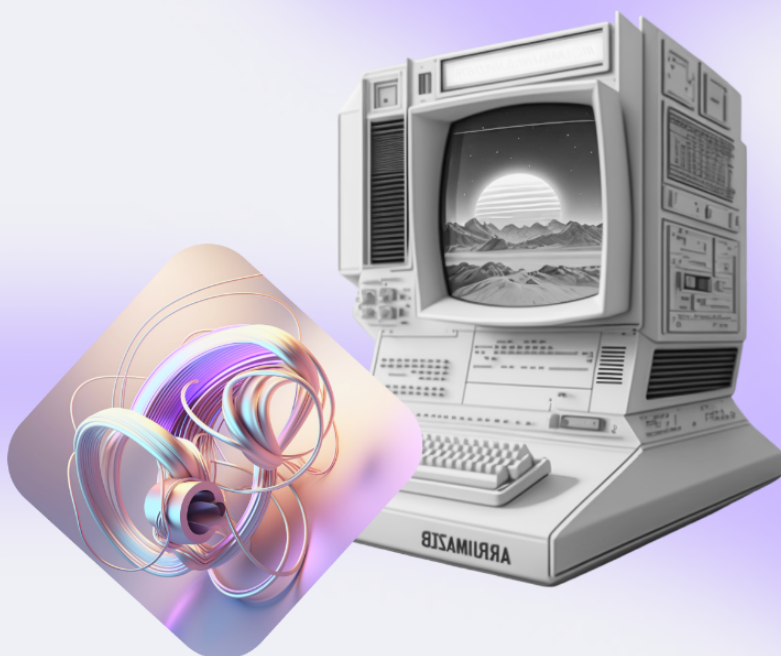


# Формы, авторизация и аутентификация

ОСНОВЫ PHP



# Оглавление

Введение	2
Обработка форм	3
Регулярные выражения	5
XSS атаки и защита от них	9
CSRF атаки и защита от них	11
Cookies	12
Сессии	13
Защита от CSRF атак	15
Идентификация пользователя на сайте	18
Пишем форму логина	19
Хранение паролей	19
Ролевая модель разграничения прав	24
Заключение	28
Домашнее задание	28
Что можно почитать еще?	28

## Введение

Продолжая изучать язык PHP, мы делаем наше приложение все сложнее и сложнее.

Поэтому в данной лекции мы:

- Узнаем, как организовать интерактивное взаимодействие с пользователем;
- Обеспечим разграничение прав;
- Научимся безопасной работе с авторизацией и аутентификацией.

До этого момента в нашем приложении мы создавали пользователя GET-запросом, указывая параметры прямо в URL. Но по ряду причин так делать не следует.

Во-первых, потому, что согласно спецификации HTTP GET-запрос не должен создавать новые сущности. Он может только получать данные с сервера.

Во-вторых, пользователю неудобно вбивать руками в URL какие-то параметры. Большинство пользователей (если не все) просто не станут так делать, а значит ваш сайт будет просто непригоден для работы.

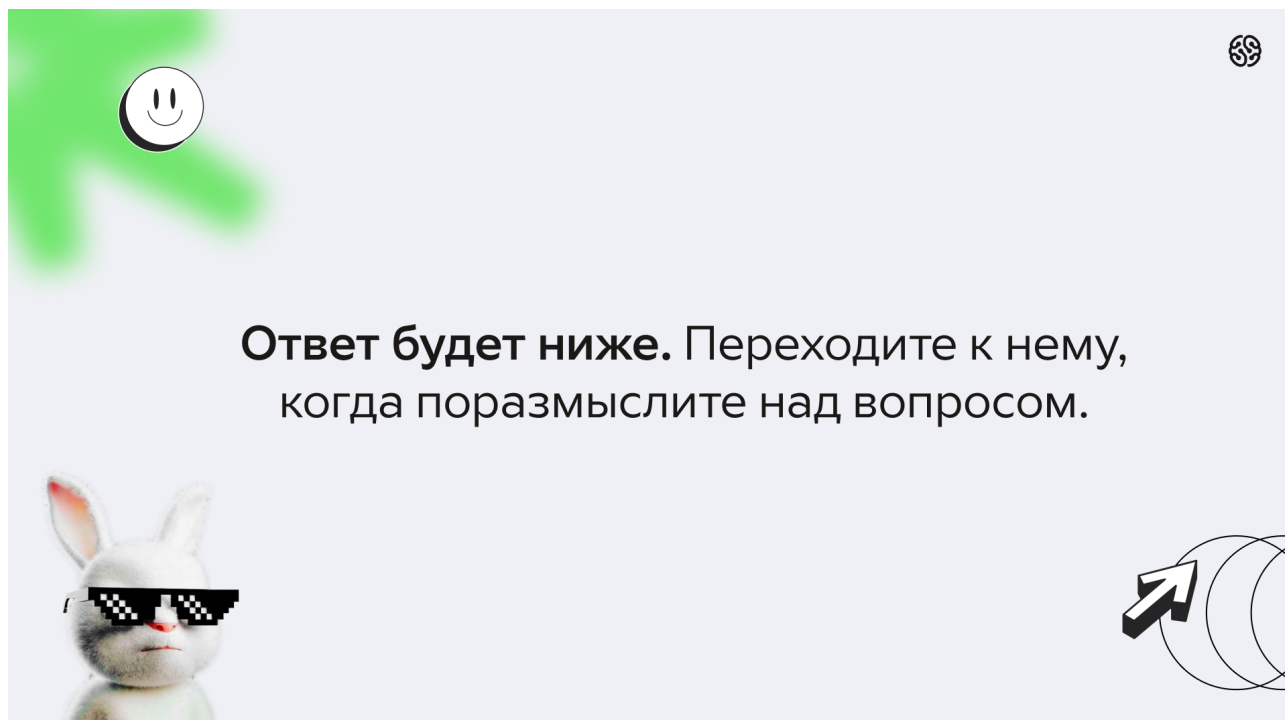
Но мы знаем, что язык HTML позволяет нам работать с формами, отправляя их данные на сервер для обработки. Поэтому мы будем погружаться в то, как можно применять формы. По сути, HTML-формы – это один из фундаментальных способов взаимодействия с пользователем в web-приложениях.

## Обработка форм

Вы наверняка помните, как выглядит HTML-форма:

```
<form action="">
  <label for="username">Name:</label>
  <input id="username" type="text" name="name">
  <input type="submit" value="Save">
</form>
```

В аспекте взаимодействия с PHP нас будет интересовать поле action. В примере выше оно пустое. И это означает, что даже, заполнив такую форму, пользователь ничего не создаст и не изменит на стороне сервера. Ведь в action хранится обработчик тех данных, которые ввел пользователь. Что должно быть указано в action для того, чтобы на сервере случился вызов логики?



Ответ: URI. То есть, адрес того контроллера, который мы хотим вызвать.

Выходит, что наше приложение уже большей своей частью готово к тому, чтобы работать с формами. В прошлой лекции мы создали возможность создавать сущности в нашей БД. Настало время сделать это удобным для пользователей нашего приложения.

Для начала создадим шаблон user-form.tpl:

```
<form action="/user/save/" method="post">
  <p>
    <label for="user-name">Имя:</label>
    <input id="user-name" type="text" name="name">
  </p>
  <p>
    <label for="user-lastname">Фамилия:</label>
    <input id="user-lastname" type="text" name="lastname">
  </p>
  <p>
    <label for="user-birthday">День рождения:</label>
    <input id="user-birthday" type="text" name="birthday"
placeholder="ДД-ММ-ГГГГ">
  </p>
  <p><input type="submit" value="Сохранить"></p>
</form>
```

В нем будет три поля, которые отражают все необходимые для нашего приложения сущности. Для этой формы создадим простой метод в контроллере:

```
public function actionEdit(): string {
    $render = new Render();
    return $render->renderPage(
        'user-form.tpl',
        [
            'title' => 'Форма создания пользователя'
        ]
    );
}
```

Обратите внимание на то, что форма будет генерироваться отдельным от сохранения методом контроллера. Если мы будем делать всё внутри одного обработчика, это перегрузит код, усложнит его поддерживаемость и читаемость.

Для сохранения мы оставим все тот же метод actionSave, но его надо будет модифицировать, ведь в форме мы указали метод передачи данных POST. Как мы проговорили в самом начале, GET-метод для создания пользователя не годится.

Поэтому нам надо поправить несколько методов в модели User. Для начала это будет метод валидации данных:

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

Здесь мы пока просто поменяли обращение к суперглобальному массиву.

То же самое сделаем и в методе задания параметров пользователя:

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

Теперь, заполнив форму, мы можем нажать клавишу «Сохранить» и нас перебросит на страницу успешного создания нового пользователя.

Как видите, само создание обработчика – довольно несложная задача. Но мы уже знаем, что любые данные от пользователя не заслуживают доверия. Поэтому нам надо уметь проверять их.

## Регулярные выражения

Если имя и фамилия пользователя – это простые строки, то вот к дате мы предъявляем достаточно строгий формат. Это день, месяц и год, разделенные знаком «дефис».

Как проверить, что переданная дата соответствует нужному нам шаблону?

Для этого в PHP существует механизм регулярных выражений.

Регулярные выражения представляют собой синтаксический способ описания шаблонов для поиска и сопоставления строк в тексте. Они являются мощным инструментом для работы с текстом, позволяя выполнять различные операции, такие как поиск подстрок, замена, разбиение текста на составляющие и проверка соответствия определенным правилам.

💡 Синтаксис регулярных выражений включает в себя комбинацию символов, метасимволов и операторов, которые позволяют задавать условия сопоставления и поиска.

Некоторые распространенные операции, которые можно выполнить с помощью регулярных выражений, включают поиск строки, начинающейся или заканчивающейся определенным образом, поиск одного или нескольких повторяющихся символов, поиск чисел, букв или специальных символов и многое другое.

💡 Например, регулярное выражение `"\d+"` будет соответствовать шаблону «любая последовательность одной или более цифр в тексте».

💡 Регулярное выражение `"^[A-Za-z]+$"` будет соответствовать строке, состоящей только из латинских букв.

Регулярные выражения могут быть сложными, особенно для неопытных пользователей, но они предоставляют мощный инструмент для работы с текстом и обработки данных, позволяя выполнять разнообразные операции поиска и сопоставления.

Общие правила построения регулярных выражений можно найти во многочисленных шпаргалках.


## Регулярные выражения

Якоря		Образцы шаблонов		
^	Начало строки +	([A-Za-z0-9-]+)	Буквы, числа и знаки переноса	
\A	Начало текста +	(\d{1,2}\d{1,2}\d{4})	Дата (напр., 21/3/2006)	
\$	Конец строки +	([^\s]+(?:\.(jpg gif png))\.\w{2})	Имя файла jpg, gif или png	
\Z	Конец текста +	(^[1-9]{1}\$ ^[1-4]{1}[0-9]{1}\$ ^[50]\$)	Любое число от 1 до 50 включительно	
\b	Граница слова +	(#[A-Fa-f0-9]{3}([A-Fa-f0-9]{3})?)	Шестнадцатичный код цвета	
\B	Не граница слова +	((?=[^\d])(?=[a-z])(?=[A-Z]).{8,15})	От 8 до 15 символов с минимум одной цифрой, одной заглавной и одной строчной буквой (полезно для паролей).	
\<	Начало слова	(\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,6})	Адрес email	
\>	Конец слова	(\<\/?[^\>]+\>)	HTML теги	
Символьные классы				
\c	Управляющий символ			
\s	Пробел			
\S	Не пробел			
\d	Цифра			
\D	Не цифра			
\w	Слово			
\W	Не слово			
\xhh	Шестнадцатичный символ hh			
\Oxxx	Восьмиричный символ xxx			
Символьные классы POSIX				
[[:upper:]]	Буквы в верхнем регистре			
[[:lower:]]	Буквы в нижнем регистре			
[[:alpha:]]	Все буквы			
[[:alnum:]]	Буквы и цифры			
[[:digit:]]	Цифры			
[[:xdigit:]]	Шестнадцатичные цифры			
[[:punct:]]	Пунктуация			
[[:blank:]]	Пробел и табуляция			
[[:space:]]	Пустые символы			
[[:cntrl:]]	Управляющие символы			
[[:graph:]]	Печатные символы			
[[:print:]]	Печатные символы и пробелы			
[[:word:]]	Буквы, цифры и подчеркивание			
Утверждения				
?=	Вперед смотрящее +			
?!	Отрицательное вперед смотрящее +			
?<=	Назад смотрящее +			
?!= или ?>	Отрицательное назад смотрящее +			
?>	Однократное подвыражение			
?()	Условие [если, то]			
?()	Условие [если, то, а иначе]			
?#	Комментарий			
Примечание		Эти шаблоны предназначены для ознакомительных целей и основательно не проверялись. Используйте их с осторожностью и предварительно тестируйте.		
Кванторы		Диапазоны		
*	0 или больше +	.	Любой символ, кроме переноса строки (\n) +	
*?	0 или больше, нежадный +	(a b)	a или b +	
+	1 или больше +	(...)	Группа +	
++	1 или больше, нежадный +	(?:...)	Пассивная группа +	
?	0 или 1 +	[abc]	Диапазон (a или b или c) +	
??	0 или 1, нежадный +	[^abc]	Не a, не b и не c +	
{3}	Ровно 3 +	[a-q]	Буква между a и q +	
{3,}	3 или больше +	[A-Q]	Буква в верхнем регистре между A и Q +	
{3,5}	3, 4 или 5 +	[0-7]	Цифра между 0 и 7 +	
{3,5}?	3, 4 или 5, нежадный +	\n	n-ая группа/подшаблон +	
Специальные символы		Примечание		
\	Экранирующий символ +	Диапазоны включают граничные значения.		
\n	Новая строка +	Модификаторы шаблонов		
\r	Возврат каретки +	g	Глобальный поиск	
\t	Табуляция +	i	Регистронезависимый шаблон	
\v	Вертикальная табуляция +	m	Многострочный текст	
\f	Новая страница +	s	Считать текст одной строкой	
\a	Звуковой сигнал	x	Разрешить комментарии и пробелы в шаблоне	
[\b]	Возврат на один символ	e	Выполнение подстановки	
\e	Escape-символ	U	Нежадный шаблон	
\N{name}	Именованный символ	Мета-символы (экранируются)		
Подстановка строк		^	[	.
\$n	n-ая непассивная группа	\$	{	*
\$2	«xyz» в /^(abc(xyz))\$/	(	\	+
\$1	«xyz» в /(?:abc)(xyz)\$/	)		?
\$'	Перед найденной строкой	<	>	
\$'	После найденной строки	Эта таблица доступна на <a href="http://www.exlab.net">www.exlab.net</a> Англоязычный оригинал на <a href="http://AddedBytes.com">AddedBytes.com</a>		
\$+	Последняя найденная строка			
&	Найденная строка целиком			
\$_	Исходный текст целиком			
\$	Символ «\$»			

Давайте попробуем составить регулярное выражение для проверки данных в нашем случае. Это должны быть по порядку:

- Две цифры
- Дефис
- Две цифры
- Дефис
- Четыре цифры


Разберем подробнее наше выражение.

 Символы ^ и \$ обозначают начало и конец строки, соответственно.

Если их не указать, то проверку сможет пройти любая строка, которая включает дату в нужном нам формате.

В скобках мы указываем шаблон даты:

- \d говорит, что мы ищем цифры
- {2} и {4} говорят, что нас интересует строго 2 или 4 цифры, соответственно
- Дефисы остаются, как есть, они не считаются, как специальные символы

 Значит, выражение будет выглядеть так:

**`^(\d{2}-\d{2}-\d{4})$`**

Кстати, проверить правильность регулярного выражения можно онлайн. Например, на сайте <https://regex101.com/>

Теперь мы должны подключить это регулярное выражение к нашему валидатору. Для этого нам нужна встроенная функция [preg\\_match](#). В качестве первого параметра она принимает регулярное выражение, а второго – строку, которую нужно проверить. Возвращает она либо количество найденных вхождений, либо false, если совпадения не были найдены.



Теперь наш код валидатора будет выглядеть так:


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

Обратите внимание на то, что внутри функции `preg_match` нужно обернуть регулярное выражение в два слеша. Если функция возвращает ложь, значит строка не прошла проверку. И нам передано некорректное значение.

Но мы здесь создали проверку от ошибочных действий хорошего пользователя. Злоумышленники могут формировать более серьезные атаки, о которых мы поговорим далее.

## XSS атаки и защита от них

Первый распространенный тип атаки на формы – это XSS.

 XSS (Cross-Site Scripting) — это тип атаки на веб-приложения, при котором злоумышленник внедряет вредоносный код (скрипты) на веб-страницу, которая затем выполняется в браузере пользователя.

Эта атака основывается на недостаточной фильтрации пользовательского ввода, который позволяет злоумышленнику внедрить и выполнить произвольный код на стороне клиента.

Существуют различные способы реализации XSS-атаки, но общая идея заключается в том, чтобы внедрить вредоносный код в веб-страницу, который будет исполняться в браузере других пользователей, посещающих эту страницу.

Мы с вами используем формы с текстовыми полями. И пока у нас на них нет никакой защиты со стороны приема пользовательских данных.


Давайте попробуем в поле имя ввести строку:

```
<script>alert('XSS! ');</script>
```

При сохранении избежать опасности нам поможет Twig, так как он превратит теги в браузерные символы. Но давайте предположим, что Twig у нас не работает. Модифицируем шаблон созданного пользователя так, чтобы он стал пропускать HTML-код. Для этого применим к значению message фильтр raw.

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

Теперь при создании пользователя мы увидим всплывающее сообщение. Разумеется, оно не несёт вреда само по себе, но определяет вектор атаки. Злоумышленники могут увидеть, что они могут внедрить более серьёзный вредоносный код.

 Целью их XSS-атаки может быть кража пользовательских данных, фишинг, перехват сессий, внедрение вредоносных скриптов или перенаправление на другие веб-сайты.

Для защиты от XSS-атак следует использовать очистку (санацию) ввода данных, правильную фильтрацию и экранирование специальных символов при отображении данных на веб-странице, а также использование контентной безопасности, такой как Content Security Policy (CSP).

У нас, как мы уже увидели, есть Twig, который частично защищает вывод. Но надо фильтровать и ввод.

Для предотвращения XSS-атак в языке PHP есть функция экранирования выводимых данных. Это htmlspecialchars(), которая преобразует специальные символы HTML в соответствующие HTML-сущности.

Таким образом, мы можем доработать наш метод создания пользователя:

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

Теперь наше приложение не подвергнется XSS-атаке.

# CSRF атаки и защита от них

Еще одним видом атаки является CSRF.



CSRF (Cross-Site Request Forgery) – это вид атаки на веб-приложения, при котором злоумышленник пытается выполнить нежелательные действия от имени пользователя.

Атакующий создает специально подготовленный запрос (часто в виде подменного URL или подменной HTML-формы), который отправляется на оригинальный сайт. Если жертва перейдет по подготовленной ссылке или отправит форму, то нежелательное действие будет выполнено.

То есть, злоумышленник создает копию формы у себя на сайте, подделывая внешний вид. При этом в action вшиваются данные запроса, которые нужны злоумышленнику. Например, в запросе передается команда на перевод денег на нужный счет или смену пароля на требуемый.

Для защиты от CSRF атак рекомендуется использовать механизмы, такие как:

1. Проверка происхождения запроса (origin checking): Веб-приложение должно проверять, что запросы поступают только с доверенных источников. Это можно сделать, например, путем добавления токена (CSRF token) в каждую форму или запрос, и проверки его при обработке запроса.
2. Использование SameSite cookies: SameSite - это атрибут для cookies, который ограничивает их отправку только на тот же сайт, с которого они были установлены, что помогает предотвратить CSRF атаки.
3. Запрос подтверждения действия (double-submit cookie): При каждой форме или запросе может быть добавлено дополнительное поле, содержащее значение, которое должно быть отправлено и проверено на сервере. Это значение должно быть связано с сеансом пользователя и храниться как cookie и как параметр запроса. Если значения не совпадают, то запрос может быть отклонен.
4. Использование CAPTCHA: Ввод CAPTCHA при выполнении важных операций может помочь предотвратить CSRF атаки, так как затрудняет автоматическое выполнение запросов злоумышленником.

🔥 Важно отметить, что защита от CSRF атак должна быть реализована на стороне сервера, поскольку клиентская сторона (браузер) может быть подвержена изменениям.

Но перед тем, как защищаться от этого типа атак на практике, нам нужно познакомиться с двумя важными механизмами в PHP.

## Cookies

💡 Cookies — это маленькие текстовые файлы на компьютере пользователя, в которых хранится некая информация от сервера.

Они могут хранить:

- предпочтения пользователей (язык, тему сайта)
- просмотренные товары
- дату и время последнего посещения

Когда пользователь выполняет какое-то действие, сервер обрабатывает его, а затем посылает браузеру информацию о том, что в cookies надо записать какие-то данные. Это отправляется в заголовках ответа. После этого с каждым HTTP-запросом пользователя браузер будет отправлять cookies на сервер.

Cookies бывают временными (они имеют срок жизни и по его истечению уничтожаются) и постоянными. Какие именно куки использовать на конкретном сайте решает его разработчик.

💡 Сами по себе cookies неопасны — это обычные текстовые файлы.

Они не будут запускать вредоносные процессы. Но они никак не шифруются, поэтому на компьютере пользователя их могут подменить. К примеру, если у пользователя на компьютере поселился вирус. Хотя, даже браузерное расширение может сделать это.

Давайте научимся пользоваться Cookies при помощи PHP!

Для установки cookie в PHP используется функция `setcookie()`.

Она принимает несколько параметров, включая имя, значение, срок действия в секундах, путь, домен и другие опции:

```
// Установка cookie на один час
setcookie('username', 'Иван Иванов', time() + 3600, '/');
```

Чтение значения cookie можно выполнить с помощью суперглобального массива `$_COOKIE`:

```
if (isset($_COOKIE['username'])) {
    $username = $_COOKIE['username'];
    echo "Привет, $username!";
} else {
    echo "Привет, гость!";
}
```

Обратите внимание, что в момент, когда вы только установите cookie, их еще не будет в наличии в суперглобальном массиве. Туда они попадут только со следующим запросом пользователя.

Чтобы обновить значение cookie, можно просто повторно использовать функцию `setcookie()`, указав новое значение. Например:

```
setcookie('username', 'Jane Smith', time() + 3600, '/');
```

Для удаления cookie необходимо установить время срока действия в прошлом:

```
setcookie('username', '', time() - 3600, '/');
```

Важно отметить, что для удаления cookie необходимо указать те же значения пути и домена, которые были использованы при установке cookie.

Cookies небезопасны и должны применяться только для того, чтобы помечать пользователя, но каждый раз проверять эти данные. Для этого нам понадобится ответный серверный механизм.

## Сессии



В PHP сессия — это механизм, который позволяет хранить информацию о пользователе на сервере в течение определенного периода времени, не размещая её в БД.

Он обеспечивает возможность отслеживать состояние пользователя и сохранять данные между разными запросами.

По сути, в сессии хранится набор в виде массива **«ключ-значение»**.

В момент, когда стартует сессия, PHP формирует уникальный идентификатор, который на клиенте сохраняется в виде cookies. Так пользователь говорит серверу, какая из хранимых сессий принадлежит именно ему. Это похоже на электронный ключ для входа в офис.

На стороне сервера генерируется файл, который содержит уникальный идентификатор.

Чтобы начать сессию в PHP, необходимо вызвать функцию `session_start()`. Это должно быть выполнено в самом начале скрипта, перед выводом любого контента на страницу. Эта функция как раз и создает уникальный идентификатор сессии для текущего пользователя и устанавливает куки с этим идентификатором.

После начала сессии вы можете сохранять и получать данные сессии. Данные сессии хранятся в суперглобальном массиве `$_SESSION`. Например, чтобы сохранить значение в сессии, вы можете просто присвоить его элементу массива `$_SESSION['key'] = 'value';`.

Для доступа к сохраненным данным просто обратитесь к элементу массива `$_SESSION['key']`.

Когда сессия больше не нужна, ее можно завершить вызовом функции `session_destroy()`. Это удалит все данные сессии и сбросит идентификатор сессии. Однако обратите внимание, что эта функция не удаляет сами файлы с данными сессии на сервере, они будут удалены при очистке устаревших сессий сервером или через определенное время неактивности.

Время жизни сессии PHP на сервере определяется несколькими факторами и может быть настроено в конфигурации PHP. По умолчанию сессии PHP сохраняются на сервере в течение **24 минут** после последнего обращения пользователя к серверу.

Однако следует отметить, что время жизни сессии может быть изменено различными способами. Например, вы можете изменить значение `session.gc_maxlifetime` в настройках PHP или использовать функцию `session_set_cookie_params()` для установки времени жизни сессии через cookie.

Также стоит учесть, что срок действия сессии может быть ограничен на стороне браузера пользователя. Браузеры обычно устанавливают время жизни cookie, связанных с сессией, и если время жизни cookie истекло, то сессия может быть считана как просроченная, несмотря на то, что она все еще существует на сервере.

Важно отметить, что PHP обрабатывает сессии с использованием куки, поэтому клиентский браузер должен поддерживать куки для работы сессий. Если куки отключены, механизм сессий может не работать должным образом.

Также следует принять во внимание некоторые меры безопасности при работе с сессиями, такие как обработка и фильтрация данных, предотвращение подделки и защита от сессионной атаки с подменой идентификаторов сессии. Как и для любых данных, принимаемых от пользователя.

## Защита от CSRF атак

Теперь у нас есть все инструменты, чтобы защитить наши формы от атак с подменой.

При работе с формой мы должны сгенерировать CSRF-токен. Для этого нам надо будет сформировать случайное значение для CSRF токена, которое в дальнейшем будет проверяться на валидность.

Это значение сохраняется в сессии на стороне сервера.



Сам CSRF токен размещается в скрытом поле формы – это будет подпись формы.

Даже если злоумышленник на своей стороне и сгенерирует такое поле, он вряд ли угадает его правильное значение.

При обработке запроса на сервере, мы получим значение CSRF токена из отправленной формы. Нам останется только сравнить его с тем, что хранится в сессии.

Если значения не совпадают, то запрос может быть отклонен, так как это может быть попытка CSRF атаки.

Давайте реализуем такую защиту у нас в приложении.

Начнем с класса Application. В методе run (а это единая точка входа) первым делом вызовем старт сессии:

```
public function run() : string {  
    session_start();  
    // ...  
}
```

Теперь для удобства создадим в классе `Render` метод генерации шаблона с формой:

```
public function renderPageWithForm(string $contentTemplateName =
'page-index.tpl', array $templateVariables = []) {
    $_SESSION['csrf_token'] = bin2hex(random_bytes(32));

    $templateVariables['csrf_token'] = $_SESSION['csrf_token'];

    return $this->renderPage($contentTemplateName, $templateVariables);
}
```

Как видите, здесь мы не пишем код дважды, но в начале создаём наш случайный токен при помощи функции [random\\_bytes](#), которая создает случайную строку в 32 байта. Затем мы преобразуем байты в строковый вид при помощи [bin2hex](#). Результат мы сохраняем в сессии и передаём в шаблон. Поскольку дальнейшие действия никак не отличаются от генерации шаблона, то в конце мы вызываем стандартный рендеринг.

Теперь переместимся в контроллер `UserController`. Там в методе `actionEdit` мы вызовем наш метод рендеринга:

```
public function actionEdit(): string {
    $render = new Render();

    return $render->renderPageWithForm(
        'user-form.tpl',
        [
            'title' => 'Форма создания пользователя'
        ]
    );
}
```

Затем в шаблоне `user-form.tpl` мы размещаем скрытое поле, которое будем получать с сервера:

```
<form action="/user/save/" method="post">
    <input id="csrf_token" type="hidden" name="csrf_token" value="{{ csrf_token }}">
    <p>
        <label for="user-name">Имя:</label>
        <input id="user-name" type="text" name="name">
    </p>
    <p>
        <label for="user-lastname">Фамилия:</label>
        <input id="user-lastname" type="text" name="lastname">
    </p>
    <p>
        <label for="user-birthday">День рождения:</label>
        <input id="user-birthday" type="text" name="birthday"
placeholder="ДД-ММ-ГГГГ">
    </p>
    <p><input type="submit" value="Сохранить"></p>
</form>
```



Не забываем очистить кэш. Если мы все сделали правильно, то в форме мы увидим код типа:

```
<input id="csrf_token" type="hidden" name="csrf_token"
value="f06bec07ddd5065e3016d4e993dcfe69570fc2b0ff9556e59ac195b304817795">
```

Настало время написать обработчик токена!

Это мы можем сделать уже в модели в методе `validateRequestData`. Но у нас в нём становится слишком много проверок. Поэтому код надо сделать более читаемым. В самом начале мы заведем переменную хранения результата. Изначально она будет иметь значение `true`, но любая не пройденная проверка будет обращать её в `false`.

Мы разделим проверки на:

1. Наличие полей в принципе
2. Проверку формата строки дня рождения
3. Проверку токена

```
public static function validateRequestData(): bool{
    $result = true;

    if(!(
        isset($_POST['name']) && !empty($_POST['name']) &&
        isset($_POST['lastname']) && !empty($_POST['lastname']) &&
        isset($_POST['birthday']) && !empty($_POST['birthday'])
    )){
        $result = false;
    }

    if(!preg_match('/^(\d{2}-\d{2}-\d{4})$/', $_POST['birthday'])){
        $result = false;
    }

    if(!isset($_SESSION['csrf_token']) || $_SESSION['csrf_token'] !=
    $_POST['csrf_token']){
        $result = false;
    }

    return $result;
}
```

Проверка токена, как видите, довольно простая. Мы проверяем его наличие в сессии и совпадение токена формы с токеном в сессии.


Теперь, если мы всё сделали правильно, форма продолжит работать. Но если мы изменим токен (сделаем его невалидным), то форма будет выдавать ошибку сохранения данных.

# Идентификация пользователя на сайте


Одной из фундаментальных точек применения интерактивных форм является идентификация пользователей. Она делится на две части:

- Аутентификация
- Авторизация

Авторизация и аутентификация — это два термина, связанных с проверкой личности и контролем доступа в компьютерных системах или сетях. Хотя эти термины часто используются взаимозаменяемо, они обозначают разные аспекты процесса проверки личности.

 Аутентификация — это процесс проверки подлинности или установления идентичности пользователя или субъекта.

При аутентификации пользователь предоставляет идентификационные данные, такие как логин и пароль, отпечаток пальца, ключ доступа или другие формы уникальных данных. Система затем сравнивает предоставленные данные с сохраненными данными, чтобы убедиться, что пользователь является тем, за кого себя выдаёт. Аутентификация обычно используется для защиты от несанкционированного доступа к системе или ресурсам.

 Авторизация — это процесс предоставления прав доступа после успешной аутентификации.

После того, как пользователь успешно прошел процесс аутентификации и его личность была проверена, система определяет, какие ресурсы или функции доступны этому пользователю на основе его привилегий или роли. Авторизация контролирует, какие действия или операции может выполнять пользователь после входа в систему. Это может включать доступ к файлам, выполнение определенных операций или изменение настроек системы.

В целом, аутентификация и авторизация вместе обеспечивают безопасность и контроль доступа в компьютерных системах, гарантируя, что только правильные пользователи получают доступ к необходимым ресурсам и информации.

# Пишем форму логина

Значит, мы должны начать с аутентификации. Для начала мы подготовим базу данных. Именно там у нас будет храниться информация о логинах и паролях. Мы применим ту же таблицу Users.

Добавим в неё два поля:

- login
- password\_hash

В первом будет храниться имя пользователя, с которым он будет входить в систему. Во втором – зашифрованный пароль.

## Хранение паролей

Хранение паролей в зашифрованном виде является одной из важных мер безопасности, и вот несколько причин, почему это необходимо:

1. Защита от несанкционированного доступа: Если пароли хранятся в открытом виде, злоумышленники или злоумышленная программа могут получить доступ к базе данных и узнать пароли всех пользователей. Зашифрованное хранение паролей делает их непригодными для использования даже в случае компрометации базы данных.
2. Обеспечение конфиденциальности: Пароли являются личными и конфиденциальными данными пользователей. Хранение паролей в зашифрованном виде помогает защитить их конфиденциальность, даже если база данных становится доступной злоумышленникам.
3. Защита от внутреннего доступа: Не всегда угрозы идут извне. Внутренние работники или администраторы могут иметь доступ к базе данных. Хранение паролей в зашифрованном виде ограничивает возможность несанкционированного использования или злоупотребления этим доступом.

Для безопасного хранения паролей рекомендуется применять хеширование паролей.



Хеширование — это процесс преобразования входных данных (например, пароля или сообщения) в неповторимую строку фиксированной длины, называемую хэш-значением или просто хэшем.

Хэш-функция принимает входные данные переменной длины и генерирует хэш-значение фиксированной длины, которое обычно представляется в виде строки шестнадцатеричных или двоичных символов.

Хорошая хэш-функция обладает несколькими важными свойствами:

1. Детерминированность: Для одинакового входа хэш-функция всегда должна генерировать один и тот же хэш.
2. Уникальность: Даже небольшое изменение входных данных должно приводить к существенно отличающемуся хэшу.
3. Необратимость: Из хэш-значения невозможно восстановить исходные данные. Хорошая хэш-функция должна быть устойчива к обратному преобразованию.
4. Равномерное распределение: Хэш-функция должна равномерно распределять хэш-значения по всему диапазону возможных значений.

Хэширование широко используется в области безопасности и проверки целостности данных. Одним из основных применений является хэширование паролей. Вместо хранения паролей в открытом виде, системы хранят их хэш-значения. При проверке аутентификации, система хэширует предоставленный пользователем пароль и сравнивает его с сохраненным хэшем. Если хэши совпадают, то пароль считается верным.

Таким образом, при хешировании пароль преобразуется в неповторимую строку фиксированной длины, которая затрудняет обратное преобразование в исходный пароль.

Давайте создадим метод получения хэша из строки, чтобы сохранить первый пароль. Для этого в слое приложения создадим класс Auth:

```
namespace Geekbrains\Application1\Application;

class Auth {
    public static function getPasswordHash(string $rawPassword): string {
        return password_hash($_GET['pass_string'], PASSWORD_BCRYPT);
    }
}
```

Как видите, метод принимает на вход строку с незашифрованным паролем. Далее внутри себя при помощи функции [password\\_hash](#) мы получаем хэш значение для заданного пароля.

Давайте для удобства в UserController создадим метод получения хэша:

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

Теперь мы можем через браузер получить хэш для своего пароля. Например, так:

[http://mysite.local/user/hash/?pass\\_string=geekbrains](http://mysite.local/user/hash/?pass_string=geekbrains)

Нам нужно сохранить это значение в БД. Давайте создадим там запись с ID 1. Логин будет admin, а в поле пароль мы запишем полученную строку. Это можно сделать вручную прямо через MySQL Workbench.

Мы готовы к тому, чтобы описать логику логина. Давайте встроим ссылку в наш шаблон main.tpl:

```
<body>  
    <div id="header">  
        {% include "auth-template.tpl" %}  
    </div>
```

В шапке нашего сайта будет подключаться специальный шаблон auth-template:

```
{% if not user_authorized %}  
    <p><a href="/user/auth/">Вход в систему</a></p>  
{% else %}  
    <p>Добро пожаловать на сайт!</p>  
{% endif %}
```

Если пользователь уже прошел аутентификацию, то мы будем показывать приветствие. Иначе – ссылку на форму логина.

Для неё создадим новый action в UserController:

```
public function actionAuth(): string {  
    $render = new Render();  
  
    return $render->renderPageWithForm(  
        'user-auth.tpl',  
        [  
            'title' => 'Форма логина'  
        ]  
    );  
}
```

Вызываемая по ссылке форма user-auth.tpl будет выглядеть несложно:

```
{% if not auth-success %}
  {{ auth-error }}
{% endif %}

<form action="/user/login/" method="post">
  <input id="csrf_token" type="hidden" name="csrf_token" value="{{ csrf_token }}">
  <p>
    <label for="user-login">Логин:</label>
    <input id="user-login" type="text" name="login">
  </p>
  <p>
    <label for="user-password">Пароль:</label>
    <input id="user-password" type="password" name="password">
  </p>
  <p><input type="submit" value="Войти"></p>
</form>
```

Обратите внимание на верх формы. Если пользователь введет некорректные данные логина и пароля, мы вернем его на форму логина и укажем ошибку.

Как видите, в форме мы обращаемся к методу actionLogin в контроллере, который попытается проверить данные и произвести аутентификацию. Либо вернет ошибку. Настало время описать логику его работы:

```
public function actionLogin(): string {
    $result = false;

    if(isset($_POST['login']) && isset($_POST['password'])) {
        $result = Application::$auth->proceedAuth($_POST['login'],
        $_POST['password']);
    }

    if(!$result) {
        $render = new Render();

        return $render->renderPageWithForm(
            'user-auth.tpl',
            [
                'title' => 'Форма логина',
                'auth-success' => false,
                'auth-error' => 'Неверные логин или пароль'
            ]
        );
    }
    else {
        header('Location: /');
        return "";
    }
}
```

В начале мы создаем переменную результата. Затем, если у нас есть переданные логин и пароль, нам нужно проверить их корректность. Для этого надо уметь обращаться к соответствующему методу в Auth.

Давайте сделаем его частью приложения, модифицировав класс Application:

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

Теперь опишем сам метод проверки логина и пароля в классе Auth:

```
public function proceedAuth(string $login, string $password): bool{
    $sql = "SELECT id_user, user_name, user_lastname, password_hash FROM
users WHERE login = :login";

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

    if(!empty($result) && password_verify($password,
$result[0]['password_hash'])){
        $_SESSION['user_name'] = $result[0]['user_name'];
        $_SESSION['user_lastname'] = $result[0]['user_lastname'];
        $_SESSION['id_user'] = $result[0]['id_user'];

        return true;
    }
    else {
        return false;
    }
}
```

Нам нужно уметь обращаться к базе данных, чтобы получить хранимый там хэш пароля. Делается это обычным уже знакомым нам способом.

Далее мы должны проверить, что в базе данных по логину в принципе существует запись. И что хранимый хэш соответствует переданному паролю. Вторую проверку делает встроенная в PHP функция [password\\_verify](#). Эта функция принимает пароль в явном виде и хэш, проверяя, могут ли они соответствовать.

Если логин и пароль введены корректно, то мы задаем в сессии имя и фамилию пользователя, возвращая в итоге true. Иначе – false.

В методе контроллера при получении false мы вызываем генерацию формы заново, указывая там ошибку. Если же логин и пароль верны, то при помощи функции header производим перенаправление нашего пользователя на главную страницу.

Осталось несколько косметических доработок. В методе renderPage класса Render добавим формирование параметра:

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

Он будет создавать маркер о том, что пользователь авторизован. Ведь в шаблоне шапки мы указали условие `if`, по которому либо показываем ссылку, либо отдаем приветствие.


## Ролевая модель разграничения прав

В действиях выше мы с вами уже создали несколько методов в контроллерах, которые не должны быть доступны для всех пользователей. Например:

- Создание пользователя (`actionSave`)
- Получение хэша (`actionHash`)

Для таких методов нужно проводить процесс авторизации, то есть проверять права на доступ.

Одним из наиболее гибких подходов здесь является RBAC.

 RBAC (Role-Based Access Control) – это модель контроля доступа, используемая для управления правами доступа пользователей к ресурсам в компьютерных системах.

В модели RBAC доступ к ресурсам основывается на ролях, которые назначаются пользователям.

В RBAC роли определяют набор разрешений, которые пользователь может иметь в системе. Вместо назначения прав доступа каждому пользователю отдельно, пользователю назначается одна или несколько ролей, и эти роли определяют, какие ресурсы и операции доступны пользователю.

Основные компоненты модели RBAC включают:

- **Роли:** Роли представляют группы разрешений или привилегий, которые назначаются пользователям в системе. Например, веб-сайт может иметь роли администратора, модератора и обычного пользователя.
- **Разрешения:** Разрешения определяют конкретные операции или доступные ресурсы, которые могут быть выполнены или доступны для ролей. Например, разрешения могут включать создание, чтение, изменение или удаление данных или доступ к определенным функциям системы.
- **Пользователи:** Пользователи — это отдельные субъекты, которым назначаются роли в системе. Пользователи получают права доступа в соответствии с ролями, назначенными им.



- **Ассоциации ролей:** В модели RBAC могут существовать связи между ролями, например, иерархия или зависимость. Некоторые роли могут быть подчинены другим ролям, что позволяет управлять наследованием разрешений.

🔥 Преимущества модели RBAC включают упрощение управления доступом, повышение безопасности, гибкость при изменении прав доступа и уменьшение сложности системы.

Модель RBAC широко применяется в различных системах и приложениях, включая операционные системы, базы данных, веб-приложения и другие, где требуется управление доступом к ресурсам.

Для хранения ролей в БД мы заведем таблицу user\_roles:

```
CREATE TABLE `user_roles` (  
  `id_user_role` int(11) NOT NULL AUTO_INCREMENT,  
  `id_user` int(11) DEFAULT NULL,  
  `role` varchar(45) DEFAULT NULL,  
  PRIMARY KEY (`id_user_role`)  
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=latin1
```

В ней по ID пользователя будут привязаны роли, которые доступны для конкретного пользователя. Например, для пользователя с ID 1 зададим роль admin.

Далее нам нужно хранить разрешения на доступ к методам. Для того, чтобы наши разрешения корректно работали, создадим абстрактный контроллер, от которого будут наследоваться все контроллеры, которым требуется иметь ограничение на вызов методов:

```
namespace Geekbrains\Application1\Domain\Controllers;  
  
use Geekbrains\Application1\Application\Application;  
  
class AbstractController {  
  
    protected array $actionsPermissions = [];  
  
    public function getUserRoles(): array{  
        $roles = [];  
  
        if(isset($_SESSION['id_user'])){  
            $rolesSql = "SELECT * FROM user_roles WHERE id_user = :id";  
  
            $handler = Application::$storage->get()->prepare($rolesSql);  
            $handler->execute(['id' => $_SESSION['id_user']]);  
            $result = $handler->fetchAll();  
  
            if(!empty($result)){  
                foreach($result as $role){  
                    $roles[] = $role['role'];  
                }  
            }  
        }  
    }  
}
```

```

        }
    }

    return $roles;
}

public function getActionsPermissions(string $methodName): array {
    return isset($this->actionsPermissions[$methodName]) ?
    $this->actionsPermissions[$methodName] : [];
}

```

В этом контроллере мы создаем свойство `actionsPermissions`, которое будет хранить разрешения на доступ к методам. Также здесь мы создадим методы получения ролей из БД по ID пользователя, а также метод доступа к закрытому полю контроллера.

Модифицируем наш `UserController` так, чтобы он наследовался от `AbstractController`-а:

```
class UserController extends AbstractController
```

В свойствах зададим перечень ограничений по доступу:

```

protected array $actionsPermissions = [
    'actionHash' => ['admin', 'manager'],
    'actionSave' => ['admin']
];

```

Нам остаётся описать логику проверки наличия прав при вызове метода. Вызов метода производится в `Application`. Логично встроиться именно в него.

В месте создания экземпляра контроллера:

```
$controllerInstance = new $this->controllerName();
```

Мы уже можем проверить, что он является экземпляром `AbstractController`. Если это так, то мы должны проверить доступность вызова метода.

Если же это простой контроллер без возможности ограничения прав, то ничего не меняется:

```
$controllerInstance = new $this->controllerName();

if($controllerInstance instanceof AbstractController){
    if($this->checkAccessToMethod($controllerInstance)){
        return call_user_func_array(
            [$controllerInstance, $this->methodName],
            []
        );
    }
}
else{
    return "Нет доступа к методу";
}

}
else{
    return call_user_func_array(
        [$controllerInstance, $this->methodName],
        []
    );
}
```

Нам остается описать метод проверки доступа к вызываемому action-у.

Метод `checkAccessToMethod` будет принимать на вход созданный экземпляр контроллера:

```
private function checkAccessToMethod(AbstractController $controllerInstance,
string $methodName): bool {
    $userRoles = $controllerInstance->getUserRoles();

    $rules = $controllerInstance->getActionsPermissions($methodName);

    $isAllowed = false;

    if(!empty($rules)){
        foreach($rules as $rolePermission){
            foreach($userRoles as $role){
                if(in_array($role, $rolePermission)){
                    $isAllowed = true;
                    break;
                }
            }
        }
    }

    return $isAllowed;
}
```

В начале мы получаем два массива – массив ролей пользователя и массив ограничений на доступ. Если в этих двух массивах найдется пересечение для конкретного метода, то мы можем вызывать его. Пересечение мы ищем при помощи `foreach`. Как только оно найдено, цикл прерывается, и мы даем доступ к вызову. Иначе вызов запрещается.

# Заключение

В данной лекции мы сформировали еще один фундаментальный аспект работы – интерактивность. Также мы научились идентифицировать нашего пользователя, управлять его ролями. И углубились в вопросы безопасности.

## Домашнее задание

1. При помощи регулярных выражений усильте проверку данных в `validateRequestData` так, чтобы пользователь не смог передать на обработку любую строку, содержащую HTML-теги (например, `<script>`).
2. Доработайте шаблон аутентификации. В нем нужно добавить две вещи:
  - В приветствии нужно выводить имя залогинившегося пользователя;
  - Также надо выводить ссылку «Выйти из системы», которая будет уничтожать сессию пользователя.
3. Переработайте имеющийся функционал приложения на формы:
  - Создание, обновление и удаление пользователя теперь должно производиться через формы;
  - Если пользователь обновляется, в форму должны быть выведены текущие значения. Это может быть сделано ссылкой из списка пользователей (рядом с каждым из них будет своя ссылка “Обновить данные”).
4. \* Создайте функцию “Запомнить меня” в форме логина:
  - В форме должен появиться checkbox “Запомнить меня”;
  - При нажатии на него в процессе логина пользователю выдается cookie, по которому происходит автоматическая авторизация, даже если сессия закончилась;
  - При логине нужно будет генерировать токен из `random_bytes()`, размещая его в cookies и БД, чтобы сравнивать их;
  - При выходе из системы токен надо деактивировать.
5. Исправьте потолстевший Абстрактный контроллер.

## 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/>