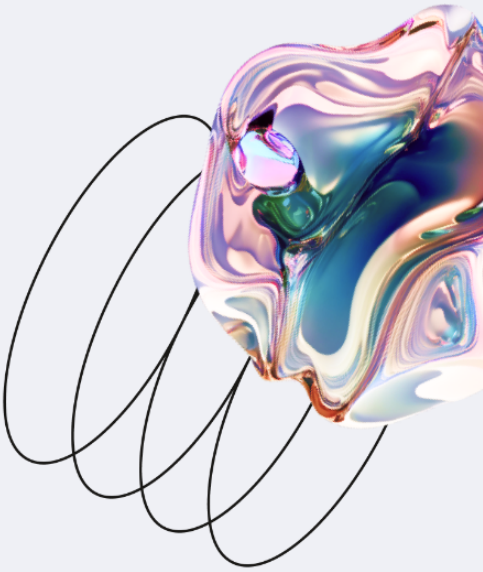
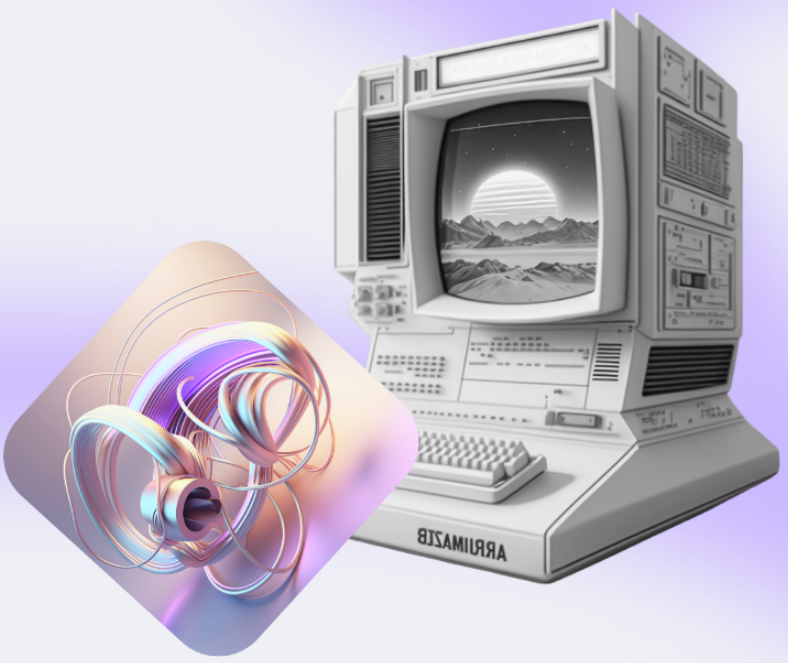
GeekBrains



Работа с БД

Основы PHP



Оглавление

Введение

Мы уже написали приложение, которое сохраняет данные. Ведь как вы помните, между запусками скриптом самостоятельно PHP имеет довольно ограниченные возможности по сохранению данных.

Для хранения данных нам обязательно нужна структурность – иначе наши программы не смогут понять, где лежат данные о пользователях, а где о товарах;

какая часть строки является артикулом товара, а какая – ценой. И базовую структуру мы уже получили в файловом хранилище. Мы точно знаем, что в нашем файле хранятся дни рождения пользователей, а внутри построчно перечислены кортежи вида «Имя Фамилия, день рождения».

Для решения простой задачи это вполне допустимый инструмент. Но в современном мире приложения хранят гораздо более сложные структуры. Предположим, что у нас есть система для обслуживания кинотеатра. Ей нужно знать не только о пользователях, но о том, какие фильмы идут сейчас, какое у них расписание, какие залы есть в кинотеатре, сколько в них мест, сколько стоит билет на каждый сеанс.

Вы уже знакомы с концепцией баз данных и понимаете, что все перечисленные сущности нельзя сохранить в одном файле. Более того, объём данных будет непрерывно расти. Представьте, что в системе одного кинотеатра есть пять залов, в каждом из которых есть по 300 посадочных мест. При этом в каждом зале в день проходит по 8 сеансов разных фильмов.

Это означает, что даже при продаже половины билетов на каждый сеанс, в день придётся фиксировать записи 6000 билетов, не говоря уже об информации об оплатах и тд.

И дальше нам нужно будет:

- посчитать продажи билетов
- знать, какие фильмы были самыми прибыльными

И так далее. При хранении в файлах нам придётся обходить целиком все файлы с данными, чтобы посчитать результат. А это крайне долго и дорого.

Значит, нам нужно решение, которое будет устраивать нас. Такая система должна:

- надежно хранить большие (более 1 млн записей) объёмы данных
- уметь выполнять по ним поиск
- быстро искать данные

И именно для решения такой задачи мы начинаем применять в нашем приложении СУБД. Мы будем ориентироваться на наиболее популярную СУБД для web-разработки – MySQL.

Словарь терминов

MySQL – это свободная и открытая СУБД, которая широко используется во множестве приложений и сетевых сервисов.

MySQL – это низкоуровневая библиотека.

PDO – это объектно-ориентированная библиотека, которая более безопасна для работы.

Исключение в программировании – это механизм обработки ошибок и исключительных ситуаций, которые могут возникнуть во время выполнения программы.

SQL-инъекция (SQL injection) – это вид атаки на веб-приложения, при которой злоумышленник внедряет вредоносный SQL-код в строку SQL-запроса, которая выполняется на стороне сервера базы данных.

MySQL

> 💡 MySQL – это свободная и открытая СУБД, которая широко используется во множестве приложений и сетевых сервисов.

MySQL была разработана шведской компанией MySQL AB, а в настоящее время ее разработкой занимается Oracle Corporation.

MySQL использует язык SQL (Structured Query Language) для выполнения запросов к хранилищу с целью получения данных.

MySQL широко применяется в веб-приложениях и веб-сервисах по нескольким причинам:

- Простота использования: MySQL предлагает простой и понятный интерфейс, что делает его доступным для разработчиков с разным уровнем опыта. SQL-запросы понятны и легко формулируются для извлечения и изменения данных.
- Высокая производительность: MySQL обладает высокой скоростью выполнения запросов и обработки больших объемов данных. Она оптимизирована для обработки многопользовательских запросов и имеет эффективные алгоритмы для ускорения выполнения операций с данными.

- Масштабируемость: MySQL поддерживает горизонтальное и вертикальное масштабирование. Горизонтальное масштабирование позволяет распределить данные по нескольким серверам, что повышает производительность и обеспечивает отказоустойчивость. Вертикальное масштабирование предоставляет возможность увеличить вычислительные ресурсы сервера, чтобы справиться с растущим объемом данных и запросов.
- Гибкость: MySQL совместима с различными операционными системами и языками программирования. Это позволяет разработчикам интегрировать MySQL в различные веб-приложения, независимо от выбранной платформы разработки.
- Поддержка сообщества: MySQL имеет огромное активное сообщество разработчиков, которое предоставляет богатый набор ресурсов, документации и помощи. Это обеспечивает быстрое решение проблем и доступ к новым функциям и улучшениям.

Добавляем СУБД в окружение

У нас уже есть настроенное окружение, в котором участвуют контейнеры с nginx и php-fpm. Разумеется, для MySQL нам нужен будет отдельный контейнер.

Как вы наверняка помните, контейнер не гарантирует сохранность информации после остановки. Но ведь база данных должна быть надежным хранилищем. Иногда говорят – **персистентным**. Это такое хранилище, которое обеспечивает сохранение данных на долгосрочной основе, даже после перезапуска или выключения системы.

Иначе говоря, MySQL должна уметь сохранять состояние своих данных даже при перезапуске контейнера. А это означает, что нам потребуется volume, в котором будут храниться файлы самой БД.

Также при первой сборке нашего образа нужно будет указать название базы данных, которой будет пользоваться наше приложение, имя пользователя и пароль. Конечно же, их можно указать в docker-compose явно, но это не является правильным и безопасным подходом. Это чувствительные (sensitive) данные, которые нужно хранить отдельно.

Благо, docker поддерживает файлы окружения, которые не принято коммитить в явном виде в репозиторий. Это подход с файлами .env (dot-env).

Файл .env в контексте Docker представляет собой файл конфигурации, который содержит переменные окружения (environment variables), используемые в вашем

контейнере Docker. Файл .env часто используется для централизованного хранения конфиденциальных данных и параметров настройки, таких как пароли, ключи API и другие конфиденциальные данные.

Формат файла .env простой: каждая строка содержит переменную окружения в формате "КЛЮЧ=ЗНАЧЕНИЕ":

```
DB_HOST=localhost

DB_USER=myuser

DB_PASSWORD=mypassword
```

В docker-compose эти переменные вызываются следующим образом:

```
services:

  myservice:

    environment:

      - DB_HOST=${DB_HOST}

      - DB_USER=${DB_USER}

      - DB_PASSWORD=${DB_PASSWORD}
```

Согласно документации наш блок информации о контейнере с MySQL будет выглядеть следующим образом:

```
#Контейнер с БД
  database:
    image: mysql:5.7
    container_name: database # имя контейнера после запуска
    environment:
      MYSQL_DATABASE: ${DB_NAME} # имя нашей БД
      MYSQL_USER: ${DB_USER} # имя пользователя, с которым будет подключаться
      MYSQL_PASSWORD: ${DB_PASSWORD} # пароль для пользователя
      MYSQL_ROOT_PASSWORD: ${DB_ROOT_PASSWORD} # администраторский пароль
    ports:
      - "3306:3306"
    volumes:
        - ./wdb:/var/lib/mysql
    networks:
      - app-network
```

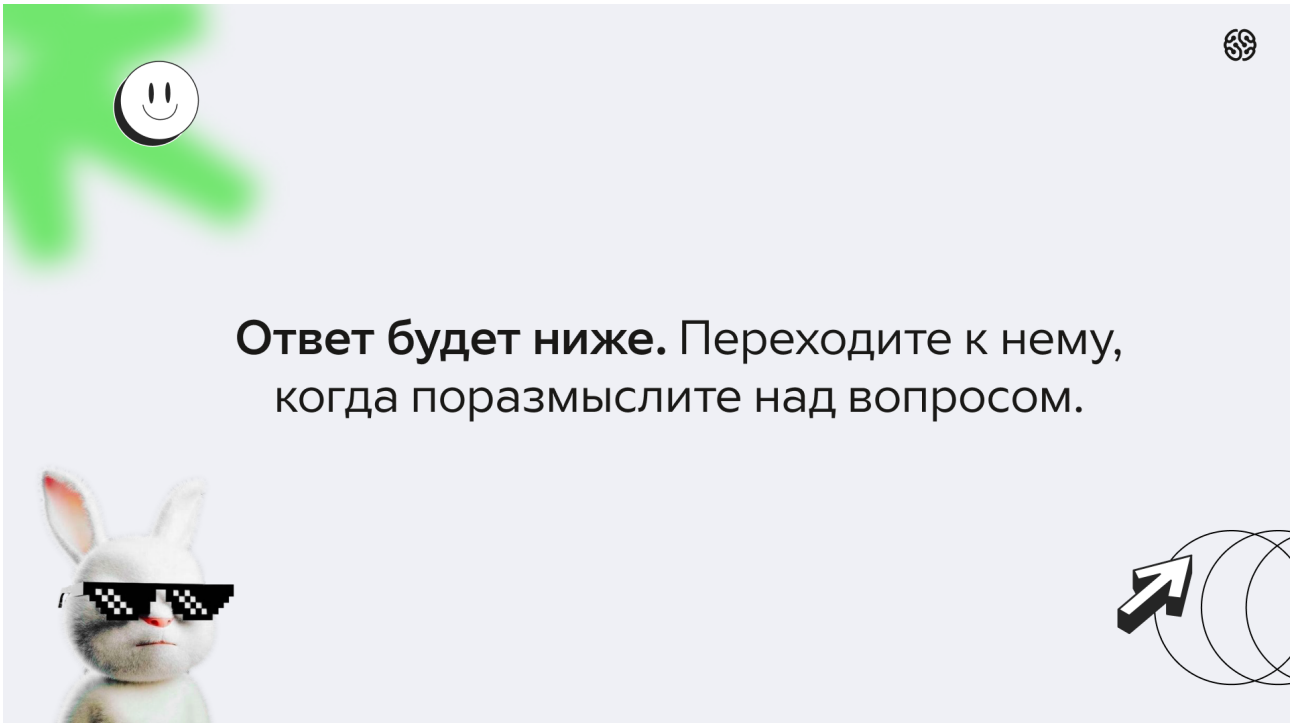
И рядом с docker-compose файлом создадим файл .env:

```
DB_NAME=application1
DB_USER=application_user
DB_PASSWORD=geekbrains!23
DB_ROOT_PASSWORD=GeekBrains&56
```

Разумеется, старайтесь использовать сложные пароли.

Если вы используете системы контроля версий, то сразу же добавьте файл .env в список игнорируемых, так как в зависимости от уровня окружения (например, окружение разработки и продуктив) пароли могут и должны меняться.

Теперь после запуска у нас появится наша база данных. Но как посмотреть, что в ней есть?



Для разработки нам потребуется клиент.

Многие ресурсы часто рекомендуют устанавливать PHPMyAdmin и другие web-клиенты. И хотя он и предоставляет графический интерфейс для выполнения различных операций, разработчики предпочитают избегать использования phpMyAdmin в производственных средах или ограничивают его доступность.

- Безопасность: PhpMyAdmin имеет потенциальные уязвимости безопасности, и его использование может представлять риск, особенно если не приняты соответствующие меры безопасности. Открытый доступ к phpMyAdmin может предоставить злоумышленникам возможность атаковать вашу базу данных.

Поэтому, если вы все же решите использовать phpMyAdmin, необходимо принять дополнительные меры для обеспечения безопасности, такие как ограничение доступа по IP, использование SSL и другие средства защиты.

- Дополнительная сложность и настройка: Включение phpMyAdmin требует настройки и поддержки дополнительного компонента в вашей инфраструктуре. Это может усложнить процесс развертывания и управления приложением. В некоторых случаях, особенно при использовании контейнеров, есть альтернативные варианты, которые могут быть более простыми и легкими в использовании, например, административные интерфейсы, предоставляемые самими фреймворками или библиотеками ORM.
- Ресурсоемкость: PhpMyAdmin может потреблять дополнительные ресурсы, как системные, так и сетевые. Это может оказывать негативное влияние на производительность вашей системы, особенно если используется большое количество параллельных соединений или баз данных с большим объемом данных.

Поэтому мы с вами будем применять нативный MySQL клиент. Это могут быть:

- MySQL Workbench
- Navicat
- HeidiSQL

Для примера подключимся к нашей БД через MySQL Workbench. Для этого после установки программы нам нужно будет в меню Database выбрать пункт Connect to database.

Далее заполняем поля следующим образом:

- Host: localhost
- Port: 3306
- Username: application_user
- Password: geekbrains!23

После этого соединяемся с БД, кликнув на иконку нашего соединения на главном экране. Здесь мы уже увидим во вкладке Schemas нашу созданную БД application1. Она пока пуста, но её наполнением мы займемся чуть позже.

Подключаемся из PHP

> 💡 Язык PHP готов к работе с СУБД mysql «из коробки» – это означает, что вам не нужно устанавливать никакие дополнительные модули для работы с выбранной нами СУБД.

Обратите внимание на то, что для других СУБД (таких как Postgres или Oracle) дополнительные модули все же придётся устанавливать).

Итак, как же можно соединиться с MySQL?

Для этого есть два пути:

- mysqli – это низкоуровневая библиотека
- PDO – это объектно-ориентированная библиотека, которая более безопасна для работы

Мы сразу будем работать с PDO, так как на данном этапе нам в первую очередь важна безопасность.

Для соединения с БД мы создаем новый объект типа PDO:

```
$connection = new PDO (
    'mysql:dbname=application1;host=database',
    'application_user',
    'geekbrains!23'
);
```

Где мы указываем:

- тип соединения, базу данных и хост
- имя пользователя
- пароль

> 🔥 Обратите внимание, что при использовании Docker контейнеров в параметре host должно находится название контейнера из docker-compose, так как контейнер с БД объединен с другими в одну внутреннюю виртуальную сеть.

В реальном приложении данные параметры следует брать из файла конфигурации, как и другие параметры подключения ко внешним ресурсам.

В переменной connection при успешном подключении будет хранится ссылка на подключение к БД. Однако, это только подключение, а нас интересует также возможность делать запросы.

Слои приложения

Мы уже делим наше приложение на слои – MVC. Но, как видите, даже в таком делении у нас появились классы Application и Router, которые нельзя однозначно отнести к M, V или C. Давайте разберемся, почему это происходит, прежде чем выделять новые слои для базы данных.

В целом слои следуют принципу инкапсуляции, защищая, то, что находится под ними, подобно фильтру. Слой говорит, какие данные и методы могут подниматься с глубоких слоев наверх.

При этом слои делят ответственность и правила размещения классов в коде. Благодаря этому легко найти нужный класс или разместить новый, зная его назначение.

Старые программы при этом часто содержат код, который можно назвать "спагети" кодом: можно вызывать и использовать всё что хочется, любые методы и структуры в любой части проекта. Применяя слои, мы стремимся к разделению ответственности (separation of concerns).

Рекомендуется выделять следующие слои:

- Домен
- Приложение
- Инфраструктура

Пока для нас домен – это все то, что содержится в директориях Models, Views и Controllers. То есть, то, что отвечает за логику пользовательского вывода.

Прикладной слой – это Application и Router. Здесь мы понимаем, какая команда к нам пришла, и что должно быть сделано.

Инфраструктурный слой содержит код, который взаимодействует с окружающим миром. Тут может быть код для:

- Общения с БД
- Отправки email-ов
- Генерации случайных чисел

Давайте немного доработаем наш код, чтобы у нас выделились слои. В директории src создадим директории:

- Domain
- Application
- Infrastructure

Директории Models, Controllers и Views переместим в Domain. Не забудьте при этом исправить определения и обращения к namespaces, так как в адресе появилась новая директория. Например, для класса Application старый namespace:

```
namespace Geekbrains\Application1;
```

сменится на новый:

```
namespace Geekbrains\Application1\Application;
```

Также нам надо будет чуть поменять наш класс Application. Ведь там у нас есть константа APP_NAMESPACE, которая теперь примет значение:

```
private const APP_NAMESPACE =
'Geekbrains\Application1\Domain\Controllers\\';
```

Также изменится свойство в классе Render:

```
private string $viewFolder = '/src/Domain/Views/';
```

Однако, обратите внимание на то, как легко мы смогли внести достаточно существенное изменение в виде корректировки структуры нашего приложения. Это и есть заслуга правильной архитектуры.

Теперь мы можем вернуться к нашему инфраструктурному слою.

Инфраструктурный слой приложения

Поскольку мы уже умеем подключаться к БД, нам стоит создать класс работы с базой, который будет представлять из себя обертку над PDO. Для чего это нужно? Дело в том, что мы можем в какой-то момент захотеть изменить PDO на какой-то другой класс (например, применим другую библиотеку). Если мы будем напрямую обращаться из моделей в PDO, такая замена будет стоить нам очень дорого – нам придётся ходить по всем местам вызова и менять логику.

При нашем же подходе мы будем контролировать применение PDO на уровне только одного класса.

Также нам нужно будет создать класс работы с конфигурацией приложения. С него мы и начнем. Создадим директорию config и файл config.ini в директории src. Это уже знакомый нам ini-файл:

```
[database]

DSN = "mysql:dbname=application1;host=database"

USER = "application_user"

PASSWORD = "geekbrains!23"
```

Теперь в src/Infrastructure мы создадим класс Config:

```
namespace Geekbrains\Application1\Infrastructure;

class Config {

    private string $defaultConfigFile = "/src/config/config.ini";

    private array $applicationConfiguration = [];

    public function __construct(){
        $address = $_SERVER['DOCUMENT_ROOT'] . $this->defaultConfigFile;

        if(file_exists($address) && is_readable($address)){
            $this->applicationConfiguration = parse_ini_file($address, true);
        }
        else {
            throw new \Exception("Файл конфигурации не найден");
        }
    }

    public function get(): array {
```

```
        return $this->applicationConfiguration;
    }
}
```

В конструкторе он будет пытаться прочитать файл. И если он не доступен, то мы получим исключение.

Исключения

> 💡 Исключение в программировании – это механизм обработки ошибок и исключительных ситуаций, которые могут возникнуть во время выполнения программы.

Когда возникает исключительная ситуация, такая как ошибка или непредвиденное поведение, программа может создать исключение и выбросить его (throw), а затем другая часть программы может перехватить это исключение и выполнить соответствующую обработку (catch).

В PHP исключения обрабатываются с помощью блоков try, catch и finally. Блок try содержит код, который может вызвать исключение. Блок catch определяет, как обработать выброшенное исключение, указывая тип исключения, которое нужно перехватить, и блок кода для обработки. Блок finally содержит код, который будет выполнен в любом случае, независимо от того, возникло исключение или нет.

Поэтому давайте создадим код, который будет ловить наше исключение.

Конфигурация приложения логично должна размещаться в классе Application. Поэтому мы создадим в классе новое статическое свойство:

```
public static Config $config;
```

Далее определим конструктор, где и будем создавать экземпляр конфигурации, чтобы мы могли обратиться к нему в любой точке нашего приложения:

```
public function __construct(){
    Application::$config = new Config();
}
```

Таким образом, исключение может быть сгенерировано в момент создания объекта Application. Это происходит в файле index.php. Поэтому и там мы немного изменим код:

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

use Geekbrains\Application1\Application\Application;

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

Теперь если наш файл конфигурации будет недоступен, то мы получим об этом сообщение, но выполнение программы не закончится ошибкой. То есть, мы правильно обработали ее.

Продолжаем формировать инфраструктурный слой

Вернемся к классу Config. Там мы создали метод get, который будет возвращать массив, сформированный из нашего ini-файла. Нам остается только правильно вызывать его при создании подключения к БД. Поэтому мы создаем в src/Infrastructure файл Storage.php:

```
namespace Geekbrains\Application1\Infrastructure;

use Geekbrains\Application1\Application\Application;
use \PDO;

class Storage {

    private PDO $connection;

    public function __construct() {
        $this->connection = new PDO(
            Application::$config->get()['database']['DSN'],
            Application::$config->get()['database']['USER'],
            Application::$config->get()['database']['PASSWORD']
            array(
                PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8"
            )
    );
    }
}
```

Как видите, теперь создание объекта подключения к БД не только полностью изолировано в классе, но и конфигурируется при помощи ini-файлов.

Также последним параметром мы явно задаем кодировку общения с СУБД. Согласно конвенциям, это будет UTF-8.

Еще одной важной деталью будет потребность в установке PDO-драйвера для MySQL в контейнеры cli и fpm. Это делается очень просто – мы добавляем в соответствующие Dockerfile по одной строке:

```
RUN docker-php-ext-install pdo_mysql
```

Настало время перенести наши данные из файлового хранилища в БД.

Переносим данные из файла

Мы хранили данные в достаточно простой структуре. Пользователь, его имя и день рождения с точки зрения отношений между данными будут иметь форму **«один-к-одному»**. Поэтому для переноса файлового хранилища в БД нам потребуется пока только одна таблица в созданной нами базе.

Назовем её Users. В ней мы будем хранить имя пользователя и день рождения. Но не стоит здесь забывать про нормализацию данных. С точки зрения нормальных форм нам надо будет сделать пару доработок.

Во-первых, каждая запись в таблице должна иметь уникальный ключ. То есть, нам нужен будет некий ID пользователя, который будет однозначно указывать на конкретный кортеж данных. Благо, в MySQL есть возможность создавать такие ключи встроенными средствами в виде автоинкрементального ключа.

Во-вторых, имя и фамилию стоит хранить в разных столбцах нашей будущей таблицы, так как они представляют собой разделяемые сущности в имени пользователя.

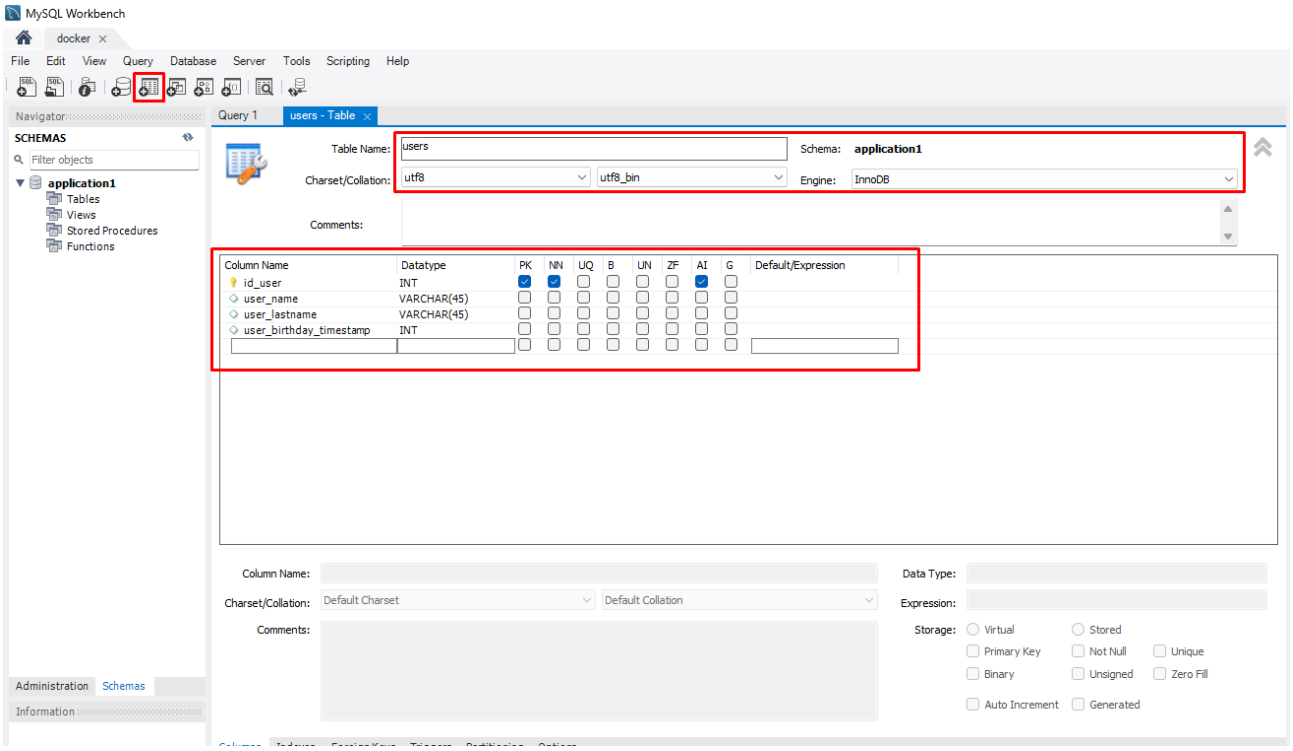
Таким образом, у нас в таблице потребуются следующие поля:

- ID
- Имя
- Фамилия
- Дата рождения

Не менее важно определить типы будущих полей, так как реляционные базы данных требуют строгой типизации хранимых данных в схеме. Поэтому у нас получается следующая структура:

- ID, целлочисленный, автоинкремент, первичный ключ
- Имя, строковый
- Фамилия, строковый
- Дата рождения, целочисленный

Теперь мы можем перейти в интерфейс Workbench и создать таблицу. Это можно сделать как через графический интерфейс:



так и при помощи SQL запроса:

```
CREATE TABLE `application1`.`users` (
  `id_user` INT NOT NULL AUTO_INCREMENT,
  `user_name` VARCHAR(45) NULL,
  `user_lastname` VARCHAR(45) NULL,
  `user_birthday_timestamp` INT NULL,
  PRIMARY KEY (`id_user`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8;
```

Теперь мы можем начать создавать здесь пользователей при помощи запросов из наших скриптов.

Учимся делать запросы

Научимся записывать пользователей в таблицу. Из прошлого домашнего задания вы уже умеете создавать пользователя в файловом хранилище. Давайте сделаем такую же логику, но на уровне БД.

Начнем с контроллера. В уже созданном нами ранее UserController нам потребуется создать метод actionSave. В нем мы должны:

1. Проверить, что все необходимые данные есть
2. Создать экземпляр пользователя
3. Сохранить его в БД
4. Показать ответ

Начнем с проверки данных. Этот процесс называется валидацией. Сам контроллер не должен уметь валидировать данные. Поэтому в модели User мы создадим статический метод валидации:

```
public static function validateRequestData(): bool{
        if(
            isset($_GET['name']) && !empty($_GET['name']) &&
            isset($_GET['lastname']) && !empty($_GET['lastname']) &&
            isset($_GET['birthday']) && !empty($_GET['birthday'])
        ){
            return true;
        }
        else{
            return false;
        }
    }
```

Как видите, он проверяет наличие необходимых параметров в запросе. И если их не хватает, то возвращает false .

Теперь в контроллере мы можем обращаться к нему:

```
public function actionSave(): string {
        if(User::validateRequestData()) {
          }
        else {
            throw new \Exception("Переданные данные некорректны");
        }
    }
```

Как видите, у нас появились новые параметры. И это надо отразить в свойствах модели User:

```
private ?string $userName;

private ?string $userLastName;

private ?int $userBirthday;
```

Теперь имя и фамилия разделены. Сразу же создадим get и set методы для нового поля:

```
public function setLastName(string $userLastName) : void {
        $this->userLastName = $userLastName;
    }
public function getUserLastName(): string {
        return $this->userLastName;
    }
```

Возвращаемся к контроллеру. Нам надо создать пользователя из параметров запроса, которые уже гарантированно присутствуют. А это означает, что можно обернуть все в один метод:

```
if(User::validateRequestData()) {

            $user = new User();

            $user->setParamsFromRequestData();

}
```

Опишем этот метод в классе модели:

```
public function setParamsFromRequestData(): void {
        $this->userName = $_GET['name'];
        $this->userLastName = $_GET['lastname'];
        $this->setBirthdayFromString($_GET['birthday']);
    }
```

Мы создали объект. Теперь нужно научиться сохранять его в базе. В контроллере допишем строку:

```
public function actionSave(): string {
        if(User::validateRequestData()) {
            $user = new User();
```

```
            $user->setParamsFromRequestData();
            $user->saveToStorage();
        }
        else {
            throw new \Exception("Переданные данные некорректны");
        }
    }
```

Нам нужно снова вернуться в модель и описать логику сохранения данных. В классе User создаем метод saveToStorage:

```
public function saveToStorage(){
        $storage = new Storage();

        $sql = "INSERT INTO users(user_name, user_lastname,
user_birthday_timestamp) VALUES (:user_name, :user_lastname, :user_birthday)";

        $handler = $storage->get()->prepare($sql);
        $handler->execute([
            'user_name' => $this->userName,
            'user_lastname' => $this->userLastName,
            'user_birthday' => $this->userBirthday
        ]);
    }
```

В нем мы создаем объект Storage, внутри которого происходит подключение к БД. Далее в виде строки мы описываем SQL-запрос вставки данных. И тут мы остановимся подробнее.

SQL-инъекции

В валидаторе мы проверяем передаваемые параметры только на факт наличия. Но нужно взять за правило тот факт, что любые пользовательские данные представляют для приложения угрозу. В чем она состоит?

> 💡 SQL-инъекция (SQL injection) – это вид атаки на веб-приложения, при которой злоумышленник внедряет вредоносный SQL-код в строку SQL-запроса, которая выполняется на стороне сервера базы данных.

Это может произойти, когда приложение не выполняет достаточную фильтрацию или экранирование пользовательского ввода, позволяя злоумышленнику внедрять дополнительные команды и изменять логику SQL-запросов.

Реализация SQL-инъекции может иметь разные формы, но основная идея заключается во внедрении SQL-кода в пользовательский ввод, который затем

выполняется на сервере базы данных. Вот несколько примеров типичных способов реализации SQL-инъекции:

1. Внедрение SQL-кода через пользовательский ввод: Злоумышленник может внедрить SQL-код, добавив его к пользовательскому вводу, который передается в запрос к базе данных. Например, если веб-приложение имеет форму для поиска, где пользователь может ввести ключевое слово, злоумышленник может добавить вредоносный SQL-код в это поле, чтобы выполнить дополнительные команды.

2. Манипуляция параметрами запроса: Если веб-приложение использует параметризованные запросы, злоумышленник может попытаться изменить значение параметра, внедрив вредоносный SQL-код. Например, если приложение имеет параметр id в запросе SELECT, злоумышленник может попытаться ввести id = 1 OR 1=1, чтобы получить все записи из таблицы.

3. Внедрение SQL-кода через URL-параметры: Если приложение строит SQL-запросы, используя значения из URL-параметров, злоумышленник может внедрить SQL-код, добавив его в URL. Например, при запросе /products?id=1; DROP TABLE users, злоумышленник может попытаться удалить таблицу users из базы данных.

Последствия SQL-инъекции могут быть серьезными, включая потерю, внесение или изменение данных, получение несанкционированного доступа к базе данных и даже выполнение удаленного кода. Для защиты от SQL-инъекции веб-приложения должны использовать параметризованные запросы или подготовленные выражения, а также должны проводить проверку и экранирование пользовательского ввода.

Говоря проще, если мы будем напрямую вставлять пользовательские параметры в SQL-запрос, то тем самым мы будем подвергать приложение риску того, что выполнится не только ожидаемый нами запрос, но и побочные вредоносные действия.

Благо механизмы PDO уже предоставляют нам возможность избегать этих рисков.

Подготовленные выражения

Запрос, который мы формируем в переменной sql – это строка подготовленного запроса. Как видите, в нем нет явно указанных данных, а перечисляются placeholder-ы в части VALUES, на место которых PDO поставит нужные данные.

Для того, чтобы обработать запрос в безопасном виде, нам нужно вызвать метод prepare. Но он вызывается у объекта соединения. Поэтому в классе Storage мы создадим метод get, который будет возвращать ссылку на соединение:

```
public function get(): PDO {
        return $this->connection;
    }
```

Далее в модели мы передаем в подготовку строку нашего запроса:

```
$handler = $storage->get()->prepare($sql);
```

Теперь нам остаётся только выполнить запрос, передав ему реальные данные, которые нужно подставить в подготовленный запрос:

```
$handler->execute([
            'user_name' => $this->userName,
            'user_lastname' => $this->userLastName,
            'user_birthday' => $this->userBirthday
        ]);
```

Остается лишь создать пользовательский вывод. Для этого возвращаемся на уровень контроллера и завершаем логику метода сохранения:

```
public function actionSave(): string {
        if(User::validateRequestData()) {
            $user = new User();
            $user->setParamsFromRequestData();
            $user->saveToStorage();

            $render = new Render();

            return $render->renderPage(
                'user-created.tpl',
                [
                    'title' => 'Пользователь создан',
                    'message' => "Создан пользователь " . $user->getUserName() .
" " . $user->getUserLastName()
```

```
                ]);
        }
        else {
            throw new \Exception("Переданные данные некорректны");
        }
    }
```

Разумеется, нам потребуется также и шаблон для вывода информации о созданном пользователе. Файл user-created.tpl будет выглядеть очень просто:

```
<h3>{{ message }}</h3>
```

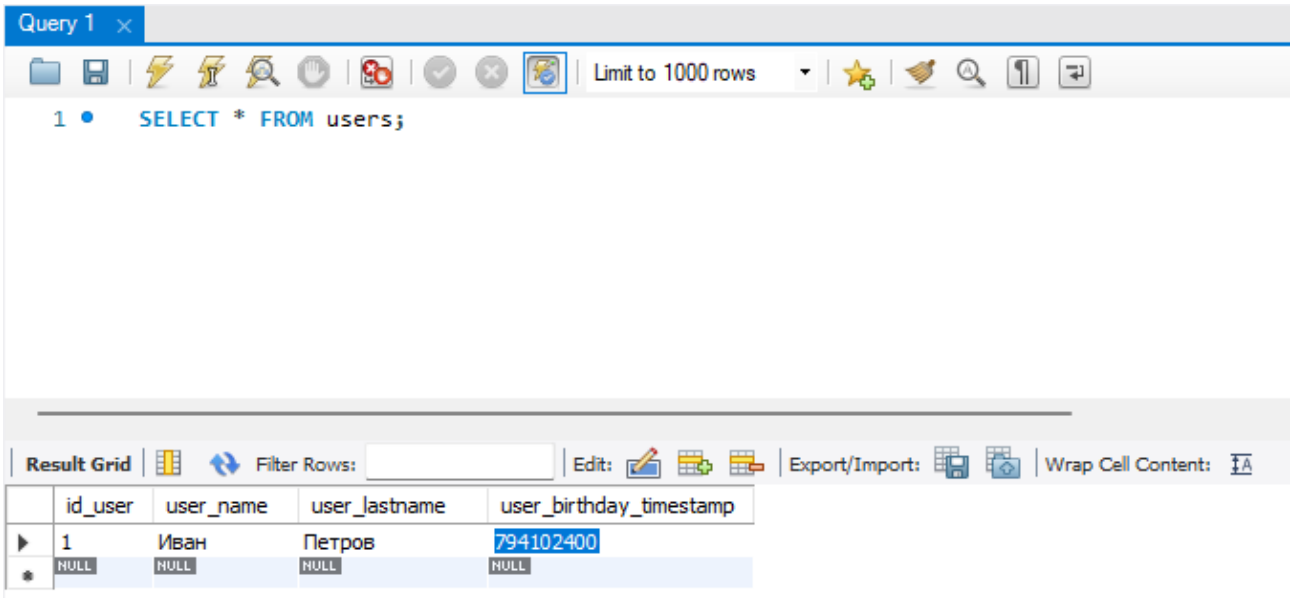
Если мы всё сделали правильно, то при вызове URL вида:

http://mysite.local/user/save/?name=Иван&lastname=Петров&birthday=02-03-1995

в браузере мы увидим ответ:

Создан пользователь Иван Петров

А в базе данных – строку:



Выборка пользователей

Поскольку у нас изменилась и структура модели, и хранилище, нам следует переписать и метод getAllUsersFromStorage в модели User. Обратите внимание, что наш контроллер снова спроектирован правильно – изменение модели не

заставляет нас менять что-либо в контроллере, а это значит, что классы имею слабую связность.

Поскольку запросами мы будем пользоваться часто, и одна страница сможет посылать несколько запросов, соединение стоит вынести на уровень приложения, чтобы не соединяться с БД каждый запрос.

Модифицируем код класса Application:

```
public static Storage $storage;


    public function __construct(){
        Application::$config = new Config();
        Application::$storage = new Storage();
    }
```

Вернемся в модель:

```
public static function getAllUsersFromStorage(): array|false {
        $sql = "SELECT * FROM users";
}
```

SQL-запрос будет простой выборкой данных. Для того, чтобы собрать полученные данные в массив, после выполнения запроса нам нужно будет применить метод fetchAll у обработчика. Задача этого метода преобразовать полученные из БД данные в ассоциативный массив:

```
$handler = Application::$storage->get()->prepare($sql);
$handler->execute();
$result = $handler->fetchAll();
```

Дальше, как и в случае с файловым хранилишем, мы обходим массив, создаем коллекцию пользователей и передаём её в шаблон:

```
    public static function getAllUsersFromStorage(): array {
        $sql = "SELECT * FROM users";

        $handler = Application::$storage->get()->prepare($sql);
        $handler->execute();
        $result = $handler->fetchAll();

        $users = [];

        foreach($result as $item){
            $user = new User($item['user_name'], $item['user_lastname'],
$item['user_birthday_timestamp']);
            $users[] = $user;
        }

        return $users;
    }
```

Поскольку структура модели изменилась, то и шаблон user-index.tpl нужно поправить, добавив вывод фамилии пользователя:

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

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

Заключение

В данной лекции мы научили наше приложение работать с актуальным хранилищем – реляционной базой данных. Также мы обеспечили безопасность работы с хранилищем. Помимо этого добавили нашей архитектуре новые слои и убедились в правильности её построения.

Домашнее задание

1. Мы стали работать с исключениями. Создайте в Render логику обработки исключений так, чтобы она встраивалась в общий шаблон. Вызов будет выглядеть примерно так

```
try{
    $app = new Application();
    echo $app->run();
}
catch(Exception $e){
    echo Render::renderExceptionPage($e);
}
```

2. Создайте метод обновления пользователя новыми данными. Например,

```
/user/update/?id=42&name=Петр
```

   Такой вызов обновит имя у пользователя с ID 42. Обратите внимание, что остальные поля не меняются. Также помните, что пользователя с ID 42 может и не быть в базе.

3. Создайте метод удаления пользователя из базы. Учитывайте, что пользователя может не быть в базе

```
/user/delete/?id=42
```

Что можно почитать еще?

1. https://phptherightway.com/
2. https://php.net
3. PHP 8 | Котеров Дмитрий Владимирович
4. Объектно-ориентированное мышление | Мэтт Вайсфельд
5. https://stackoverflow.com/questions/34034730/how-to-enable-color-for-php-cli - как покрасить вывод консоли
6. https://phptoday.ru/post/gotovim-lokalnuyu-sredu-docker-dlya-razrabotki-na-php - варианты образа Composer
7. https://anton-pribora.ru/articles/php/mvc/