSQL базы данных?

NoSQL (от англ. *not only SQL — не только SQL*) — обозначение широкого класса разнородных систем управления базами данных, появившихся в конце 2000-х — начале 2010-х годов и существенно отличающихся от традиционных реляционных СУБД с доступом к данным средствами языка SQL. Применяется к системам, в которых делается попытка решить проблемы масштабируемости и доступности за счёт полного или частичного отказа от требований атомарности и согласованности данных.

Согласно рейтингу Boodet.Online, самыми удобными системами для обработки нереляционных баз данных являются: MongoDB, Apache Cassandra и Google Cloud BigTable. У всех трех широкий функционал и высокий уровень гибкости.

MongoDB

MongoDB — это качественный бесплатный продукт, который чаще всего используют при работе с NoSQL. Решение позволяет менять схемы данных в процессе работы, масштабироваться по горизонтали. Интерфейс очень простой — в нем легко разберется любой сотрудник компании, не обязательно быть IT-профессионалом.

Почему мы поставили Mongo на первое место в списке лидеров обработки нереляционных баз данных? Все дело в новой функции от разработчиков. Теперь в решении есть глобальная облачная БД, что дает возможность развернуть управляемую MongoDB через AWS, Azure, GCP.

Apache Cassandra

Apache Cassandra — это продукт с открытым исходным кодом, а значит, достаточно гибкий, адаптируемый практически для любых задач. Идентичность узлов упрощает масштабирование для наращивания архитектуры БД.

Apache Cassandra подойдет для масштабных проектов. Продукт обеспечивает высокую скорость чтения и записи. Даже если часть решения использует SQL, можно применить подобные SQL операторы: DDL, DML, SELECT. Для более высокого уровня безопасности есть резервное копирование и восстановление.

Apache Cassandra — это один из немногих инструментов обработки баз данных, который гарантирует безотказность работы (подробнее читайте в своем SLA).

Google Cloud BigTable

Неплохой продукт от Google, который гарантирует задержку обработки не более 10 мс. BigTable уделяют безотказности много внимания. Например, благодаря функции репликации базы данных более долговечны, доступны и устойчивы при зональных сбоях. Это отличный вариант для работы с Big Data в режиме реального времени (машинное зрение, AI) — можно изолировать рабочую нагрузку для приоритетной аналитики.

Berkeley DB, MemcacheDB, Redis, Riak, Amazon DynamoDB

84. БД: Какие типы данных есть в MySQL?

Символьные типы

- **CHAR**: представляет строку фиксированной длины.

  Длина хранимой строки указывается в скобках, например, CHAR(10) - строка из десяти символов. И если в таблицу в данный столбец сохраняется строка из 6

символов (то есть меньше установленной длины в 10 символов), то строка дополняется 4 пробелами и в итоге все равно будет занимать 10 символов

Тип CHAR может хранить до 255 байт.

- **VARCHAR**: представляет строку переменной длины.

  Длина хранимой строки также указывается в скобках, например, VARCHAR(10). Однако в отличие от CHAR хранимая строка будет занимать именно столько места, сколько необходимо. Например, если определеная длина в 10 символов, но в столбец сохраняется строка в 6 символов, то хранимая строка так и будет занимать 6 символов плюс дополнительный байт, который хранит длину строки.

  Всего тип VARCHAR может хранить до 65535 байт.

Начиная с MySQL 5.6 типы CHAR и VARCHAR по умолчанию используют кодировку UTF-8, которая позволяет использовать до 3 байт для хранения символа в заивисимости от языка (для многих европейских языков по 1 байту на символ, для ряда восточно-европейских и ближневосточных - 2 байта, а для китайского, яполнского, корейского - по 3 байта на символ).

Ряд дополнительных типов данных представляют текст неопределенной длины:

- **TINYTEXT**: представляет текст длиной до 255 байт.
- **TEXT**: представляет текст длиной до 65 КБ.
- **MEDIUMTEXT**: представляет текст длиной до 16 МБ
- **LARGETEXT**: представляет текст длиной до 4 ГБ

Числовые типы

- **TINYINT**: представляет целые числа от -128 до 127, занимает 1 байт

- **BOOL**: фактически не представляет отдельный тип, а является лишь псевдонимом для типа TINYINT(1) и может хранить два значения 0 и 1. Однако данный тип может также в качестве значения принимать встроенные константы **TRUE** (представляет число 1) и **FALSE** (предоставляет число 0).

  Также имеет псевдоним **BOOLEAN**.

- **TINYINT UNSIGNED**: представляет целые числа от 0 до 255, занимает 1 байт

- **SMALLINT**: представляет целые числа от -32768 до 32767, занимает 2 байта

- **SMALLINT UNSIGNED**: представляет целые числа от 0 до 65535, занимает 2 байта

- **MEDIUMINT**: представляет целые числа от -8388608 до 8388607, занимает 3 байта

- **MEDIUMINT UNSIGNED**: представляет целые числа от 0 до 16777215, занимает 3 байта

- **INT**: представляет целые числа от -2147483648 до 2147483647, занимает 4 байта

- **INT UNSIGNED**: представляет целые числа от 0 до 4294967295, занимает 4 байта

- **BIGINT**: представляет целые числа от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807, занимает 8 байт

- **BIGINT UNSIGNED**: представляет целые числа от 0 до 18 446 744 073 709 551 615, занимает 8 байт

- **DECIMAL**: хранит числа с фиксированной точностью. Данный тип может принимать два параметра precision и scale: DECIMAL(precision, scale).

  Параметр precision представляет максимальное количество цифр, которые может хранить число. Это значение должно находиться в диапазоне от 1 до 65.

  Параметр scale представляет максимальное количество цифр, которые может содержать число после запятой. Это значение должно находиться в диапазоне от 0 до значения параметра precision. По умолчанию оно равно 0.

  Например, в определении следующего столбца:

  1

  salary DECIMAL(5,2)

  Число 5 - precision, а число 2 - scale, поэтому данный столбец может хранить значения из диапазона от -999.99 до 999.99.

  Размер данных в байтах для DECIMAL зависит от хранимого значения.

  Данный тип также имеет псевдонимы **NUMERIC**, **DEC**, **FIXED**.

- **FLOAT**: хранит дробные числа с плавающей точкой одинарной точности от -3.4028 * 1038 до 3.4028 * 1038, занимает 4 байта.

  Может принимать форму FLOAT(M,D), где M - общее количество цифр, а D - количество цифр после запятой

- **DOUBLE**: хранит дробные числа с плавающей точкой двойной точности от -1.7976 * 10308 до 1.7976 * 10308, занимает 8 байт. Также может принимать форму DOUBLE(M,D), где M - общее количество цифр, а D - количество цифр после запятой.

  Данный тип также имеет псевдонимы **REAL** и **DOUBLE PRECISION**, которые можно использовать вместо DOUBLE.

Типы для работы с датой и временем

- **DATE**: хранит даты с 1 января 1000 года до 31 декабря 9999 года (с "1000-01-01" до "9999-12-31"). По умолчанию для хранения используется формат yyyy-mm-dd. Занимает 3 байта.
- **TIME**: хранит время от -838:59:59 до 838:59:59. По умолчанию для хранения времени применяется формат "hh:mm:ss". Занимает 3 байта.
- **DATETIME**: объединяет время и дату, диапазон дат и времени - с 1 января 1000 года по 31 декабря 9999 года (с "1000-01-01 00:00:00" до "9999-12-31 23:59:59"). Для хранения по умолчанию используется формат "yyyy-mm-dd hh:mm:ss". Занимает 8 байт
- **TIMESTAMP**: также хранит дату и время, но в другом диапазоне: от "1970-01-01 00:00:01" UTC до "2038-01-19 03:14:07" UTC. Занимает 4 байта
- **YEAR**: хранит год в виде 4 цифр. Диапазон доступных значений от 1901 до 2155. Занимает 1 байт.

Тип Date может принимать даты в различных форматах, однако непосредственно для хранения в самой бд даты приводятся к формату "yyyy-mm-dd". Некоторые из принимаемых форматов:

- yyyy-mm-dd - 2018-05-25

- yyyy-m-dd - 2018-5-25

- yy-m-dd - 18-05-25

  В таком формате двузначные числа от 00 до 69 воспринимаются как даты в диапазоне 2000-2069. А числа от 70 до 99 как диапазон чисел 1970 - 1999.

- yyyymmdd - 20180525

- yyyy.mm.dd - 2018.05.25

Для времени тип Time использует 24-часовой формат. Он может принимать время в различных форматах:

- hh:mi - 3:21 (хранимое значение 03:21:00)
- hh:mi:ss - 19:21:34
- hhmiss - 192134

Примеры значений для типов DATETIME и TIMESTAMP:

- 2018-05-25 19:21:34
- 2018-05-25 (хранимое значение 2018-05-25 00:00:00)

Составные типы

- **ENUM**: хранит одно значение из списка допустимых значений. Занимает 1-2 байта
- **SET**: может хранить несколько значений (до 64 значений) из некоторого списка допустимых значений. Занимает 1-8 байт.

Бинарные типы

- **TINYBLOB**: хранит бинарные данные в виде строки длиной до 255 байт.
- **BLOB**: хранит бинарные данные в виде строки длиной до 65 КБ.
- **MEDIUMBLOB**: хранит бинарные данные в виде строки длиной до 16 МБ
- **LARGEBLOB**: хранит бинарные данные в виде строки длиной до 4 ГБ

85. БД: Разница между LEFT JOIN, RIGHT JOIN, INNER JOIN?

Внутреннее объединение является наиболее используемым. INNER JOIN возвращает только те строки, где соблюдается условие объединения.

Следующие два вида объединений используют модификатор (LEFT или RIGHT), который определяет, какие данные таблицы включаются в результирующий набор.

Примечание: LEFT JOIN и RIGHT JOIN также называются LEFT OUTER JOIN и RIGHT OUTER JOIN (соответственно левое и правое внешнее объединение).

Эти виды объединений используются в запросах, когда мы хотим вернуть все данные из одной таблицы и добавить к ним связанные данные из другой таблицы (*если таковые есть*).

Если связанных данных во второй таблице нет, мы получаем в выводе все данные только из «первичной» таблицы.

То есть, это запрос информации о конкретной вещи с дополнительными сведениями, если они есть.

86. БД: Разница между JOIN и UNION?

JOIN и UNION делают одно и то же - объединяют SELECT запросы. Только JOIN добавляет столбцы в результирующую таблицу, а UNION пририсовывает к концу имеющейся таблицы ещё одну.

JOIN полезен если есть некая таблица и захотели выбрать связанные записи. Например, взяли таблицу "продажи" и дополнительно выбрали наименование клиента у каждой операции. Склеивание таблиц пойдёт по внешним ключам.

UNION полезен если нужно соединить 2 таблицы. К примеру ситуация. Есть продажи оптовые, хранятся в таблице оптовых продаж. И розничные. В таблице розничных продаж. Делаешь

select ...from "оптовые"

union

select ...from "розничные"

Только надо помнить что union требует одинаковости колонок во всех SELECT. И получаешь суммарные продажи по всему предприятию.

87. БД: Что такое индексы? Как они влияют на время выполнения SELECT, INSERT?

Индекс (англ. *index*) — объект базы данных, создаваемый с целью повышения производительности поиска данных. Таблицы в базе данных могут иметь большое количество строк, которые хранятся в произвольном порядке, и их поиск по заданному критерию путём последовательного просмотра таблицы строка за строкой может занимать много времени. Индекс формируется из значений одного или нескольких столбцов таблицы и указателей на соответствующие строки таблицы и, таким образом, позволяет искать строки, удовлетворяющие критерию поиска. Ускорение работы с использованием индексов достигается в первую очередь за счёт того, что индекс имеет структуру, оптимизированную под поиск — например, сбалансированного дерева.

Для оптимальной производительности запросов индексы обычно создаются на тех столбцах таблицы, которые часто используются в запросах. Для одной таблицы может быть создано несколько индексов. Однако увеличение числа индексов замедляет операции добавления, обновления, удаления строк таблицы, поскольку при этом приходится обновлять сами индексы. Кроме того, индексы занимают дополнительный объем памяти, поэтому перед созданием индекса следует убедиться, что планируемый выигрыш в производительности запросов превысит дополнительную затрату ресурсов компьютера на сопровождение индекса.

Преимущества индексации следующие:

Ускорить запросы

Создайте уникальный индекс, чтобы обеспечить уникальность данных в таблице данных.

Добиться целостности данных, ускорить связь между таблицами и таблицами

Сократить время группировки и сортировки

Увеличение индекса также имеет много недостатков, в основном в следующих аспектах:

Создание и ведение индексов требует времени, и потребляемое время будет увеличиваться по мере увеличения объема данных.

Индексы должны занимать дисковое пространство.

В дополнение к пространству данных, занимаемому таблицей данных, каждый индекс также занимает определенное физическое пространство.

Если существует большое количество индексов, файл индекса может достичь максимального размера файла быстрее, чем файл данных.

Когда данные в таблице добавляются, удаляются и изменяются, индекс также должен поддерживаться динамически, что снижает скорость обслуживания данных.

88. БД: Что такое хранимые процедуры, функции и триггеры в MySQL? Для чего они? Приведите примеры использования.

Хранимая процедура MySQL представляет собой подпрограмму, хранящуюся в базе данных. Она содержит имя, список параметров и операторы **SQL**. Все популярные системы управления базами данных поддерживают хранимые процедуры. Они были введены в **MySQL 5**.

В MySQL **функция** - это хранимая программа, в которую вы можете передавать параметры и возвращать значение.

Триггер — это хранимая процедура, которая не вызывается непосредственно, а исполняется при наступлении определенного события (вставка, удаление, обновление строки). Поддержка триггеров в MySQL началась с версии 5.0.2

89. БД: Как организовать сохранность вложенных категорий в MySQL?

Древовидные структуры - это такие структуры, где есть родители и дети, например, каталог товаров:

```
Бытовая техника (id=1)
    Телевизоры (id=2)
        Плазменные (id=3)
        LCD (id=4)
    Холодильники (id=5)
        Маленькие (id=6)
        Средние (id=7)
        Большие (id=8)
```

```
Профессиональная техника (id=9)
    ...
```

Типичные задачи, которые встречаются при работе с такими структурами:

- выбрать всех детей элемента
- выбрать всех потомков (детей и их детей) элемента
- выбрать цепочку предков элемента (родитель, его родитель, и так далее)
- переместить элемент (и его потомков) из одной группы в другую
- удалить элемент из таблицы (со всеми потомками)

У каждой записи есть идентификатор — уникальное число, он на схеме написан в скобках (думаю, это ты и так знаешь). Рассмотрим способы хранения таких данных.

Добавить колонку parent_id (метод Adjacency List)

Мы добавляем к каждой записи колонку `parent_id` (и индекс на нее), которая хранит id родительской записи (если родителя нет — NULL). Это самый простой, но самый неэффективный способ. Вот как будет выглядеть вышеприведенное дерево:

```
Бытовая техника (id=1, parent_id=NULL)
    Телевизоры (id=2, parent_id=1)
        Плазменные (id=3,parent_id=2)
        LCD (id=4,parent_id=2)
    Холодильники (id=5,parent_id=1)
```

Выбрать всех детей просто: `SELECT WHERE parent_id = ?`, но другие операции требуют выполнения нескольких запросов и на больших деревьях особо неэффективны. Например, выбор всех потомков элемента с идентификатором `:id`

- выбрать список детей :id (`SELECT WHERE parent_id = :id`)
- выбрать список их детей (`SELECT WHERE parent_id IN (:children1)`)
- выбрать список детей детей (`SELECT WHERE parent_id IN (:children2)`)

И так, пока мы не дойдем до самого младшего ребенка. После этого надо еще отсортировать и объединить результаты в дерево.

Плюсом, впрочем, является быстрая вставка и перемещение веток, которые не требуют никаких дополнительных запросов, и простота реализации. Если можно эффективно кешировать выборки, это в общем-то нормальный и

работающий вариант (например, для меню сайта). Это может быть годный вариант для часто меняющихся данных.

Иногда еще добавляют поле `depth`, указывающее глубину вложенности, но его надо не забыть обновлять при перемещении ветки.

Теория: http://xpoint.ru/forums/computers/dbms/mysql/thread/34068.xhtml, http://www.opennet.ru/docs/RUS/hierarchical_data/

Closure table — усовершенствование предыдущего способа

В этом способы мы так же добавляет поле `parent_id`, но для оптимизации рекурсивных выборок создаем дополнительную таблицу, в которой храним всех потомков (детей и их детей) и их глубину относительно родителя каждой записи. Поясню. Дополнительная таблица выглядит так:

```
parent_id    child_id   depth
1            1          0       // Перечислены все дети записи с id = 1
1            2          1
1            3          2
1            4          2
1            5          1
....
2            2          0
2            3          1
2            4          1
```

Чтобы узнать всех потомков записи, мы (в отличие от предыдущего способа), делаем запрос к дополнительной таблице: `SELECT child_id FROM closure_table WHERE parent_id = :id`, получаем `id` потомков и выбираем их их основной таблицы: `SELECT WHERE id IN (:children)`. Если таблицы хранятся в одной БД, запросы можно объединить в один с использованием JOIN.

Данные потом надо будет вручную отсортировать в дерево.

Узнать список предков можно тоже одним запросом к таблице связей: `SELECT parent_id FROM closure_table WHERE child_id = :id ORDER BY depth`

Минусы метода: нужно поддерживать таблицу связей, она может быть огромной (размер посчитайте сами), при вставке новых записей и при перемещении веток нужны сложные манипуляции. Если таблица часто меняется, это не лучший способ.

Плюсы: относительная простота, быстрота выборок.

Теория: http://dirtsimple.org/2010/11/simplest-way-to-do-tree-based-queries.html

Nested sets

Идея в том, что мы добавляем к каждой записи поля `parent_id`, `depth`, `left`, `right` и выстраиваем записи хитрым образом. После этого выборка всех потомков (причем уже отсортированных в нужном порядке) делаетсяпростым запросом вида `SELECT WHERE left >= :a AND right <= :b`

Минусы: необходимость пересчитывать `left`/`right` при вставке записей в середину или удалении, сложное перемещение веток, сложность в понимании.

Плюсы: скорость выборки

Теория: http://www.webscript.ru/stories/04/09/01/8197045

В общем-то, годный вариант для больших таблиц, которые часто выбираются, но меняются нечасто (например, только через админку, где не критична производительность).

Materialized Path

Идея в том, что записи в пределах одной ветки нумеруются по порядку и в каждую запись добавляется поле path, содержащее полный список родителей. Напоминает способ нумерации глав в книгах. Пример:

```
Бытовая техника (id=1, number=1, path=1)
    Телевизоры (id=2, number=1, path=1.1)
        Плазменные (id=3, number=1, path=1.1.1)
        LCD (id=4, number=2, path=1.1.2)
    Холодильники (id=5, number=2, path=1.2)
```

При этом способе path хранится в поле вроде TEXT или BINARY, по нему делается индекс. Выбрать всех потомков можно запросом `SELECT WHERE path LIKE '1.1.%' ORDER BY path`, который использует индекс.

Плюс: записи выбираются уже отсортированными в нужном порядке. Простота решения и скорость выборок высокая (1 запрос). Быстрая вставка.

Минусы: при вставке записи в середину надо пересчитывать номера и пути следующих за ней. При удалении ветки, возможно тоже. При перемещении ветки надо делать сложные расчеты. Глубина дерева и число детей у родителя ограничены выделенным дял них местом и длиной path

Этот способ отлично подходит для древовидных комментариев.

90. Composer: Что такое Composer?

Composer — это пакетный менеджер уровня приложений для языка программирования PHP, который предоставляет средства по управлению зависимостями в PHP-приложении. Composer разработали и продолжают поддерживать два программиста Nils Adermann и Jordi Boggiano. Они начали разрабатывать Composer в апреле 2011, а первый релиз состоялся 1 марта 2012. Идея создания пакетных менеджеров уровня приложений не нова и его авторы вдохновлялись уже существовавшими на тот момент времени npm для Node.js и bundler для Ruby.

Composer работает через интерфейс командной строки и устанавливает зависимости (например библиотеки) для приложения. Он также позволяет пользователям устанавливать PHP-приложения, которые доступны на packagist.org, который является его основным репозиторием, где содержатся все доступные пакеты.

91. Composer: Чем отличается require от require-dev?

В require добавляются зависимости, которые необходимы для работы пакета.

В require-dev добавляются зависимости, которые нужны *для разработки самого пакета*. Например, phpunit/phpunit. Опция require-dev является root-only опцией, то есть она не читается и зависимости из неё *не ставятся* когда пакет устанавливается не напрямую, а как зависимость корневого пакета-проекта.

То же можно сказать и о ряде других опций composer.json:

- autoload-dev, куда стоит прописывать namespace для тестов.
- minimum-stability.
- prefer-stable.
- repositories.
- config.
- scripts

92. Bitrix: Система кэширования

https://dev.1c-bitrix.ru/learning/course/index.php?COURSE_ID=43&CHAPTER_ID=03485&LESSON_PATH=3913.2704.3485

Кеширование - технология, позволяющая сохранять результаты работы редко обновляемых и ресурсоемких кусков кода (например, активно работающих с базой данных) в специальном хранилище для более быстрого доступа к ним.

Система Bitrix Framework включают в себя разные технологии кеширования:

- **Кеширование компонентов** (или **Автокеширование**) - все динамические компоненты, которые используются для создания веб-страниц, имеют встроенную поддержку управления кешированием.
  Для использования этой технологии достаточно включить автокеширование одной кнопкой на административной панели. При этом все компоненты, у которых был включен режим автокеширования, создадут кеши и полностью перейдут в режим работы без запросов к базе данных.
- **Неуправляемое кеширование** - возможность задать правила кеширования ресурсоемких частей страниц. Результаты кеширования сохраняются в виде файлов в каталоге `/bitrix/cache/`. Если время кеширования не истекло, то вместо ресурсоемкого кода будет подключен предварительно созданный файл кеша.
  Кеширование называется неуправляемым, поскольку кеш не перестраивается автоматически после модификации исходных данных, а действует указанное время после создания, которое задается в диалоге **Параметры компонента**.
  Правильное использование кеширования позволяет увеличить общую производительность сайта на порядок. Однако необходимо учитывать, что неразумное использование кеширования может привести к серьезному увеличению размера каталога `/bitrix/cache/`.
- **Управляемый кеш** - автоматически обновляет кеш компонентов при изменении данных. Для часто обновляемого большого массива данных использование управляемого кеша неоправданно.
- **HTML кеш** лучше всего включить на какой-нибудь редко изменяющийся раздел с регулярным посещением анонимных посетителей. Технология проста в эксплуатации, не требует от пользователя отслеживать изменения, защищает дисковой квотой от накрутки данных и самовосстанавливает работоспособность при превышении квоты или изменении данных. (Считается устаревшей, рекомендуется использовать технологию Композитный сайт.) В плане многосайтовости поддерживает только многосайтовость на одном домене.
- **Кеширование меню**. Для кеширования меню применяется специальный алгоритм, который учитывает тот факт, что большая часть посетителей - это незарегистрированные пользователи.
  Кеш меню управляемый и обновляется при редактировании меню или изменении прав доступа к файлам и папкам через административный интерфейс и API.

93. Bitrix: Сервис Локатор

https://dev.1c-bitrix.ru/learning/course/index.php?COURSE_ID=43&LESSON_ID=14032&LESSON_PATH=3913.3516.5062.14032

Сервис локатор (локатор служб) - это шаблон проектирования для удобной работы с сервисами приложения.

Идея сервиса в том, что вместо создания конкретных сервисов напрямую (с помощью new), используется специальный объект (сервис локатор), который будет отвечать за создание, нахождение сервисов. Своего рода реестр.

Класс `\Bitrix\Main\DI\ServiceLocator` реализует интерфейс PSR-11.

94. Bitrix: Контроллеры

https://dev.1c-bitrix.ru/learning/course/index.php?COURSE_ID=43&LESSON_ID=6436&LESSON_PATH=3913.3516.5062.3750.6436

Контроллеры - это часть MVC архитектуры, которая отвечает за обработку запроса и генерирование ответа.

Контроллеры состоят из действий, которые являются основной сутью и их в конечном итоге запрашивает пользователь для получения результата. В одном контроллере может быть одно или несколько действий.

Контроллеры должны быть унаследованы от \Bitrix\Main\Engine\Controller или его потомков. Контроллеры могут располагаться внутри модуля, либо внутри компонента в файле **ajax.php** и быть контроллером для компонента.

Создание действий, это создание просто методов в конкретном контроллере. Метод обязан быть public и иметь суффикс Action.

Жизненный цикл контроллера

При обработке запроса Application создаёт контроллер на основе соглашения по именованию. Далее работу выполняет контроллер:

- *Controller::init()* будет вызван после того, как контроллер создан и сконфигурирован.
- Контроллер создает объект действия
    - Если действие не удалось создать, выкидывается исключение.
- Контроллер вызывает метод подготовки параметров *Controller::prepareParams*.
- Контроллер вызывает метод *Controller::processBeforeAction(Action $action)*, в случае возврата *true* выполнение продолжается.
- Контроллер выкидывает событие модуля **main** *{полное_имя_класс_контроллера}::onBeforeAction*, в случае возврата *EventResult* не равном `EventResult::SUCCESS` выполнение блокируется. На данном событии выполняются префильтры.
- Контроллер запускает действие

    - Параметры действия будут сопоставлены с данными из запроса
- Контроллер выкидывает событие модуля **main** *{полное_имя_класс_контроллера}::onAfterAction*. На данном событии выполняются постфильтры.
- Контроллер вызывает метод`Controller::processAfterAction(Action $action, $result)`.
- Приложение получает результат выполнения действия и если это данные, то создает `\Bitrix\Main\Engine\Response\AjaxJson` с этими данными, либо отправляет объект ответа от действия.
- Приложение вызывает метод *Controller::finalizeResponse($response)*, передавая финальный вариант ответа, который будет отправлен пользователю после всех событий и подготовок.
- Вывод $response пользователю.

95. Bitrix: Вложенные транзакции

https://dev.1c-bitrix.ru/learning/course/index.php?COURSE_ID=43&LESSON_ID=14380&LESSON_PATH=3913.3516.5062.14380

Транза́кция (англ. transaction) - группа последовательных операций с базой данных, которая представляет собой логическую единицу работы с данными. Транзакция может быть выполнена либо целиком и успешно, соблюдая целостность данных и независимо от параллельно идущих других транзакций, либо не выполнена вообще, и тогда она не должна произвести никакого эффекта.

Для транзакций нужно использовать API драйвера БД `\Bitrix\Main\DB\Connection`, методы startTransaction(), commitTransaction() и rollbackTransaction().

Если транзакции вложенные, то повторные старты транзакций создают именованные точки сохранения MySQL `SAVEPOINT`. Промежуточные коммиты ничего не коммитят. Последний закрывающий коммит коммитит все произведенные с начала первой транзакции изменения.

Сложности возникают, если какие-то из вложенных транзакций захотят откатить изменения в своей транзакции. В этом случае мы попадаем в неопределенную ситуацию, т.к. итоговую цель всех изменений знает только конечный сценарий (см. старое правило).

Поэтому мы действуем в парадигме "вложенные роллбеки не поддерживаются" и даем возможность конечному сценарию решить, как поступить правильно. При наступлении события вложенного роллбека происходит частичный откат к соответствующей точке

сохранения `ROLLBACK TO SAVEPOINT ...` и выбрасывается исключение `\Bitrix\Main\DB\TransactionException`.

Далее возможны три сценария:

- Исключение никто не ловит, скрипт прекращает работу по ошибке, MySQL автоматически откатывает все незакомиченные изменения.
- Исключение обрабатывает вызывающий код, он решает, что внутренние роллбеки для правильной работы не важны, и продолжает работу с последующим коммитом своих изменений.
- Исключение обрабатывает вызывающий код, он решает, что нет смысла продолжать, и откатывает свои изменения. При этом если вызывающий код сам является вложенной транзакцией, то генерируется новое исключение - и так до самого первого уровня в конечном сценарии.

```
$conn = \Bitrix\Main\Application::getConnection();

$conn->query("truncate table test");

try{
    $conn->startTransaction();

    $conn->query("insert into test values (1, 'one')");

    // nested transaction
    $conn->startTransaction();

    $conn->query("insert into test values (2, 'two')");

    if (true) {
        $conn->commitTransaction();
    }  else  {
        $conn->rollbackTransaction();
    }
    // end of nested transaction

    $conn->commitTransaction();
}catch(\Bitrix\Main\DB\TransactionException $e){
    $conn->rollbackTransaction();
}
```

Рекомендуется использовать транзакции в API, чтобы делать атомарные изменения, обеспечивать целостность данных и изолировать изменения. Транзакции должны

охватывать непосредственно модификацию данных; эти изменения должны рассматриваться как одна логическая операция. Вложенные транзакции поддерживаются. Тем не менее, вложенные роллбеки должны быть обработаны конечным сценарием и в целом не рекомендуются.

96. Bitrix: ORM. Концепция

https://dev.1c-bitrix.ru/learning/course/index.php?COURSE_ID=43&CHAPTER_ID=05748&LESSON_PATH=3913.3516.5748

ORM (англ. Object-relational mapping, рус. Объектно-реляционное отображение) - технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая “виртуальную объектную базу”. Операции выборки и сохранения в БД - однотипные, с одинаковыми параметрами и фильтрами. По возможности, таблицы сущностей обслуживаются минимумом нового кода. Стандартные события добавления/изменения/удаления доступны автоматически.

Для реализации этих целей введены понятия:

- Сущности (*Bitrix\Main\Entity\Base*);
- Поля сущностей (*Bitrix\Main\Entity\Field* и его наследники);
- Датаменеджер (*Bitrix\Main\Entity\DataManager*).

Сущность описывает таблицу в БД, в том числе содержит поля сущностей. Датаменеджер производит операции выборки и изменения сущности. На практике же работа в основном ведется на уровне датаменеджера.

Для операций записи используются три метода:*add*, *update*, *delete*.

Перед записью новых данных в БД нужно обязательно проверять их на корректность. Для этого предусмотрены **валидаторы**.

Поддерживаются события:

- *OnBeforeAdd* (параметры: fields)
- *OnAdd* (параметры: fields)
- *OnAfterAdd* (параметры: fields, primary)
- *OnBeforeUpdate* (параметры: primary, fields)
- *OnUpdate* (параметры: primary, fields)
- *OnAfterUpdate* (параметры: primary, fields)
- *OnBeforeDelete* (параметры: primary)
- *OnDelete* (параметры: primary)
- *OnAfterDelete* (параметры: primary)

Иногда может возникнуть необходимость хранить данные в одном формате, а работать с ними в программе уже в другом. Самый распространенный пример: работа с массивом и его сериализация перед сохранением в БД. На этот случай предусмотрены параметры поля `'save_data_modification'` и `'fetch_data_modification'`. Определяются они аналогично валидаторам, через callback.

97. Bitrix: ORM. Выборки в отношениях 1:N и N:M

https://dev.1c-bitrix.ru/learning/course/index.php?COURSE_ID=43&LESSON_ID=3250&LESSON_PATH=3913.3516.5748.5063.3250

При работе с отношениями 1:N и N:M в общем случае и с множественными свойствами инфоблока в частности можно столкнуться с двумя проблемами.

Логика LIMIT

Интуитивное ожидание логики работы LIMIT:

```
$iblockEntity = IblockTable::compileEntity(...);

  $query = $iblockEntity->getDataClass()::query()
      ->addSelect('NAME')
      ->addSelect('MULTI_PROP_1')
  ->setLimit(5);

  $elements = $query->fetchCollection();
```

Ожидать в данном примере выборку 5 элементов будет ошибкой. Лимит указывается не на уровне объектов, а на уровне SQL запроса:

```
 SELECT ... FROM `b_iblock_element`
      LEFT JOIN `b_iblock_element_property` ...
  LIMIT 5
```

Фактически будет выбрано 5 значений свойств с соответствующими элементами. Поэтому в выборке может оказаться менее 5 элементов, или вовсе 1 элемент с не полностью выбранными значениями свойства.

Выбор полей соотношений в одном запросе

Если выбирать несколько полей отношений в одном запросе, то результатом будет декартово произведение всех записей. Например:

```
   $iblockEntity = IblockTable::compileEntity(...);

    $query = $iblockEntity->getDataClass()::query()
        ->addSelect('NAME')
        ->addSelect('MULTI_PROP_1')
        ->addSelect('MULTI_PROP_2')
        ->addSelect('MULTI_PROP_3');

    $elements = $query->fetchCollection();
```

```
Выполнится запрос вида:
   SELECT ... FROM `b_iblock_element`
        LEFT JOIN `b_iblock_element_property` ... // 15 значений свойства
        LEFT JOIN `b_iblock_element_property` ... // 7 значений свойства
        LEFT JOIN `b_iblock_element_property` ... // 11 значений свойства
```

И если интуитивно кажется, что будет выбрано 15 + 7 + 11 = 33 строки, то фактически будет выбрано 15 * 7 * 11 = 1155 строк. Если свойств или значений в запросе еще больше, то счет может идти на миллионы результирующих записей, и как следствие - о нехватке памяти в приложении для получения всего результата.

Решение проблем

Для обхода этих проблем был добавлен класс *Bitrix\Main\ORM\Query\QueryHelper* с универсальным методом **decompose**:

```
   /**
    ** Декомпозиция запросов с 1:N и N:M отношениями
    **
    ** @param Query $query
    ** @param bool $fairLimit При установке этой опции сначала выбираются ID
объектов, а следующим запросом остальные данные с фильтром по ID
    ** @param bool $separateRelations При установке этой опции каждое 1:N
или N:M отношение выбирается отдельным запросом
    ** @return Collection
    **/
    public static function decompose(Query $query, $fairLimit = true,
$separateRelations = true)
```

Параметр **fairLimit** приводит к двум запросам: сначала выбирается *primary* записей с заданным *Limit / Offset* в запросе, и после этого для всех *primary* одним запросом выбираются все отношения.

Дополнительный параметр **separateRelations** позволяет выполнить отдельный запрос на каждое отношение, чтобы не возникало декартова произведения всех записей.

В качестве результата будет возвращена готовая коллекция объектов с уже объединенными данными.

98. Bitrix: Роутинг

https://dev.1c-bitrix.ru/learning/course/index.php?COURSE_ID=43&CHAPTER_ID=013764&LESSON_PATH=3913.3516.5062.13764

Роутинг (маршрутизация) — процесс определения оптимального маршрута данных в сетях связи.

Для пользовательских модулей использование собственных роутов в папке модуля на данный момент не предусмотрено.

Для запуска новой системы роутинга нужно перенаправить обработку 404 ошибок на файл **routing_index.php** в файле **.htaccess**:

```
#RewriteCond %{REQUEST_FILENAME} !/bitrix/urlrewrite.php$
#RewriteRule ^(.*)$ /bitrix/urlrewrite.php [L]

RewriteCond %{REQUEST_FILENAME} !/bitrix/routing_index.php$
RewriteRule ^(.*)$ /bitrix/routing_index.php [L]
```

Файлы с конфигурацией маршрутов располагаются в папках `/bitrix/routes/` и `/local/routes/`. Для подключения файла следует описать его в файле .settings.php в секции *routing*:

```
'routing' => ['value' => [
  'config' => ['web.php', 'api.php']
]],

// подключатся файлы:
// /bitrix/routes/web.php, /local/routes/web.php,
// /bitrix/routes/api.php, /local/routes/api.php
```

Формат файла предполагает возврат замыкания, в которое передается объект конфигурации маршрутов:

```
use Bitrix\Main\Routing\RoutingConfigurator;

return function (RoutingConfigurator $routes) {
    // маршруты
};
```

Описание маршрутов начинается с определения метода запроса. Поддерживаются 3 комбинации методов:

```
$routes->get('/countries', function () {
    // сработает только на GET запрос
});

$routes->post('/countries', function () {
    // сработает только на POST запрос
});

$routes->any('/countries', function () {
    // сработает на любой тип запроса
});
```

Для указания произвольного набора методов следует использовать метод **methods**:

```
$routes->any('/countries', function () {
    // сработает на любой тип запроса
})->methods(['GET', 'POST', 'OPTIONS']);
```

Для определения параметра в адресе используются фигурные скобки:

```
$routes->get('/countries/{country}', function ($country) {
    return "country {$country} response";
});
```

По умолчанию для параметров используется паттерн `[^/]+`. Для указания своего критерия используется метод маршрута `where`:

```
$routes->get('/countries/{country}', function ($country) {
    return "country {$country} response";
```

```
})->where('country', '[a-zA-Z]+');
```

Для удобства и систематизации списка маршрутов присвойте маршруту уникальный идентификатор - имя:

```
$routes->get('/path/with/name', function () {
    return 'path with name';
})->name('some_name');
```

В роутинге поддерживается несколько видов контроллеров:

- Контроллеры Bitrix\Main\Engine\Controller:

```
$routes->get('/countries', [SomeController::class, 'view']);

    // будет запущено действие SomeController::viewAction()
```

- Отдельные действия контроллеров *Bitrix\Main\Engine\Contract\RoutableAction*:

```
$routes->get('/countries', SomeAction::class);
```

- Замыкания:

```
$routes->get('/countries/', function () {
        return "countries response";
    });
```

    В качестве аргументов возможно указать объект запроса Bitrix\Main\HttpRequest, объект текущего маршрута *Bitrix\Main\Routing\Route*, а также именованные параметры маршрута в любой комбинации:

```
use Bitrix\Main\HttpRequest;
use Bitrix\Main\Routing\Route;

$routes->get('/countries/{country}', function ($country, HttpRequest
$request) {
    return "country {$country} response";
});

$routes->get('/countries/{country}', function (Route $route) {
```

```
    return "country {$route->getParameterValue('country')} response";
});
```

- Для обратной совместимости с публичными страницами предусмотрен класс *Bitrix\Main\Routing\Controllers\PublicPageController*:

```
$routes->get('/countries/', new PublicPageController('/countries.php'));
```

99. Bitrix: Приложения и контекст

https://dev.1c-bitrix.ru/learning/course/index.php?COURSE_ID=43&LESSON_ID=3511&LESSON_PATH=3913.3516.5062.3511

Приложение - это объект, отвечающий за инициализацию ядра. Приложение является базовой точкой входа (маршрутизатором) для обращения к глобальным сущностям ядра: соединение с источниками данных, управляемый кеш и т.п. Также приложение содержит глобальные данные, которые относятся к самому сайту и не зависят от конкретного хита. То есть, приложение является неизменяемой частью, не зависящей от конкретного хита.

Любой конкретный класс приложения является наследником абстрактного класса Bitrix\Main\Application.

Конкретный класс Bitrix\Main\HttpApplication отвечает за обычный http-хит на сайте.

Приложение поддерживает шаблон Singleton (Одиночка). Т.е. в рамках хита существует только один экземпляр конкретного типа приложения. Его можно получить инструкцией

```
$application = Application::getInstance();
```

Контекст - это объект, отвечающий за конкретный хит. Он содержит запрос текущего хита, ответ ему, а также серверные параметры текущего хита. То есть это изменяемая часть, зависящая от текущего хита.

Любой конкретный класс контекста является наследником абстрактного класса Bitrix\Main\Context. Осуществляется поддержка двух конкретных классов контекста: *Bitrix\Main\HttpContext* и *Bitrix\Main\CliContext*. Конкретный класс *Bitrix\Main\HttpContext* отвечает за обычный http-хит на сайте.

Чтобы получить контекст выполнения текущего хита, можно воспользоваться кодом

```
$context = Application::getInstance()->getContext();
```

Если было инициализировано приложение типа *Bitrix\Main\HttpApplication*, то этот вызов вернет экземпляр контекста типа *Bitrix\Main\HttpContext*.

Контекст содержит в себе запрос текущего хита. Для того, чтобы получить запрос, можно воспользоваться кодом:

```
$context = Application::getInstance()->getContext();
$request = $context->getRequest();
```

Запрос представляет собой экземпляр класса, являющегося наследником класса Bitrix\Main\Request. В случае обычного http-запроса запрос будет являться экземпляром класса Bitrix\Main\HttpRequest, расширяющего *Bitrix\Main\Request*. Этот класс является по сути словарем, предоставляющим доступ к парам "ключ-значение" входящих параметров.

Для того чтобы обратиться к входящему параметру, переданному методами GET или POST, можно использовать код:

```
$value = $request->get("some_name");
$value = $request["some_name"];
```

100. Bitrix: События. События в D7

https://dev.1c-bitrix.ru/learning/course/index.php?COURSE_ID=43&CHAPTER_ID=03493&LESSON_PATH=3913.3516.3493

https://dev.1c-bitrix.ru/learning/course/index.php?COURSE_ID=43&LESSON_ID=3113&LESSON_PATH=3913.3516.3493.3113

Иногда бывает необходимо повлиять на ход выполнения какой-нибудь API функции. Но если ее изменить, то эти изменения будут утеряны при очередном обновлении. Для таких случаев и разработана **система событий**. В ходе выполнения некоторых API функций, в определённых точках установлены вызовы определённых функций, так называемых **обработчиков события**.

Какие функции-обработчики должны быть вызваны в каком месте (при каком событии) - нужно устанавливать вызовом функции, регистрирующей обработчики. В данный момент их две: Bitrix\Main\EventManager::addEventHandler и Bitrix\Main\EventManager::registerEventHandler.

В **D7**, по сравнению со старым ядром, снижены требования к данным, которые должен иметь код, порождающий событие. Пример отправки события:

```
$event = new Bitrix\Main\Event("main", "OnPageStart");
$event->send();
```

При необходимости есть возможность на стороне, отправляющей событие, получить результат обработки события принимающими сторонами.

```
foreach ($event->getResults() as $eventResult)
{
    switch($eventResult->getType())
    {
        case \Bitrix\Main\EventResult::ERROR:
            // обработка ошибки
            break;
        case \Bitrix\Main\EventResult::SUCCESS:
            // успешно
            $handlerRes = $eventResult->getParameters(); // получаем то, что
вернул нам обработчик события
            break;
        case \Bitrix\Main\EventResult::UNDEFINED:
            /* обработчик вернул неизвестно что вместо объекта класса
\Bitrix\Main\EventResult
            его результат по прежнему доступен через getParameters
            */
            break;
    }
}
```

Для уменьшения количества кода могут быть созданы наследники класса *Bitrix\Main\Event* для специфических типов событий. Например, *Bitrix\Main\Entity\Event* делает более комфортной отправку событий, связанных с модификацией сущностей.