PHP

Grundlagen, Dynamische Webseiten, Datenbanken

Themenübersicht

Basics

- Grundlagen
- Sprachkonstrukte
- Funktionen
- Closures
- Namespaces

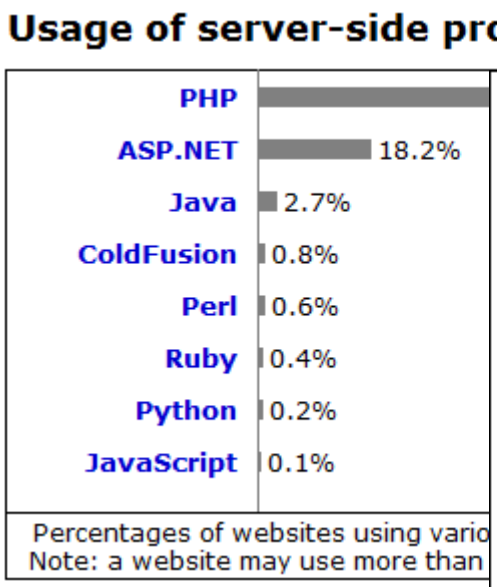
Dynamische Webseiten

- Formulare
- Datenverarbeitung, Validierung, filter-Funktionen
- Datenbanken, SQL-Injections, Passwort-API
- Cookies & Sessions

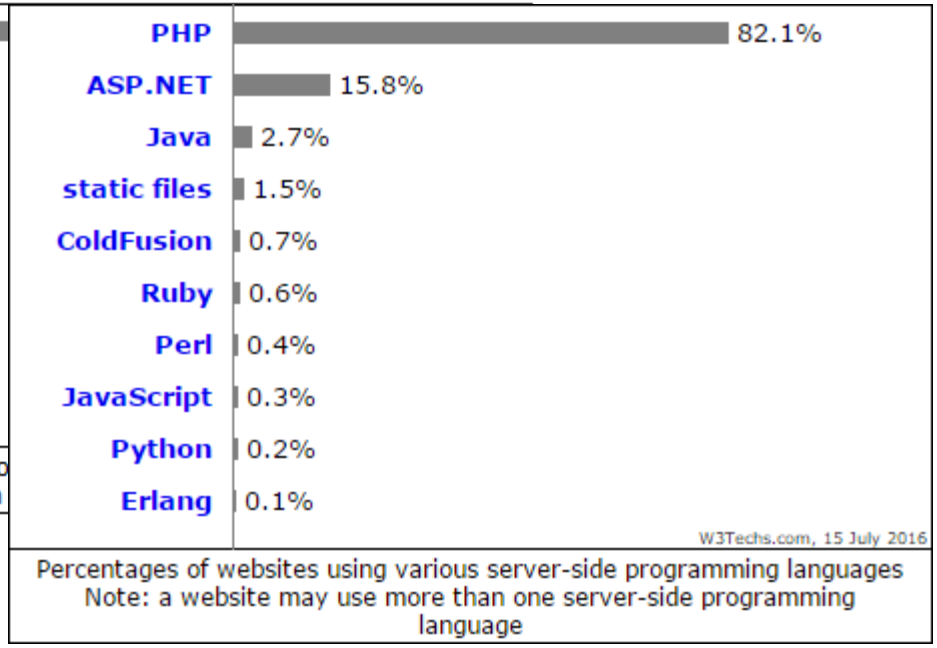
- PHP 7
- Composer & PHP-Frameworks

Motivation

Verbreitung serverseitiger Sprachen



Quelle: w3techs.com, Jan 2014



Quelle: w3techs.com, Jul 2016

- Weitverbreiteste serverseitige Sprache
- Für Webentwickler ist PHP oft unvermeidbar

Grundlagen

Entwickelt 1995 von Rasmus Lerdorf als „Personal Home Page“

- Später umbenannt in „PHP: Hypertext Processor“

Historie

- Lange Zeit eine sehr simple Skriptsprache
- „Richtige“ Objektorientierung erst seit PHP 5.0
- PHP 6 als ursprünglicher Nachfolger von PHP 5 geplant
  - > Gescheitert auf Grund zahlreicher Probleme
  - > Versionsnummer 6 wurde nie vergeben
- Aktuelle Version: PHP 7 (Juni 2016)

Grundlagen

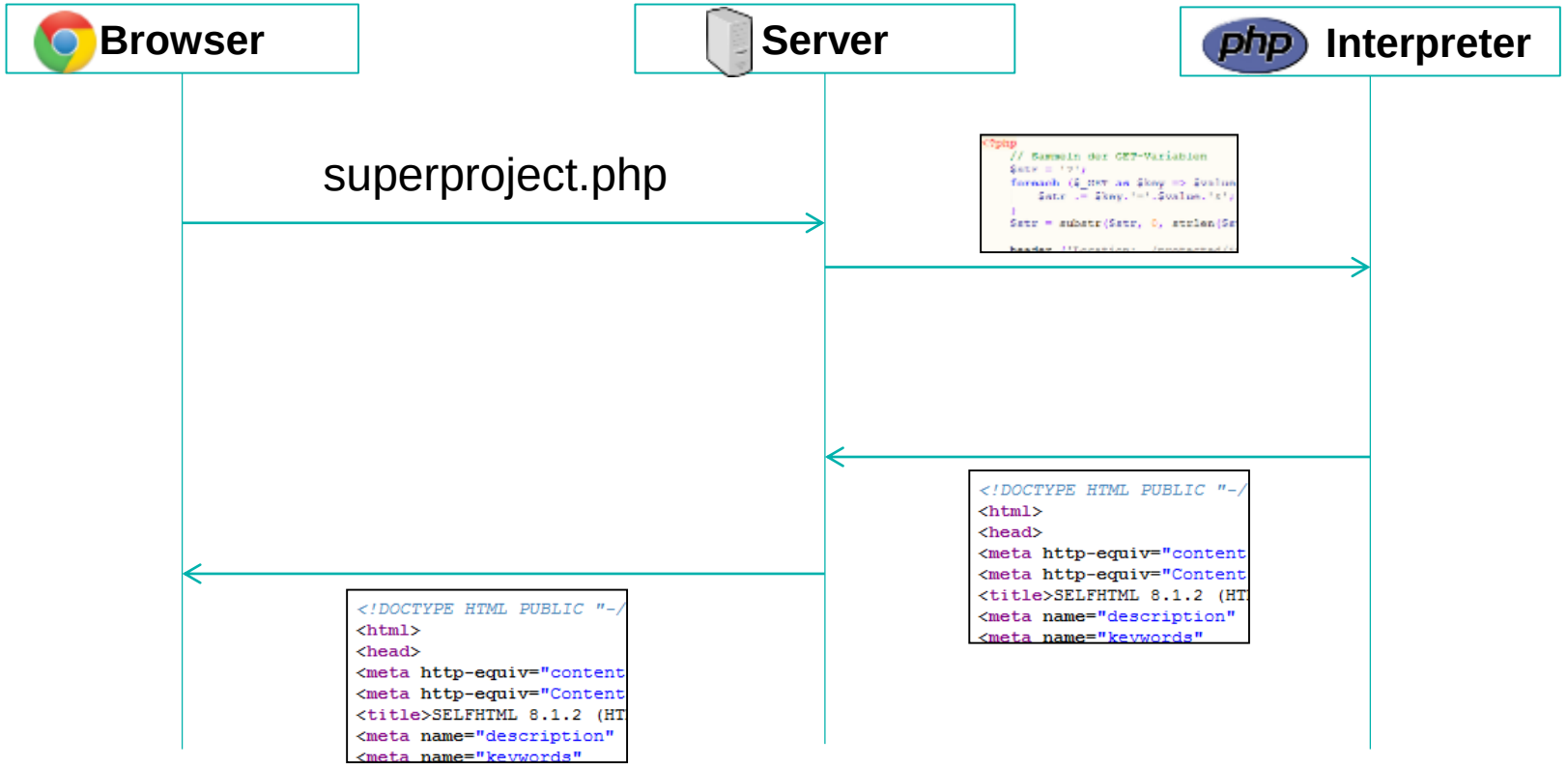
Voraussetzungen an den Server

- Installierter PHP-Interpreter notwendig
- Typisch für Webentwicklung ist ein „LAMP"-Stack
  - > Linux, Apache, MySQL, PHP

Ablauf

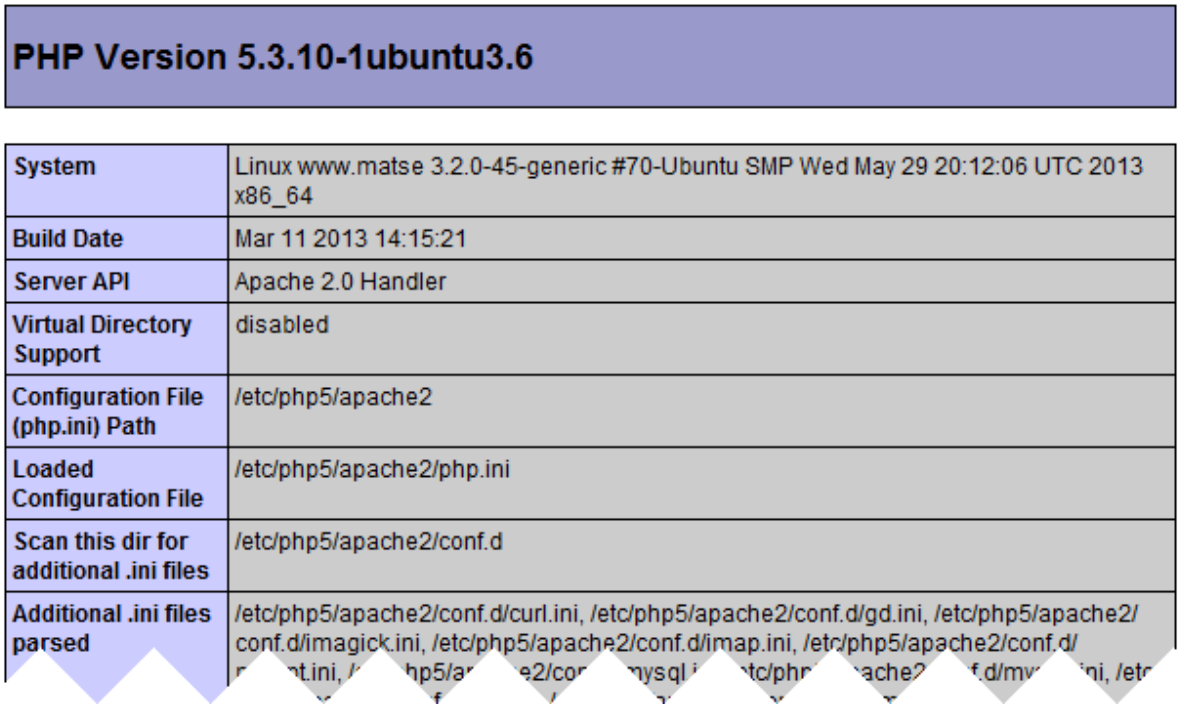
- Browser stellt den Request an den Server
- Dateiendung „.php" signalisiert dem Server, dass die Datei PHP-Code beinhaltet
- PHP-Interpreter durchläuft die Datei und interpretiert den Quellcode
- Ergebnis wird an den Browser zurückgesendet

Aufruf einer PHP-Datei (vereinfacht)



Beispiel: test.php

```
<?php
    phpinfo();
?>
```



PHP Version 5.3.10-1ubuntu3.6

System	Linux www.matse 3.2.0-45-generic #70-Ubuntu SMP Wed May 29 20:12:06 UTC 2013 x86_64
Build Date	Mar 11 2013 14:15:21
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc/php5/apache2
Loaded Configuration File	/etc/php5/apache2/php.ini
Scan this dir for additional .ini files	/etc/php5/apache2/conf.d
Additional .ini files parsed	/etc/php5/apache2/conf.d/curl.ini, /etc/php5/apache2/conf.d/gd.ini, /etc/php5/apache2/conf.d/imagick.ini, /etc/php5/apache2/conf.d/imap.ini, /etc/php5/apache2/conf.d/

Syntax
Beispiele

beispiel2.php

```
<?php
    $titel = 'Web-Engineering';
    $gruss = 'Willkommen im Kurs!';
?>
<!DOCTYPE html>
<html>
<head>
    <title><?php echo $titel; ?></title>
</head>
<body>
    <?php echo $gruss; ?>
</body>
</html>
```

Syntax
Beispiele

beispiel2.php

- Produziert folgenden Code:

```
<!DOCTYPE html>
<html>
<head>
    <title>Web-Engineering</title>
</head>
<body>
    Willkommen im Kurs!
</body>
</html>
```

Syntax

Vieles bekannt aus anderen Sprachen

- if, for, while…
- Funktionen
- Klassen und Objekte

Einige Konzepte unbekannt

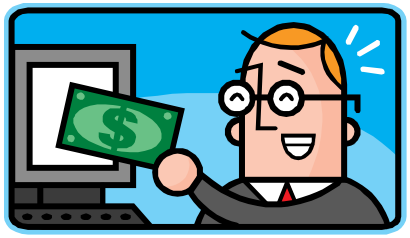
- Fehlende Variablendeklaration
- Closures
- Cookies & Sessions

Variablen

- Dargestellt durch Dollar-Zeichen ($) gefolgt vom Namen
- Keine explizite Deklaration!
- Kein fester Typ!
- Nicht initialisierte Variablen haben einen vom Typ abhängigen Vorgabewert (false, null, leerer String oder leeres Array)

```php
<?php
    $name = 'Max';
    $id = 42;
    $pi = 3.14;

    $foo += 5;
?>
```



Datentypen

Typ	Beschreibung	Beispiel/Literal
boolean	Wahrheitswert	true, false
integer	Ganzzahl	0, 123, -12, 0xFF
float / double	Fließkommazahl (IEEE 754)	3.14, 1.2e5
string	Zeichenkette	'foo', "bar"
array	(assoziatives) Arrray	array(), array(1, 2, 3), array('key' => 'val')
object	Objekt	new Car (wobei Car eine Klasse ist)
NULL	Variable ohne Wert	NULL
resource	Referenz zu einer Ressource	mysql_connect('localhost', …)
callable (nicht primär)	Funktionszeiger	function() { }

Weitere Informationen: http://php.net/manual/language.types.php

Typumwandlung

- Implizit:

```php
$foo  = '5';                  // '5'
$foo += 3;                    // 8
$foo  = '10 Mann' + $foo;     // 18
$foo += true;                 // 19
$foo += 1.5;                  // 20.5
$foo += array();              // FATAL ERROR
```

> Führt schnell zu Problemen
> Besser explizit casten
  - Datenbanken erwarten oft einen bestimmten Typen

> Implizit casten besser vermeiden!

Weitere Informationen: https://php.net/manual/language.types.type-juggling.php

Typumwandlung

- Explizit:

```
$foo = '1.5';           // '1.5'
$foo = (double) $foo;   // 1.5
$foo = (int) $foo;      // 1
$foo = (boolean) $foo;  // true
$foo = (string) $foo;   // '1'
```

cast to	false	1	0	1.7	-1.2	''	'1'	'php'	array()
boolean	false	true	false	true	true	**false**	true	**true**	**false**
int	0	1	0	1	-1	0	1	**0**	0
double	0	1	0	1.7	-1.2	0	1	**0**	0
string	''	'1'	'0'	'1.7'	'-1.2'	''	'1'	'php'	ERROR

Vergleich von Variablen

```php
if (true == 'php' && 'php' == 0 && 0 != true) {
    echo 'true == "php" == 0 != true';
}
```

> Mit == auf Gleichheit zu prüfen kann zu Problemen führen
  - Beispiel: strpos

$a === $b

- Typstarker Vergleich!
- Nur bei gleichem Typen und gleichem Inhalt von $a und $b wird true zurückgegeben

Typschwache Vergleiche mittels ==

	TRUE	FALSE	*1*	*0*	*-1*	*"1"*	*"0"*	*"-1"*	NULL	*array()*	*"php"*	*""*
TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
1	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE
-1	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
"1"	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE
array()	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
"php"	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE

Quelle: http://php.net/manual/de/types.comparisons.php

Typstarke Vergleiche mittels ===

	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
1	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-1	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"1"	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE
array()	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE
"php"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE

Quelle: http://php.net/manual/de/types.comparisons.php

Erzeugung eines Strings

- Einfache Anführungszeichen
- Steuerzeichen ($, \n) werden ignoriert (bzw. nicht interpretiert)

- Beispiele:

```
echo 'Einfacher String';
echo 'Auch ein Zeilenumbruch
ist kein Problem';

echo 'I\'ll be back';   // I'll be back
echo 'Dateien: C:\\*'; // Dateien: C:\*

$fuenf = 5;
echo '2+3 = $fuenf';    // 2+3 = $fuenf
```

Erzeugung eines Strings

- Doppelte Anführungszeichen
- Steuerzeichen ($, \n) werden interpretiert
- Variablen werden erkannt
- Beispiele:

```php
echo "Zeilenumbruch folgt\n";

$fuenf = 5;
echo "2+3 = $fuenf";     // 2+3 = 5
echo "2+3 = {$fuenf}";   // 2+3 = 5
```

- Für Strings, die keine Variablen enthalten wird meist ein einfaches Anführungszeichen verwendet
- Für SQL-Queries wird meist ein doppeltes Anführungszeichen verwendet

Verkettung

- Punktoperator
- Beispiele:

```php
$uni = 'RWTH' . ' ' . 'Aachen'; // 'RWTH Aachen'

$sql = "SELECT `id` FROM `users` "
     . "WHERE `name` = 'Max' ";

$id = 305;
$ort = 'Raum ' . $id; // 'Raum 305'

$ort .= ' (3. OG)';   // 'Raum 305 (3. OG)'
```

Syntax
Arrays

Eigenschaften

- Keine feste Größe
- Assoziative Arrays möglich (eigentlich sogar eher der Standard)
- Mischformen von String und Integern als Key erlaubt
  - > Bedeutet nicht, dass dies auch sinnvoll ist…
- Als Wert ist alles erlaubt!
- Numerische Arrays können Lücken haben
  - > Ebenfalls nicht zu empfehlen…

Erstellung:

```php
$normal = array();
$mix1   = array(2, 4, 8, 'Tomate');
$mix2   = array('foo' => 'bar', 12 => true);
$colors = array('rgb1' => array(0,0,255));
```

Zugriff und Bearbeitung:

```php
$arr = array();  // Alternativ: []
$arr[] = 'EINS'; // array('EINS')
$arr[] = 'ZWEI'; // array('EINS', 'ZWEI')

$arr[0] = 'NEU'; // array('NEU', 'ZWEI')
$arr['a'] = 5;   // array(0 => 'NEU', 1 => 'ZWEI',
                 //       'a' => 5)

echo $arr[1];    // 'ZWEI'
```

Grundsätzliches bekannt aus Java

- Bedingungen
  - > if
  - > else if
  - > else

- Schleifen
  - > for
  - > while
  - > do-while
  - > break
  - > continue

foreach-Schleife

```php
foreach ($arr as $value) {
    // ...
}

foreach ($arr as $key => $value) {
    // ...
}
```

Weitere Informationen: http://php.net/manual/control-structures.foreach.php

Dateien mittels *require* und *include* einbinden

- Ermöglicht Auslagerung und Mehrfachnutzung des Codes
- Bindet den Code aus der Datei ein
- Geltungsbereich von der Position des Befehlsaufruf abhängig

```
require './lib/Database.class.php'
```

> „Fatal Error", falls ein Fehler beim Einbinden auftritt
  - Die Ausführung wird sofort abgebrochen

```
include './lib/Encoding.class.php'
```

> Warning, falls ein Fehler beim Einbinden auftritt
  - Skript wird weiterausgeführt

Dateien mittels *require* und *include* einbinden

- *include* nur bei Dateien, die für den Programmablauf optional sind
  - > I.d.R. ist die Benutzung von *require* sinnvoller
- „Geklammerter Syntax" möglich

```
require('./lib/Database.class.php');
include('./lib/Encoding.class.php');
```

- Eine Pfadangabe führt dazu, dass nicht im *include_path* gesucht wird
  - > include_path gibt an in welchem Verzeichnis PHP einzubindende Dateien suchen soll (für Bibliotheken zu gebrauchen)

- *require_once* und *include_once* binden die Datei nur einmal ein, auch wenn der Funktionsaufruf öfter erfolgt

```
require_once('./lib/Database.class.php');
```

Sonstige Befehle

```
echo 'Text';
```

> Gibt den/die übergebenen Parameter aus

```
exit;
```

> Beendet die Ausführung des Skripts


```
unset($variable)
```

> Löscht die angegebene Variable

Sonstige Befehle

```
bool isset($variable)
```

> Prüft ob eine Variable exisitert und nicht *NULL* ist

```
bool empty($var)
```

> Prüft, ob eine Variable einen Wert enthält
> Für die folgenden Werte wird *TRUE* zurückgegeben:
  - Leere Zeichenkette (''), leeres Array (*array()*)
  - `0, 0.0, '0.0'`
  - *NULL, FALSE*
  - In einer Klasse deklarierte, aber nicht belegt Variable

Sonstige Befehle

```
bool define($name, $value)
```

> Setzt Konstante die in allen Geltungsbereichen verfügbar ist
> Nur für Werte benutzen, die für die ganze Applikation relevant sind

> Beispiele:

```
define('ROOT_DIR', '/my_project');
define('VERSION', '3.1.24');
```

> Auslesen des Wertes ohne $-Zeichen:

```
echo VERSION;
```

Exceptions

```
try {
    funcWhichMightThrowException();
} catch (Exception $e) {
    echo 'Problem calling ...: ' . $e->getMessage();
}
```

- Eigene Exceptionklassen können von Exception abgeleitet werden

```
class MyException extends Exception {...}
throw new MyException();
```

- Beim Abfangen kann der Exceptionname angegeben werden

- Die meisten PHP-Funktionen werfen im Fehlerfall keine Exception

Weitere Informationen: http://php.net/manual/language.exceptions.php

Es gibt auch generische Exception-Handler-Funktionen!

```php
set_exception_handler('exception_handler');

function exception_handler($e) {
    echo 'Ups! Nicht behandelte Exception: ' .
        $e->getMessage();
}
throw new Exception('Problem aufgetreten!');
```

Hier bricht das Skript ab:

```
Ups! Nicht behandelte Exception: Problem aufgetreten!
```

Unterschied: Errors & Exceptions

- Errors werden von PHP bei Fehlersituationen geworfen (und führen z.B. zum Abbruch des Skripts).

- Aber auch Errors können gefangen werden:

```
    errorHandler($errLevel, $errMsg, $errFile, $errLine) {
        throw new ErrorException($errMsg, $errLevel, $errLevel,
            $errFile, $errLine);
    }
    set_error_handler("errorHandler");
```

- Fatal-Errors können ebenfalls gefangen werden:

```
    register_shutdown_function("fatal_handler");
```

Weitere Informationen: https://www.sitepoint.com/error-handling-in-php/

Syntax
Funktionen

Wie Variablen sind auch Funktionen nicht typsicher!

- Funktionen können manchmal Zahlen, manchmal Strings zurückgeben
- In der Regel ist es aber sinnvoll, dass eine Funktionen nur einen Typen zurückgibt.

Angabe von Vorgabewerten möglich

- Beispiel:

```php
function machKaffee($typ = 'Kaffee') {
    return 'Eine Tasse ' . $typ;
}

echo machKaffee();              // 'Eine Tasse Kaffee'
echo machKaffee('Cappucino');   // 'Eine Tasse Cappucino'
echo machKaffee('Espresso');    // 'Eine Tasse Espresso'
```

Syntax
Funktionen

Call-By-Reference

```php
function toggle(&$value) {
    $value = ($value + 1) % 2;
}
$a = 1;
toggle($a);
// $a === 0
```

Variable Anzahl von Parametern

```php
function tolerant() {
    $numargs = func_num_args(); // Anzahl
    $arg1 = func_get_arg(0);    // 1. Parameter
    $args = func_get_args();    // Array mit Parametern
}
tolerant(1, 2, 4);
```

Variablen im Geltungsbereich außerhalb der Funktion sind nur durch „global"-Schlüsselwort zu erreichen

- Beispiel:

```
$a = 1;
$b = 2;
function incA() {
    global $a;    // $b ist unbekannt
    $a++;
}

incA();
echo $a; // 2
```

Closures

- Anonyme Funktionen ohne Namen
- Funktionszeiger wird in Variable gespeichert
  - > Variable kann wie Funktion benutzt werden
  - > Kann als Argument übergeben werden
- Einfacher Austausch oder die Übergabe von Funktionen möglich

Closures

- Einfache anonyme Funktion

```
$ar = range(1, 10);              // Array mit [1,..,10]

$mySort = function($a, $b) {  // Anonyme Funktion
    return $b - $a;              // gespeichert in $mySort
};

usort($ar, $mySort);          // Eigene Funktion
                              // zur Sortierung nutzen
```

Weitere Informationen: https://php.net/manual/en/function.usort.php

Closures

```
$dayPrepare = function() {
    $dayNames = array('Mo', 'Di', 'Mi', 'Do',
                      'Fr', 'Sa', 'So');

    return function($day) use ($dayNames) {
        return $dayNames[$day];
    };
};

$dayNameGetter = $dayPrepare();

echo $dayNameGetter(1); // Di
```

Anonyme Funktion in einer anonymen Funktion

Variable *$dayNames* aus äußerem Geltungsbereich wird benutzt

Aufruf der inneren anonymen Funktion, die immer noch Zugriff auf die Variable *$dayNames* hat

Syntax
Objektorientierung

Klassengestützte OOP (ähnlich zu Java)

- Nutzung von Objekten:

```
$auto = new Auto();
$ps = $auto->getPs();
```

- Vererbung (mittels Schlüsselwort *extends*)

- Zugriffsmodifizierer
  - > *public* – Zugriff auf das Attribut auch von außerhalb
  - > *protected* – Nur erbende Klassen haben Zugriff
  - > *private* – Zugriff nur innerhalb der Klasse möglich

Klassengestützte OOP

- Zugriff auf Membervariablen

```
$this->name     // Innerhalb der Klasse
$objekt->name   // Außerhalb der Klasse (falls public)
self::name      // Zugriff auf statische Variable
parent::name    // Zugriff auf Elternklasse (statisch)
```

- Konstruktor

```
function __construct() { … }
```

- Destruktor

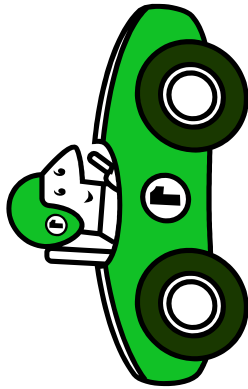
```
function __destruct() { … }
```

Syntax
Objektorientierung

```php
class Auto {
    private $farbe = 'blau';
    private $ps;
    public static $speedLimit = 100;

    public function __construct($ps = 60) {
        $this->ps = $ps;
    }
    public function getPs() {
        return $this->ps;
    }
}

$auto = new Auto();         // Instanz der Klasse Auto
$geschw = $auto->getPs(); // 60
```



Syntax
Namespaces

- Beispiel:

```php
<?php
    namespace SuperProject\Lib;
    const CONF_FILE = 'super.conf';
    function myFunction() {
        return 'Hallo';
    }
    class MyClass {
        static function random() { return 4; }
    }
?>

<?php
    echo \SuperProject\Lib\CONF_FILE;
    echo \SuperProject\Lib\myFunction();
    echo \SuperProject\Lib\MyClass::random();
?>
```

Weitere Informationen: http://php.net/manual/language.namespaces.php

Benutzung eines Namespaces

- Beispiel:

```
use \SuperProject\Lib as L;

echo L\CONF_FILE;
```

- Auflösung eines Namens:
  - > Absolut: \A\B\C
  - > Relativ: A\B\C (ohne führenden Slash)
  - > Vergleichbar mit Dateipfaden

Datenhaltung meist über Datenbanken

- Dennoch manchmal Zugriff auf Dateien notwendig

```
resource fopen($filename, $mode)
```

> Öffnet Datei $filename
> $mode gibt den Zugriffstypen an: 'r' zum Lesen, 'w' zum Schreiben, 'a' zum Anhängen
> Tritt ein Fehler auf wird false zurückgegeben

```
bool fclose($handle)
```

> Schließt den Dateizeiger
> Rückgabewert true, wenn das Schließen funktioniert hat

Syntax
Dateizugriff

```
string fread($handle, $length)
```

> Liest von dem Dateizeiger bis zu $length Bytes

```
int fwrite($handle, $string)
```

> Schreibt den Inhalt von $string in die Datei, auf die der Dateizeiger zeigt
> Liefert Anzahl der geschriebenen Bytes zurück oder FALSE im Fehlerfall

```
bool file_exists($filename)
```

> Gibt TRUE zurück, wenn die Datei oder das Verzeichnis exisitert

```
int filesize($filename)
```

> Liefert die Größe der übergebenen Datei zurück.

Syntax
Dateizugriff

Vereinfachter Zugriff auf Dateien:

```
string file_get_contents($filename)
```

> Liefert den Inhalt der Datei $filename zurück

```
int file_put_contents($filename, $data)
```

> Schreibt $data in die Datei mit dem Namen $filename
> Wenn die Datei nicht exisitert, wird sie erstellt

Beispiel

```php
<?php
    $fileName    = 'title_form.html';
    $fileContent = file_get_contents($fileName);

    echo $fileContent; // Ausgabe des Dateiinhaltes
                       // (auch mit HTML-Tags)

    $fileContent = htmlspecialchars($fileContent);
    echo $fileContent; // Ausgabe mit „entschärften"
                       // HTML-Tags

?>
```

Dynamische Webseiten

Dynamische Webseiten

Statische Webseiten

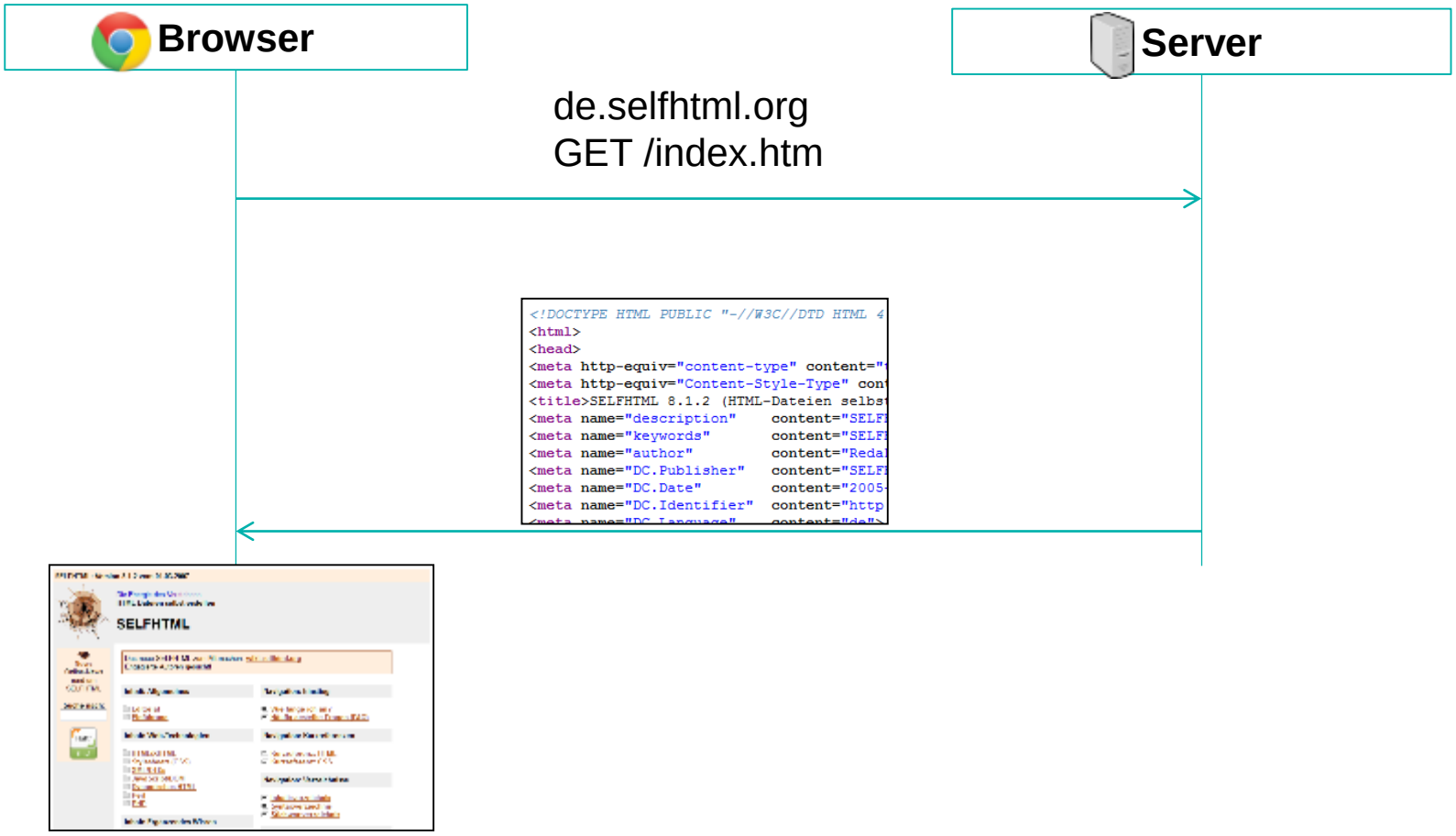
- HTML/CSS-Dateien auf dem Server
  - > Auslieferung an den Browser
  - > Stellen immer den gleichen Inhalt dar
  - > Beispiel: Webvisitenkarte

Dynamische Webseiten

- Erzeugung der Seite im Moment der Anforderung
  - > Darstellung aktueller Daten möglich
  - > Inhalt abhängig von Aktionen des Benutzers
  - > Beispiel: Suchformular

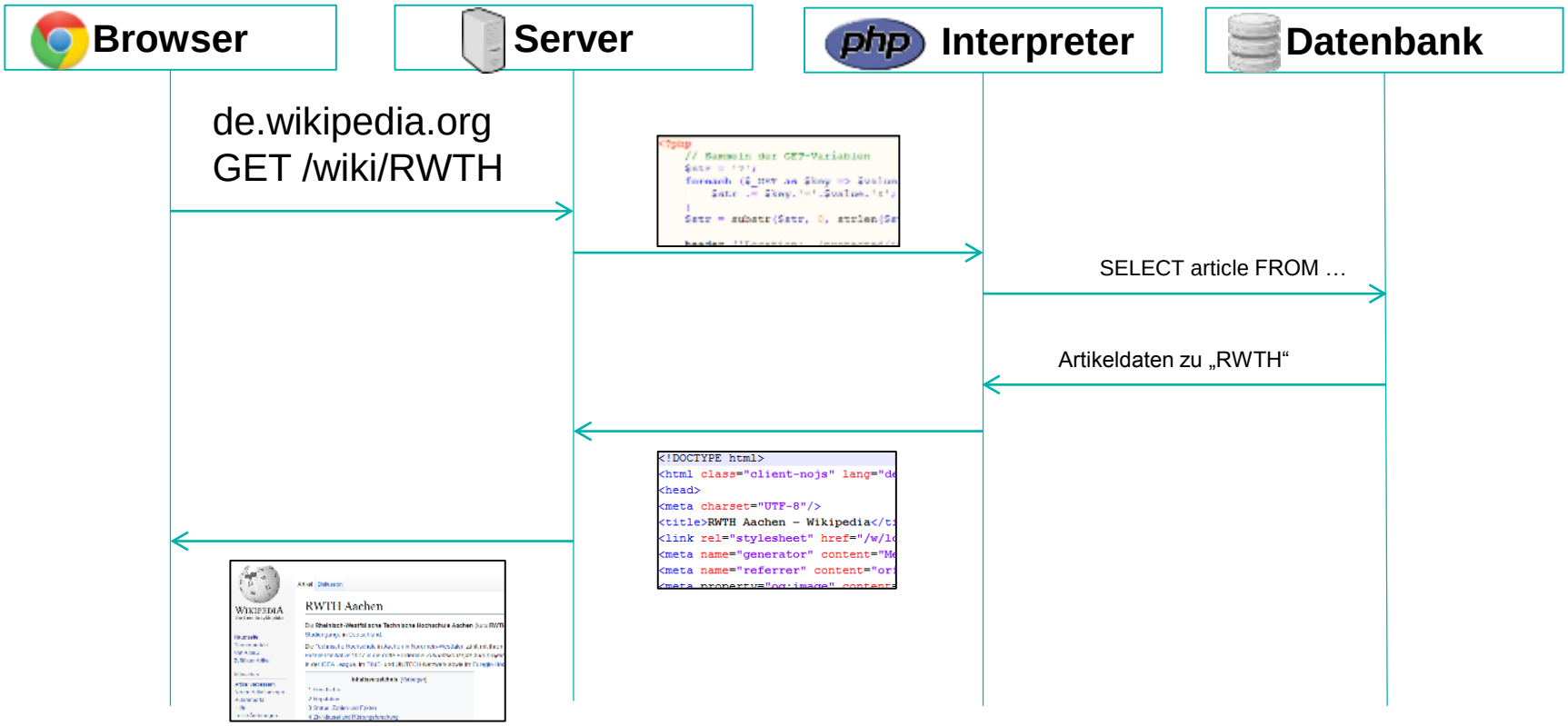
Dynamische Webseiten

Aufruf einer statischen Webseite (vereinfacht)



Dynamische Webseiten

Aufruf einer dynamischen Webseite (vereinfacht)



Beispiel: Login-Formular

```
<form action="do.php?q=login" method="post">
    <input type="text" name="username" />
    <input type="password" name="pw" />
    <input type="submit" value="Login" />
</form>
```

- Übermittelt Parameter „username“ und „pw“ per POST-Request

- Auswertung des GET-Parameters (q=login)
  - > Auf welcher Seite befinden wir uns?
- Auswertung des Formulars
  - > Ist „username“ und „pw“ korrekt?
  - > Evtl. per Datenbankabfrage herausfinden?

GET-Request

- Übertragung der Daten über die Adresszeile
- Parameter werden durch das Zeichen ? In der URL eingeleitet
- Parameter werden in PHP im Array $_GET gespeichert
- Nicht geeignet zur Übertragung großer Datenmengen oder sensibler Daten

- Beispiel: `index.php?id=5&foo=bar`
  > Zugriff im PHP-Skript:

```php
$_GET['id']  // 5
$_GET['foo'] // bar
```

POST-Request

- Übertragung der Daten im HTTP-Body
  - > Daten sind nicht in der URL sichtbar

- Zugriff auf POST-Daten in PHP:

```php
$_POST['text']   // liefert POST-Parameter text
```

Validierung der Nutzereingaben

- Beschränkungen des Clients können umgangen werden
- Beispiel:

```
<input type="text" name="plz" maxlength="5" />
```

- Parameter kann trotzdem mehr als 5 Zeichen haben
  - > User umgeht aktiv die Beschränkung
  - > Client unterstützt/versteht die Beschränkung nicht
- Serverseitige Prüfung von Nutzerdaten ist zwingend erforderlich

- **„Never trust the client“**

Filter-Funktionen

- Vereinfacht Validierung der Eingaben
- Liefern *null*, *false* oder eine nach *$filter* gültige Variable zurück
  - > *FILTER_VALIDATE_** validiert Eingaben (alles oder nichts!)
  - > *FILTER_SANITIZE_** korrigiert Eingaben

```
mixed filter_input($type, $variable_name, $filter
      [, $options])
```

> Filtert GET/POST-Eingaben

```
mixed filter_var($variable, $filter [, $options])
```

> Filtert Variablen oder Werte

Weitere Informationen: http://php.net/manual/en/filter.filters.php

filter_input-Beispiel:

```php
$heightInCm = filter_input(INPUT_GET, 'height', FILTER_VALIDATE_FLOAT);
if ($heightInCm === null) {
    // Parameter 'height' nicht gesetzt
} else if ($heightInCm === false) {
    // Parameter 'height' ist kein gültiger Float-Wert
} else {
    // Gültiger Float-Wert, $heightInCm ist ein Double!
}

$range = array (
    'options' => array ('min_range' => 1, 'max_range' => 100)
);
$choice = filter_input(INPUT_POST, 'choice', FILTER_VALIDATE_INT, $range);
// $choice ist nun null, false oder ein Integer im Bereich von 1 bis 100

$data = filter_input(INPUT_POST, 'data', FILTER_UNSAFE_RAW);
// Parameter 'data' so lassen wie er ist
```

filter_var-Beispiel:

```php
$strFromDb = 'xss <h1>lol</h1>';
$output = filter_var($strFromDb, FILTER_SANITIZE_SPECIAL_CHARS);
    // 'xss &#60;h1&#62;lol&#60;/h1&#62;'

$number = filter_var('123 456 abc 7', FILTER_SANITIZE_NUMBER_INT);
    // '1234567', $number ist immer noch vom Typ String
```

Sinnvoll die existierenden Filter-Funktionen zu nutzen (statt eigene nachzubauen)

- Auch ein Filter, wie *FILTER_UNSAFE_RAW* kann sinnvoll sein um anderen Entwicklern sofort klar zu machen, dass die Variable „unsicher“ ist

Weitere Informationen: http://php.net/manual/en/filter.filters.php

Datenbankzugriff

Datenhaltung mittels Dateien nicht zu empfehlen

- Datenbanken!

Die meisten gängigen Datenbanken lassen sich mit PHP nutzen

- (Oracle) MySQL
- MariaDB
- Oracle Database
- SQLite
- MS SQL
- PostgreSQL
- IBM Informix
- MongoDB (NoSQL)

Wir nutzen MySQL

Weitere Informationen (nicht wirklich…): http://howfuckedismydatabase.com/

mysql_* Funktionen

- Unübersichtliche API, da sehr alt
- Keine OOP
- Prepared Statements nicht möglich

mysqli_* Funktionen/Klassen (MySQL *improved*)

- OOP
- Unterstützung von Prepared Statements
- „Cutting edge"-MySQL-Support

Weitere Informationen: http://php.net/manual/en/mysqli.overview.php

PDO (PHP Data Objects)

- OOP
- Unterstützung von Prepared Statements
- Unterstützung vieler Datenbanktypen
  - > Migration der Datenbank mit wenig Aufwand möglich

Wir nutzen PDO als Verbindungsklasse

- Grundsätzliches funktioniert ähnlich zu mysqli

Weitere Informationen: http://php.net/manual/de/book.pdo.php

Verbindung aufbauen

```php
$dsn = 'mysql:dbname=testdb;host=127.0.0.1';
$user = 'dbuser';
$password = 'dbpass';

try {
    $dbh = new PDO($dsn, $user, $password);
} catch (PDOException $e) {
    echo 'Connection failed: ' . $e->getMessage();
}
```

SQL-Abfrage mittels PDO::query

```php
$sth = $dbh->query("SELECT * FROM `user`
                    WHERE id = '" . $_GET['id'] . "'");

$row = $sth->fetch();
echo $row['name'];
```

- Anfällig für SQL Injections
- Sollte für Abfragen nicht benutzt werden

Alternative: Prepared Statements

Exkurs: SQL Injection

- Fehlende Validierung/Maskierung von Benutzereingaben führt zu ungewollten Datenbankstatements
- Beispiel:

	Erwarteter Aufruf
Aufruf	`http://example.com/page.php?id=42`
Erzeugtes SQL	`SELECT titel, text FROM artikel WHERE ID=42;`

	Angriff mittels SQL-Injection
Aufruf	`http://example.com/page.php?id=42;DROP+TABLE+user`
Erzeugtes SQL	`SELECT titel, text FROM artikel WHERE ID=42;DROP TABLE user;`

Exkurs: SQL Injection

- Zweites Beispiel:

	Erwarteter Aufruf

Aufruf	`http://example.com/login.php (POST)`
	
	`user=admin&pw=honigkuchen`

Erzeugtes SQL	`SELECT id FROM user WHERE name = 'admin'` `AND pw = 'cee199b741247512eb298a34c1fa18f5ca704bae'`

	Angriff mittels SQL-Injection

Aufruf	`http://example.com/login.php`
	
	`user=admin'+OR+1='1&pw=asdf`

Erzeugtes SQL	`SELECT id FROM user WHERE name = 'admin' OR 1='1'` `AND pw = '3da541559918a808c2402bba5012f6c60b27661c'`

Weitere Informationen: https://www.owasp.org/index.php/SQL_Injection

Prepared Statements

- Vorbereitete Anweisung wird an die Datenbank gesendet
  - > Platzhalter für Variablen

- Ausführung des Statements nach Einsetzen der Variablen
  - > Datenbanksystem kümmert sich um das Escapen
  - > Verhindert SQL-Injections und prüft die Gültigkeit von Parametern

- Geschwindigkeitsvorteil bei mehrfachen Ausführen
  - > Statement liegt dem Datenbanksystem bereits vor

Weitere Informationen: http://php.net/manual/de/pdo.prepared-statements.php

Prepared Statements

- Beispiel

```php
$calories = 100;
$color = 'red';

$sth = $dbh->prepare('SELECT name FROM fruits
    WHERE calories < :calories AND color = :color');
$sth->bindParam(':calories', $calories);
$sth->bindParam(':color', $color);

$sth->execute();
while($row = $sth->fetch()){
    echo $row['name'] . '<br/>';
}
```

SQL-Abfrage mittels Prepared Statement

```
PDOStatement PDO::prepare($statement)
```

> Bereitet ein SQL-Statement vor (wird an den DB-Server gesendet)

```
bool PDOStatement::bindParam($parameter, &$variable)
```

> Bindet die PHP-Variable an den Platzhalter im SQL-Statement
> Wird die Variable verändert, wirkt sich das auf das Statement aus

```
bool PDOStatement::bindValue($parameter, $value)
```

> Bindet den Wert an den Platzhalter im SQL-Statement

```
bool PDOStatement::execute()
```

> SQL-Statement mit angegebenen Daten ausführen

SQL-Abfrage mittels Prepared Statement

```
mixed PDOStatement::fetch([ $fetch_style ])
```

> Liefert Zeile des Ergebnisses
> Mittels $fetch_style kann angegeben werden in welcher Form die Zeile zurückgeben werden soll

```
Array PDOStatement::fetchAll([ $fetch_style ])
```

> Liefert ein Array mit allen Ergebniszeilen zurück

```
bool PDOStatement::closeCursor()
```

> Beendet die Anfrage
> Je nach Datenbank notwendig, bevor ein zweites Statement ausgeführt werden kann. Bei MySQL i.d.R. nicht notwendig

Weitere Informationen: http://wiki.hashphp.org/PDO_Tutorial_for_MySQL_Developers

Sichere Speicherung von Passwörtern

- Wir wollen Passwörter nicht im Klartext speichern
  - > Gefahr, dass die Datenbank in falsche Hände gerät
  - > Nutzer verwenden oft das gleiche Passwort auf mehreren Seiten
- Besser: Hash-Werte speichern

- Beispiel:

```
echo password_hash('superPW123', PASSWORD_DEFAULT);
// liefert z.B. $2y$10$dNvqygJOO4B/9HHS/aedre.BIdVclwmDK7d4tr0yxS4q5PRl0cRUG
// oder         $2y$10$haURtemZLULGRZatmmfeKuemcoUHunYpBUfE6EqZ6lnJW579gXa56
// oder         $2y$10$jNafet8iW5/NgmC85mJbde2ipWgpwvyuNA.YGYPxYxEUhA1rzxnzy
```

- Gleicher Funktionsaufruf liefert unterschiedliche Werte?
  - > Salt!
  - > Mehr dazu später (im Sicherheitsteil)

Weitere Informationen: http://php.net/manual/en/function.password-hash.php

Passwort-API

- Einfache Möglichkeit Passwörter sicher zu speichern

```
string password_hash($password, $algo [, $options])
```

> Beispiel:

```
password_hash('superPassword', PASSWORD_BCRYPT);
```

- Überprüfung ob Passwort korrekt ist:

```
boolean password_verify($password, $hash)
```

> Beispiel:

```
if (password_verify($input, $storedPw)) {
    // correct!
}
```

Weitere Informationen: http://php.net/manual/en/function.password-hash.php

Zustände
Cookies & Sessions

Zustand des PHP Skriptes geht nach Ausführung verloren

- Variablen sind nur für einen Aufruf gültig

Wie können mehrstufige Operationen/Transaktionen vorgenommen werden?

- Der Server muss erkennen, dass der nächste Aufruf der Seite in einem Kontext geschieht
- ISO/OSI, Schicht 5? Session?
  > Nein, im HTTP-Protokoll ist das Aufgabe der Applikation!

Persistente PHP Skripte

- Parameter, der den aktuellen „Schritt" speichert
- Cookies & Sessions bieten die Möglichkeit den Inhalt von Variablen länger zu speichern

Zustände
Cookies

Cookies bieten die Möglichkeit clientseitig Daten zu speichern

- Benutzer muss sich nicht mehrfach anmelden
- Aber auch Tracking des Nutzerverhaltens möglich

- Cookies werden im Header der Seite zurückgeliefert
- Das Setzen von Cookies muss daher erfolgen bevor Inhalt ausgegeben wird

- Beispiel:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value
Set-Cookie: foo=bar; Expires=Wed, 09 Jun 2021 10:18:14 GMT

(Inhalt der Seite)
```

Speicherung des Cookies als „Datei“ auf dem Client

- Tatsächlich meist mit Hilfe von Datenbanken realisiert

Client sendet beim nächsten Aufruf der Webseite den Cookie mit

- Nur die Webseite, die den Cookie gesetzt hat, kann den Cookie lesen

- Beispiel:

```
GET / HTTP/1.1
Host: www.example.org
Cookie: name=value; foo=bar
…
```

Setzen eines Cookies in PHP

```
bool setcookie($name, $value [, $expire [, $path]])
```

> Speichert Cookie mit Namen $name und Wert $value
> $expire ist ein Unix Timestamp (Integer), der angibt wann der Cookie verfällt
> $path gibt an von welchen Verzeichnissen der Cookie gelesen werden kann
> Weitere Parameter möglich (siehe PHP-Doku)

- Beispiel:

```
setcookie('fontSize', '+1', time()+60*60*24*30, '/');
```

> Cookie „fontSize" mit Wert „+1" gesetzt, der in 30 Tagen abläuft und von allen Verzeichnissen in der Domain aus gelesen werden kann

Weitere Informationen: http://php.net/manual/en/function.setcookie.php

Auslesen eines Cookies über das Cookie-Array:

- Beispiel:

```
$_COOKIE['fontSize'] // liefert '+1' zurück
```

- Kann nicht im selben Skriptdurchlauf gespeichert und wieder gelesen werden
  - > Cookie wird erst beim Senden des Clients im Array eingetragen
  - > Erst der nächste Skriptdurchlauf kann den Cookie lesen

Zustände
Sessions

Cookies
- Speichern alle Informationen clientseitig ab
- Daten können vom Benutzer manipuliert werden
- Nur die Speicherung eines Strings möglich

Sessions
- Speichern die Daten serverseitig ab
- Client erhält nur eine Session-ID (als Cookie)
  - > Um das Setzen des Cookies kümmert sich PHP
  - > Alternativ auch möglich die Session-ID per URL-Rewriting zu übergeben
    - PHP-Option: session.use_cookies
- Daten, die per Session gespeichert werden, werden auch nur auf dem Server gespeichert
  - > Keine Datenmanipulation möglich
  - > Komplexe Variablen (Arrays,Objekte) sind möglich

Weitere Informationen: http://php.net/manual/de/intro.session.php

Benutzung in PHP

```
bool session_start()
```

- Erzeugt eine Session oder nimmt die aktuelle wieder auf
  - Generiert falls nötig Session-ID und setzt den entsprechenden Cookie
- Muss aufgerufen werden, bevor irgend etwas an den Browser geschickt wird

```
bool session_destroy()
```

- Beendet Sitzung und löscht Variablen

Setzen und Lesen von Session-Werten

- Im `$_SESSION`-Array können Werte gespeichert werden, die seitenübergreifen verfügbar sind

- Beispiel:

```php
$_SESSION['logged_in'] = true;
```

- Zugriff analog über das Array

Beispiel

```php
<?php
    session_start();

    if (!isset($_SESSION['zaehler'])) {
        $_SESSION['zaehler'] = 1;
    } else {
        $_SESSION['zaehler']++;
    }

    echo 'Sie haben diese Seite ' . $_SESSION['zaehler'] .
        ' mal aufgerufen';
?>
```

PHP Output Buffering

Warum sehe ich die Ausgabe „Hello World!“ erst nach 5 Sekunden?

```php
<?php
    echo 'Hello ';
    sleep(5);
    echo 'World!';
?>
```

Und wieso sehe ich die Ausgabe schon früher?

```php
<?php
    $size = 1024 * 8;
    for($i = 1; $i <= $size; $i++) {
        echo '.';
    }
    sleep(5);
    echo 'Hello World';
?>
```

PHP Output Buffering

Output Buffering

- Mechanismus bei dem die Ausgaben nicht direkt zum Browser verschickt werden, sondern vielmehr in „Chunks“ eingeteilt werden
- Die Ausgaben werden also gepuffert und (standardmäßig) in Einheiten von 4KB an den Browser geschickt

- Wenn dieser übertragen wurde (also bei 4K an Daten), dann kann nachher auch kein HTTP-Header mehr gesetzt werden
  - > Deshalb immer sicherstellen, dass Funktionen, die header verändern (header, session_start, setcookie) aufgerufen werden, bevor die ersten Daten im HTTP body übermittelt werden

Weitere Informationen: http://php.net/manual/de/book.outcontrol.php

PHP 7

PHP 6?

- „PHP 6“ galt als gescheitert, es gab nie ein Release

PHP 7

- Veröffentlicht: Dezember 2015
  - > Zum Vergleich: PHP 5.0 Juli 2004 veröffentlicht (11 Jahre dazwischen!)
- Schneller dank Refactoring der Codebasis
- Neue Features
  - > Typdeklarationen, Fehlerbehandlung, …
- Abwärtskompatiblität
  - > teilweise aufgegeben: http://php.net/manual/de/migration70.incompatible.php

Weitere Informationen: http://php.net/manual/migration70.new-features.php

Typdeklarationen

```
function foo(Person $a, string $b) : int {
    // Erwartet, dass $a ein Objekt vom Typ Person ist
    // und, dass $b ein String ist
    // Gibt eine Zahl zurück
};
```

- Strike Typisierung möglich

  ```
  declare(strict_types=1);
  ```

  > Verhindert automatisches casten (z.B. von int in String)

- Einfacher Fehlerquellen zu finden
- Schwieriger Spaghetticode zu schreiben

Fehlerbehandlung

- Fatale Fehler führen nun auch zu einer Exception
  - > Neue Exception-Klasse Error (abgeleitet von Throwable)
- Bisher wurde das Skript einfach angehalten und erzeugte so nette weiße Seiten (leer)

Neue Operatoren

- Spaceship-Operator <=>
  - > Vergleich zweier Werte auf größer, gleich und kleiner
  - > 2 <=> 1 liefert 1 1 <=> 1 liefert 0 1<=> 2 liefert -1
- Ist-Null-Check Operator ??
  - > Setzt Variable, wenn sie noch nicht gesetzt ist

```
$name = $firstName ?? „Guest“;
```

statt

```
if (!empty($firstName)) $name = $firstName;
else $name = "Guest";
```

Composer
Module in PHP

Anwendungsfall:

- Man benötigt bestimmte Funktionalitäten

Beispiel:

- Saubere Trennung von HTML und PHP-Code erwünscht
  > Lösung: Template-Engine

- Muss ich die Template-Engine jetzt selber implementieren?

NEIN!*

Composer
Module in PHP

Was ist der Composer?

- Dependency Manager (Verwaltung von Abhängigkeiten)
  - > Modul-Organisation
  - > Beschaffung aller benötigten Pakete pro Projekt
- zugehöriges Repository: Packagist.org
- Abhängigkeiten werden rekursiv aufgelöst! (Beispiel „zendframework“)
  - > https://packagist.org/packages/zendframework/zendframework

Existierende Module nutzen!

- Viele Funktionen (z.B. Template-Engine) existieren bereits und können im eigenen Projekt verwendet werden

Nutzung

- Composer benötigt zum Durchführen des Downloads+Installation der Pakete und dem etwaigen Einspielen von Updates eine Datei mit dem Namen **composer.json** im Hauptverzeichnis der Anwendung
- In dieser wird das require-Objekt gefüllt (mit Angabe der Version)
- Packagist.org hilft dabei, dass passende „require“ oder „include“ zu finden
- `php composer.phar install` im Verzeichnis startet den Download und Installationsvorgang
- `composer.phar update` ein Update

Composer

1. **composer.json**

```
{
    "require":  {
            "monolog/monolog": "1.0.*"
    }
}
```

2. **„composer install“**

3. Es gibt zum automatisierten Laden von Bibliotheken ein autoload-Skript:

```
require __DIR__ . '/vendor/autoload.php';
```

PHP-Frameworks

Composer ermöglicht das Wiederverwenden und Teilen einzelner Komponenten

- Evtl. wollen wir ein „Gesamtpaket“ zum Starten
- Ein Framework enthält bereits „Best Practices“ beim Entwickeln und löst viele bereits gelöste Probleme

Die meisten Frameworks sind komponentenbasiert

- Typische Komponenten:
  - > Benutzerverwaltung
  - > Datenbankzugriff
    - Object Relational Mapping
  - > MVC-Unterstützung
  - > Caching-Systeme
  - > Vereinfachung des Zusammenspiels von JavaScript und PHP

PHP-Frameworks

Zend Framework

- Das wohl bekannteste PHP-Framework
  - > Strikt objektorientiert

Symfony

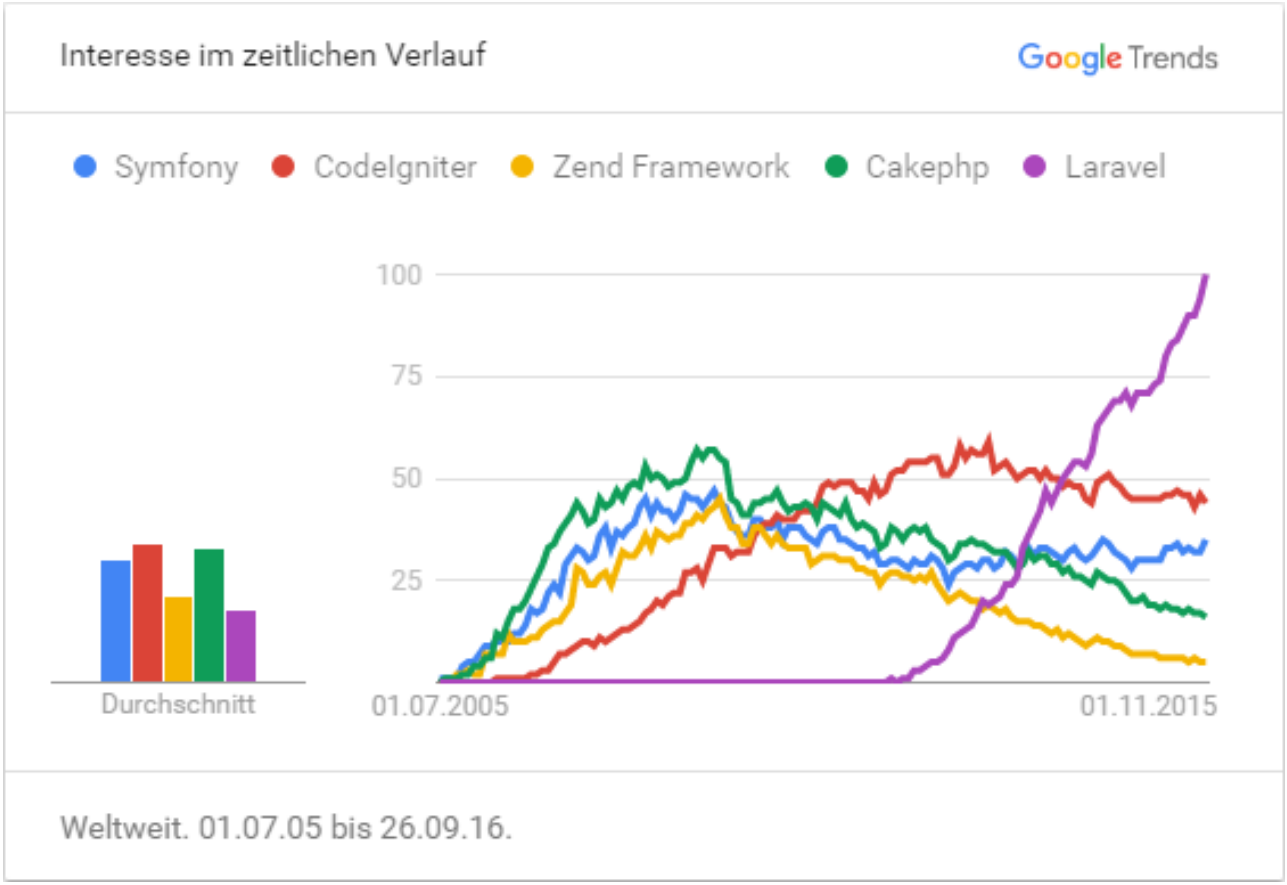
- Beeinflusst durch andere Web Frameworks (Ruby on Rails, Django)
- Benutzt viele andere Opensource-Frameworks

Laravel

- Recht „neu“, aber bereits weit verbreitet

Sinnvoll sich bei Projektstart mit Frameworks zu beschäftigen

PHP-Frameworks



Quelle: https://trends.google.de/trends/explore?date=all&q=Symfony,CodeIgniter,Zend%20Framework,Cakephp,Laravel