

Web 2.0 Technologien 2

Kapitel 2:

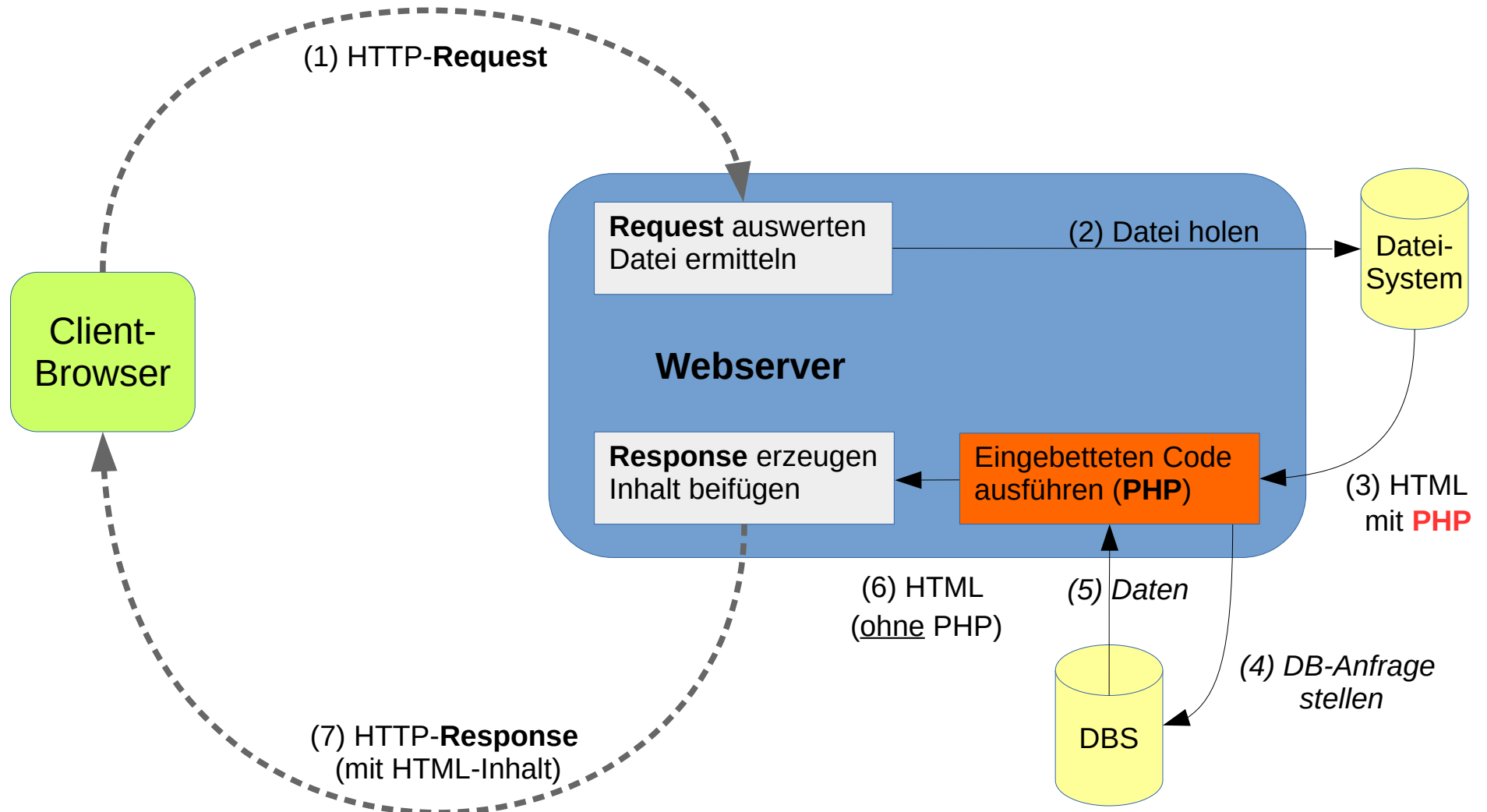
Serverseitige Techniken: **PHP**

Serverseitige Techniken

- **Nächste Schritte: Serverseitige Techniken**
 - **Statische Inhalte** ausliefern
 - **Webserver** z.B. **Apache**, nginx, IIS, ...
 - **Dynamische Inhalte** erzeugen, komplexe Anfragen verarbeiten
 - Serverseitige **Programmierung** z.B. **PHP**, ASP, JSP, .NET, Node.js, ...
 - **Datenmanagement** (Daten speichern, abrufen, verknüpfen)
 - **Datenbank**-Schnittstelle z.B. zu **MySQL**, Postgres, DB2, ...
- **Verbreitete Standard-Lösung: LAMP**
 - Betriebssystem **Linux** (oder Windows → **XAMPP**)
 - Webserver **Apache**, Datenbank **MySQL**, Sprache **PHP**

Webserver: intern generierte Webseite

- Abruf einer **dynamisch** erzeugten Webseite



PHP-Grundlagen

- **PHP** = „**P**HP: **H**ypertext **P**reprocessor“
- **Idee:**
 - *Man fügt in eine fertige HTML-Seite genau dort Code ein, wo ein berechneter Inhalt landen werden soll.*
 - *Beispiel:*

`<body> 1 + 2 = <?php echo 1+2; ?> </body>`

- **blau** ist hier HTML,
- **rot** ist PHP-Code,
- **lila+fett** ist die syntaktische Klammerung des PHP-Codes
- Lädt man die Webseite, erhält man die Ausgabe

1 + 2 = 3

- Genauer: Der vom Server gelieferte HTML-Text lautet hier

`<body> 1 + 2 = 3 </body>`

PHP-Grundlagen

- **Idee**

- Der PHP-Code wird also auf dem Server ausgeführt und durch die Ausgabe (*echo / print*) des Codes ersetzt.
 - Außerhalb des Webservers ist kein PHP-Quellcode mehr zu finden.
- Der resultierende Text darf alles enthalten was HTML/CSS/... kann
 - also auch **HTML**-Tags, **CSS**-Styledefinitionen, **Javascript**, ...
- Der PHP-Code muss nicht vorab übersetzt werden.
 - Er wird bei jedem Aufruf durch den Webserver direkt ausgeführt.

- **Vorteil:**

- Sehr effiziente Software-Entwicklung (kurze Zyklen)
- Leichter Einstieg, schnelle Erfolge

- **Nachteile:**

- fehleranfällig, u.a. keine statische Typ-Prüfung

PHP-Grundlagen

- **PHP ist leicht zu lernen**

- Syntax ähnelt C++ / Java / Javascript
- Etwas ungewohnt: Variablennamen fangen mit „\$“ an.

- **Beispiel: Von 1 bis 10 zählen**

HTML

```
<h1>PHP-Schleife</h1>
```

```
<?php
```

```
    $limit = 10;
```

```
    $i = 1;
```

```
    while ($i <= $limit) {
```

```
        echo "Loop number " . $i . " <br> \n";
```

```
        $i = $i + 1;
```

```
    }
```

```
?>
```

PHP

```
<h1>PHP-Schleife</h1>
```

```
Loop number 1 <br>
```

```
Loop number 2 <br>
```

```
Loop number 3 <br>
```

```
Loop number 4 <br>
```

```
Loop number 5 <br>
```

```
Loop number 6 <br>
```

```
Loop number 7 <br>
```

```
Loop number 8 <br>
```

```
Loop number 9 <br>
```

```
Loop number 10 <br>
```

Die Ausgaben in PHP (echo) ersetzen später den PHP-Code

PHP-Grundlagen

- **PHP und HTML können stark verzahnt werden**

- Wird von PHP auf HTML zurück geschaltet, so wirkt das, als ob der HTML-Code dort ausgegeben würde. Das funktioniert auch innerhalb von Kontrollstrukturen!

- **Beispiel: Von 1 bis 10 zählen**

HTML

```
<h1>PHP-Schleife</h1>
```

```
<?php
```

```
    $limit = 10;
```

```
    $i = 1;
```

```
    while ($i <= $limit) {
```

```
?>
```

```
    Loop number <?php echo $i; ?> <br>
```

```
<?php
```

```
    $i = $i + 1;
```

```
    }
```

```
?>
```

PHP

HTML

PHP

PHP

```
<h1>PHP-Schleife</h1>
Loop number 1 <br>
Loop number 2 <br>
Loop number 3 <br>
Loop number 4 <br>
Loop number 5 <br>
Loop number 6 <br>
Loop number 7 <br>
Loop number 8 <br>
Loop number 9 <br>
Loop number 10 <br>
```

Die (HTML-) Zeile wird in der (PHP-) while-Schleife 10 mal ausgegeben.

PHP-Grundlagen

- **PHP-Sprachreferenz**

- **php.net** (*offizielle Referenz*)
 - <http://php.net/manual/de/langref.php>
 - Sehr detailliert (vollständige Sprachbeschreibung)

- **PHP-Tutorials**

- php.net:
 - <http://de2.php.net/manual/de/>
 - Einstieg: <http://de2.php.net/manual/de/intro-what-is.php>
 - Leider sehr kurz
- w3schools.com:
 - <http://www.w3schools.com/php/>
 - Hinweis: Die englische Fassung hat weniger Fehler
- **Quakenet/#php Tutorial**
 - <http://unix.oppserver.net/php-tut/>
 - Sehr ausführlich und gut gemacht!

PHP-Grundlagen: Ausführung

- Standard: **PHP-Scripte durch Webserver ausführen**
 - PHP-Scripte werden auf dem Webserver als Dateien abgelegt.
 - Sie enthalten verzahnt **HTML-Quelltext** und **eingebettete PHP-Scripte**
 - Der Webserver erkennt PHP-Dateien an der Datei-Namensendung „.php“ und behandelt sie entsprechend
 - Der Webserver muss dazu ggf. konfiguriert bzw. erweitert werden (Z.B. Apache: **mod-php**, z.B. aus Debian-Paket **libapache2-mod-php**)
 - Der Webserver kann auch konfiguriert werden, z.B. alle HTML-Dateien als PHP-Dateien zu behandeln.
 - Rufen wir die Datei über den Webserver ab, werden ...
 - die PHP-Scripte ausgeführt und durch ihre Ausgabe ersetzt
 - Es ist von außen nicht mehr zu erkennen, welche HTML-Teile aus PHP-Code stammen.
 - die resultierende HTML-Datei wie gewohnt an den aufrufenden Browser geschickt und dort dargestellt. (→ Inhalt von W2T-1)
 - Das erzeugte HTML-Dokument muss als Ganzes syntaktisch korrekt sein, soll den gewünschten Inhalt (→ Elementbaum) haben und ggf. im Quelltext gut formatiert sein.

PHP-Grundlagen: Ausführung

- **PHP-Scripte durch Webserver ausführen (Beispiel)**

- Beispiel: (Übungs-Testserver, hier `scilab-0100.cs.uni-kl.de`)

- Datei `test.php` unter `~/htdocs/` ablegen:

```
<!DOCTYPE html>
<html>
  <body>
    Hallo <?php echo "John\n<b>Doe</b>"; ?>!
  </body>
</html>
```

- Aufruf der URL `http://scilab-0100.cs.uni-kl.de/test.php` im Browser

- Anzeige im Browser:

Hallo John **Doe!**

Übungsfrage:
Warum sieht man
den Umbruch nicht
im Browser?

- Quelltext-Anzeige
im Browser (Ctrl-U):

```
<!DOCTYPE html>
<html>
  <body>
    Hallo John
    <b>Doe</b>!
  </body>
</html>
```

"\n" erzeugt
Zeilenumbruch

PHP-Grundlagen: Ausführung

- Zum Testen: **PHP-Scripte manuell** ausführen

- Das Ausführen der PHP-Anteile einer PHP-Datei kann auch manuell erfolgen.
- Dazu rufen wir den php-Interpreter mit dem Kommando „`php -f Dateiname`“ auf

- Datei `test.php`:

```
<body>
  Hallo <?php echo "John\n<b>Doe</b>"; ?>!
</body>
```

- Aufruf von php:

```
php -f test.php
```

- Ausgabe:

```
<body>
  Hallo John
<b>Doe</b>!
</body>
```

- Die Kommandozeilenversion von PHP muss dazu ggf. installiert werden (Z.B. Debian-Paket `php-cli`)

PHP-Grundlagen: Ausführung

- Praxistipp: **PHP als lokale Scriptsprache**

- Man kann PHP auch als Scriptsprache benutzen, um Kommandozeilen-Tools zu realisieren
 - Das ist z.B. nützlich, um kleine Datenbank-Werkzeuge zu bauen.

- Beispiel:

- Datei „myscript.php“ anlegen:

```
#!/usr/bin/php -f  
<?php  
    echo "Script-Demo -- übergebener Parameter:\n";  
    print_r($argv);  
?>
```

Sorgt für den Aufruf von
“/usr/bin/php -f myscript.php“.
Zeile wird nicht ausgegeben.

- Datei als Ausführbar kennzeichnen:

```
chmod u+x myscript.php
```

- Script aufrufen:

```
./myscript.php xxx
```

```
Script-Demo -- übergebener Parameter:  
Array  
(  
    [0] => ./php-script.php  
    [1] => xxx  
)
```

PHP-Grundlagen

- **PHP-Kommentare**

- Einzeilige Kommentare: Ab „//“ oder „#“ bis zum Ende der Zeile
- Mehrzeilige Kommentare: Zwischen „/*“ und „*/“

```
<!DOCTYPE html>
<html>
<body>
<?php
    // A single line comment    echo "1";
    # A single line comment    echo "2";
    /* A multi line
       comment */              echo "3";
                               echo "4";
?>
</body>
</html>
```

- **Übungsfrage: Welche der echo-Anweisungen werden ausgegeben?**
- PHP-Skripte werden auch innerhalb von HTML-Kommentaren ausgeführt.

```
<!-- HTML Kommentar mit <?php echo "PHP"; ?> -->
```

PHP

```
<!-- HTML Kommentar mit PHP -->
```

PHP-Grundlagen

• Variablen

- Variablennamen beginnen immer mit einem **\$**-Zeichen
 - Vergisst man das, wird eine Konstante mit dem Namen (erfolglos?) gesucht
 - Variablennamen sind Case-sensitive
- Variablen müssen nicht deklariert werden
 - **Variablen haben keinen Typ** (aber ihr jeweils aktueller **Wert** hat einen Typ!)
 - Man kann Variablen einfach verwenden, z.B. einen Wert zuweisen

• Zuweisung von Werten zu Variablen

- Variablen, die noch keine Zuweisung erhalten haben, haben den Wert **NULL**
- Einen Wert zuweisen kann man mit dem Zuweisung-Operator **=**

```
<?php $vorname = "John"; ?>  
Hallo <?php echo $vorname; ?>!
```

PHP

Hallo John!

PHP-Grundlagen

• Sequenzen von Anweisungen

- Anweisungen werden mit Semikolon („;“) voneinander getrennt
 - Kommt dahinter keine Anweisung, kann (aber muss kein) Semikolon stehen

```
<?php
    $vorname = "John";
    $nachname = "Doe";
?>
<body>
Hallo <?php echo $vorname; ?>!
</body>
```

Semikolon hier
nicht nötig.

• Groß-Klein-Schreibung

- Schlüsselworte (z.B. „if“), Funktionsnamen und Konstanten (z.B. „true“) sind nicht Case-Sensitive
 - Sie können gemischt groß oder klein geschrieben werden
- Variablennamen sind aber Case-Sensitiv (s.o.)
 - \$a und \$A sind zwei verschiedene Variablen!

PHP-Grundlagen

- **Kontrollstrukturen**

- PHP kennt typische Kontrollstrukturen wie in anderen (C-/Java-artigen) imperativen Programmiersprachen

- `if` (ausdruck) anweisung
- `if` (ausdruck) anweisung `else` anweisung
- `if` (ausdruck) anweisung `elseif` (ausdruck) anweisung
- `while` (ausdruck) anweisung
- `foreach` (array_expression `as` \$value) anweisung
- `foreach` (array_expression `as` \$key => \$value) anweisung
- `for` (expr1; expr2; expr3) anweisung
- ...

- Beispiel

- ```
foreach (array(3,5,7) as $k => $v) {
 echo "Wert für $k ist $v\n";
}
```

- <https://www.php.net/manual/de/language.control-structures.php>



# PHP-Grundlagen

## • Mehrere Arten von String-Literalen

– Einfache Anführungszeichen: 'Text'

- Inhalt wird 1:1 übernommen

– Doppelte Anführungszeichen: "Text"

- Bestimmte Zeichenketten werden interpretiert und durch Sonderzeichen ersetzt

- "\n" → LF (Linefeed = Neue Zeile),
- "\r" → CR (Carriage Return),
- "\\" → '\',
- "\" → '\"',
- ...

- Es gibt noch weitere Sequenzen:

<https://www.php.net/manual/de/language.types.string.php#language.types.string.syntax.double>

- Variablennamen (`$name` und `{ $name }`) werden durch ihren Wert ersetzt

Der Vollständigkeit halber:  
Es gibt noch 2 weitere Arten,  
*Heredoc* und *Nowdoc*

Nützlich wenn nach  
dem Var.-Namen z.B. ein  
Buchstabe steht.  
(Warum?)

Der Mechanismus ist  
sehr komplex, z.B. ist zur  
Ausgabe eines Array-Wertes  
auch so etwas möglich:  
"Ergebnis: \$arr[0]"

```
<?php
 $vorname = "John";
 $nachname = "Doe";
 echo "Hallo $vorname {$nachname}!"
?>
```

PHP →

Hallo John Doe!

# PHP-Grundlagen

- **String-Werte können mit „.“ verkettet werden**

- Der `.`-Operator verkettet zwei Strings

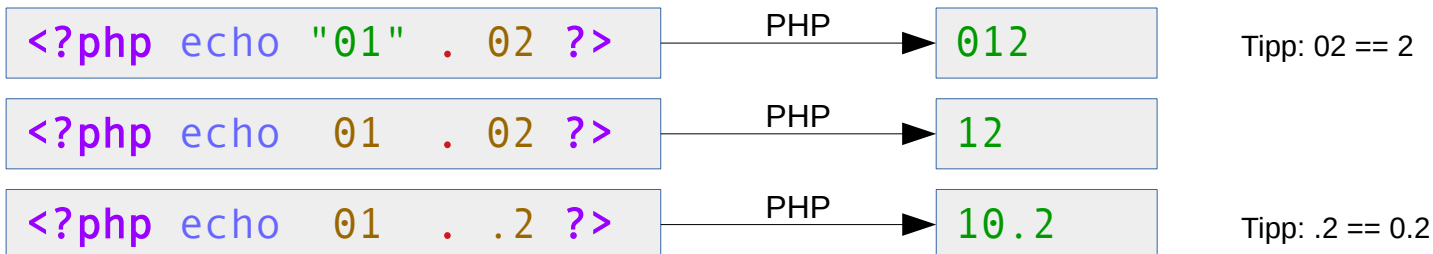
- Das folgende Script erzeugt die Ausgabe „Hallo John Doe!“

```
<?php
 $vorname = "John";
 $nachname = "Doe";
?>
Hallo <?php echo $vorname . " " . $nachname ?>!
```

- **Der `.`-Operator kann aber nur Strings verarbeiten**

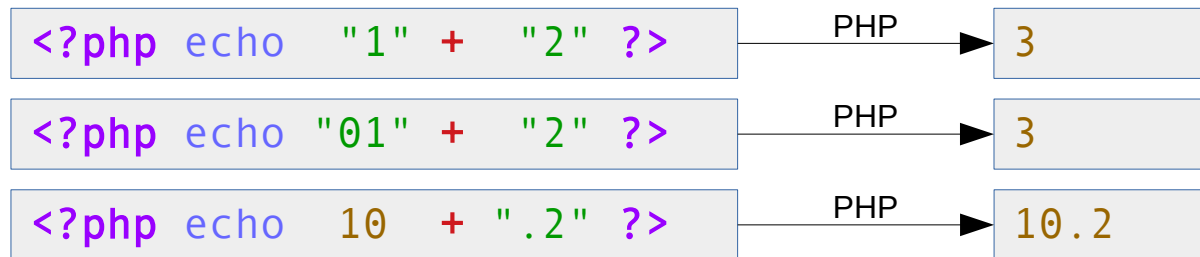
- Andere Typen werden ggf. in Strings **umgewandelt**.
- Das kann teilweise verblüffende Effekte haben.

Die Ergebnisse haben jeweils den Typ String



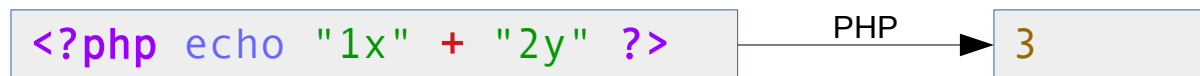
# PHP-Grundlagen

- **Typkonvertierungen** (Beispiel String → Integer)
  - Allgemein werden Typen in PHP bei Bedarf flexibel **konvertiert**
    - Bsp.: Numerische Operatoren (wie **+**, **-**, **\***) brauchen Zahlen als Parameter
      - Übergibt man z.B. einen String, wird dieser in eine Zahl konvertiert

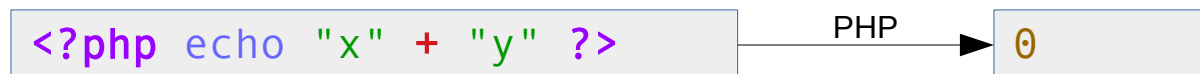


Die Ergebnisse haben hier jeweils den Typ integer oder float

- Die Konvertierung String nach Zahl endet, sobald der Rest nicht mehr als Zahl interpretiert werden kann



- Der leere String wird bei Umwandlung in eine Zahl als 0 interpretiert



# PHP-Grundlagen

---

- **Datentypen**

- Skalare Typen:

- `boolean`      Werte: `true`, `false`
    - `integer`      z.B. `123`, `-5`
    - `float`        z.B. `1.0`, `5.7e3 == 5700.0`, `1e-3 == 0.001`
    - `string`        z.B. `"Hallo"`

- Strukturierte Typen

- `array`        z.B. `array(2, 3, 5, 7)`  
z.B. `array("Hund", "Katze", 123)`  
z.B. `array("matnr"=>12345, "name"=>"Müller")`

- Object, Callable, Iterable

- Spezielle Typen

- Ressource
    - `NULL`        Wert: `NULL`

- Siehe <https://www.php.net/manual/de/language.types.php>

# PHP-Grundlagen

---

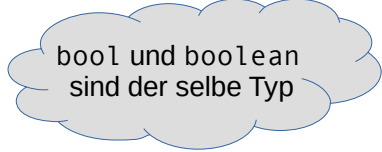
- **Explizite Typkonvertierungen**

- Typen können auch explizit konvertiert werden

- Dazu wird der Typname in Klammern vor den Ausdruck gestellt

- z.B. `(bool) 1` (Ergebnis: `true`)

- z.B. `(boolean) 0` (Ergebnis: `false`)



bool und boolean  
sind der selbe Typ

- Es gibt folgende Konvertierungen

- `(int)`, `(integer)` - cast to integer

- `(bool)`, `(boolean)` - cast to boolean

- `(float)`, `(double)`, `(real)` - cast to float

- `(string)` - cast to string

- `(array)` - cast to array

- `(object)` - cast to object

- `(unset)` - cast to NULL

# PHP-Grundlagen

---

- **Typkonvertierungen**

- Einige Typkonvertierungswerte sind besonders wichtig:
- Folgende Werte werden als **boolean false** interpretiert
  - **boolean false** selbst
  - **integer 0**
  - **float 0.0**
  - Der leere **string**, und der **string "0"**
  - Ein **array** ohne Elemente
  - Der spezielle Typ **NULL** (also auch nicht initialisierte Variablen)

Wir können alle diese Type z.B. direkt als if-Bedingung nutzen

- z.B.: `$list = array(1,2,3); /* ... */ if ($list) { ... }`
- Diese Werte werden bei der Konvertierung zu **integer** entsprechend auch als 0 interpretiert.
  - Beachten Sie die Anomalie bei den beiden String-Werten!
- <https://www.php.net/manual/de/language.types.type-juggling.php>

# PHP-Grundlagen

- **Ausdrücke (Expression): Zerlegung**

- Durch Operatoren (z.B. **+**) werden Ausdrücke (z.B. **1+2**) gebildet

- **Ausdruck-Analyse**

- Ausdrücke werden rekursiv (von oben nach unten) in **Teilausdrücke** zerlegt

**Ziel: Baumstruktur** zur Ausdrucksanalyse

- Knoten sind Ausdrücke, die nach unten zerlegt werden

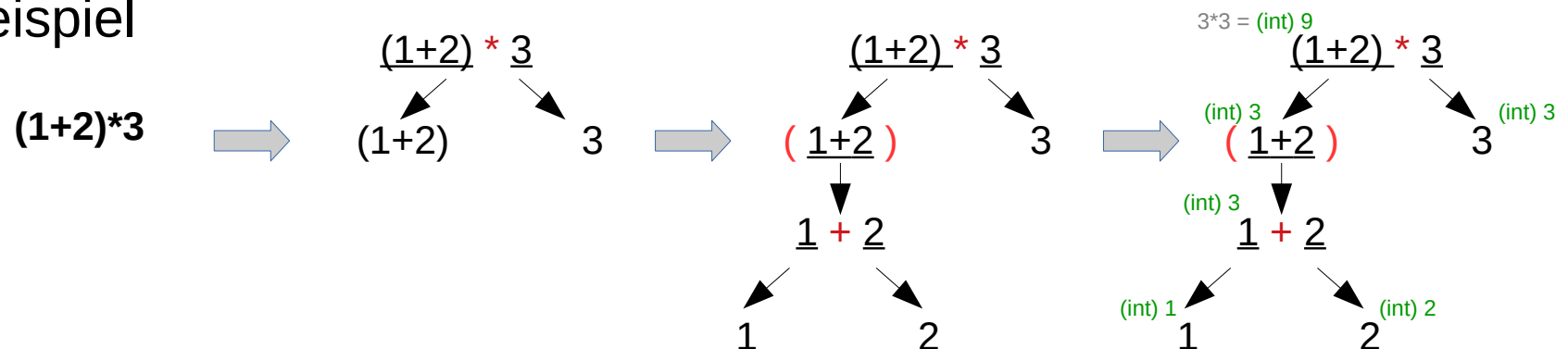
- Die Blätter sind schließlich nur noch atomare Ausdrücke (nicht mehr zerlegbar)

- die Werte und Typen der atomaren Ausdrücke (z.B. Variablen) bestimmt

- schrittweise (von unten nach oben) Werte u. Typen der Teilbäume bestimmt

**Ziel: Wert und Typ** des gesamten Baums bestimmen.

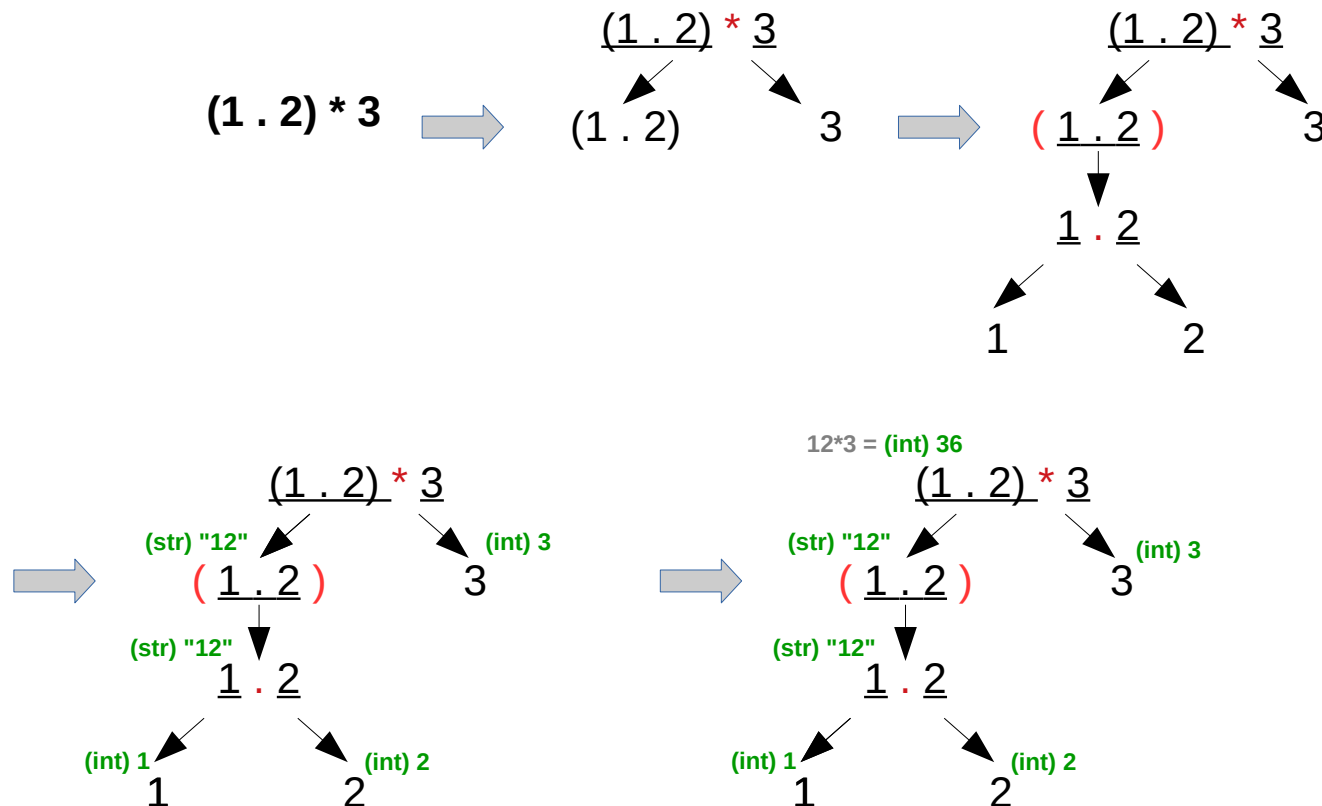
- **Beispiel**



# PHP-Grundlagen

- **Ausdrücke (Expression): Typkonvertierungen**

- Bei der Berechnung der Ergebnisse muss man auf implizite Typkonvertierungen achten
- Beispiel

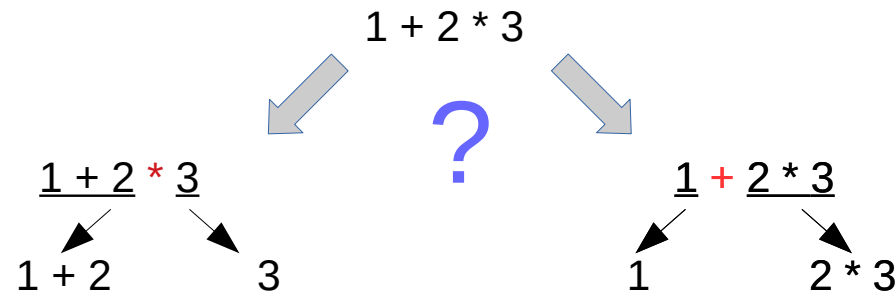




# PHP-Grundlagen

- **Ausdrücke (Expression): Präzedenzen**

- Die Zerlegung scheint manchmal nicht eindeutig zu sein



- Die Operatoren haben dazu eine **Rangfolge (Präzedenz)**

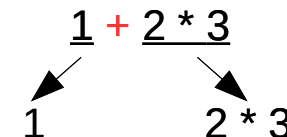
- <https://www.php.net/manual/de/language.operators.precedence.php>
- In der Tabelle höher stehende Werte „binden stärker“  
→ die schwächeren Operatoren werden immer zuerst zerlegt

Die Zeile „+ -“  
ist auf php.net  
leider um eine  
Spalte „verrutscht“.

- Ergebnis im obigen Beispiel:

- \* bindet stärker als +

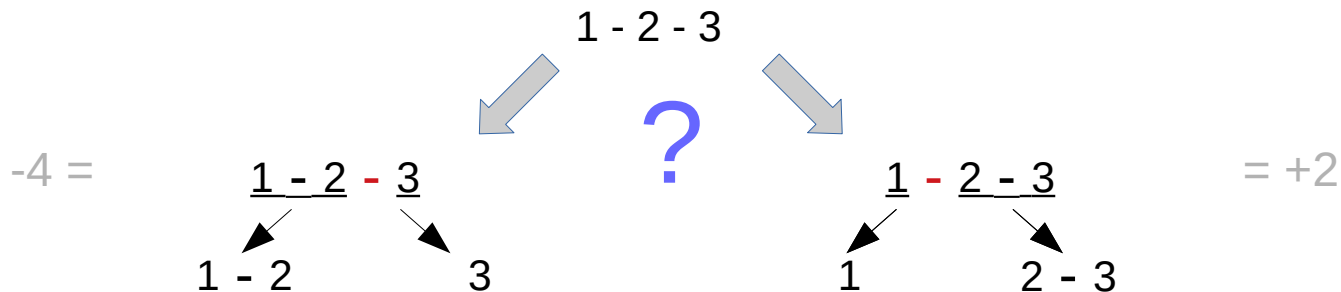
$1 + 2 * 3$



# PHP-Grundlagen

- **Ausdrücke (Expression): Assoziativität**

- Die Zerlegung scheint *weiterhin* manchmal nicht eindeutig zu sein

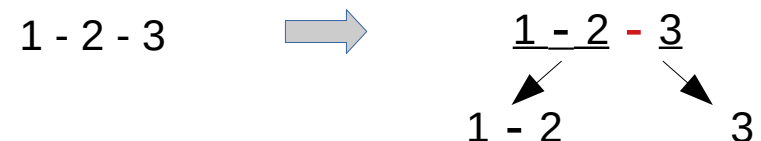


- Der Vorrang Operatoren gleicher Präzedenz (gleiche Zeile) wird über die **Assoziativität** (erste Spalte) geregelt

- <https://www.php.net/manual/de/language.operators.precedence.php>
- links-assoziativ bedeutet der linke Operator „bindet stärker“

- Ergebnis im obigen Beispiel:

- - (minus) ist links-assoziativ  
→ linker Operator bindet stärker



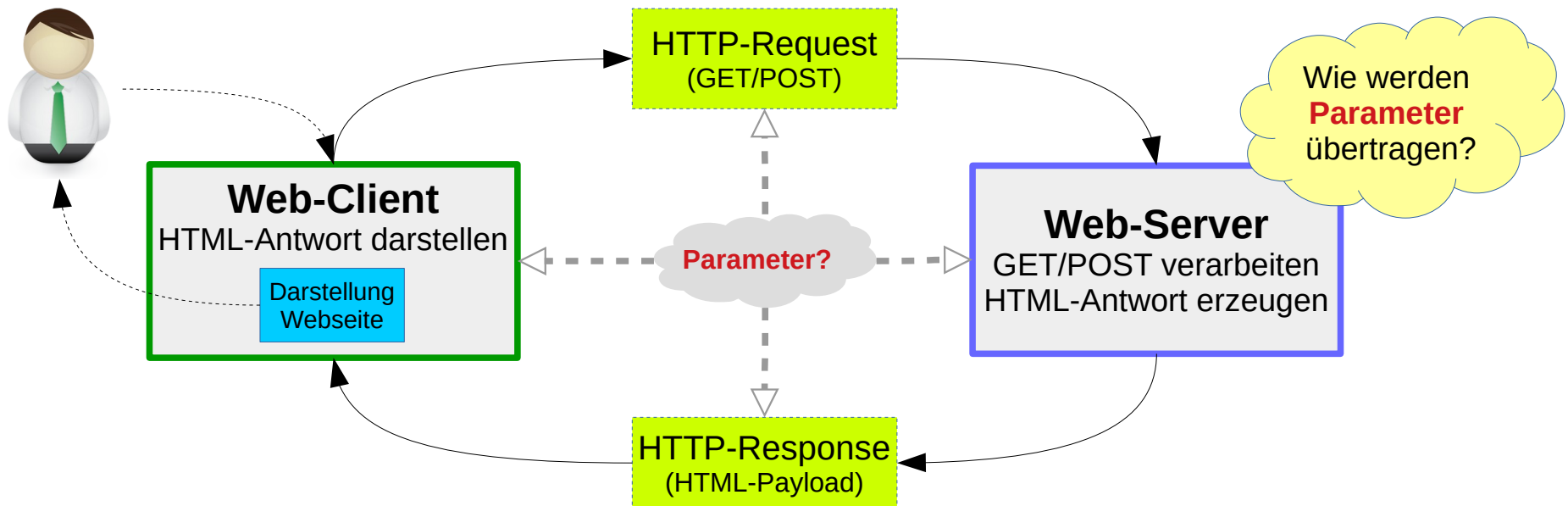
# Struktur von Web-Applikationen

---

- **Vorüberlegung zur Struktur von Web-Applikationen**
  - Wir machen nun einige Vorüberlegungen zur Struktur von Web-Applikationen
    - Austausch von Parametern und Daten
    - Authentifizierung und Autorisierung
  - Was sind die Schnittstellen und Lösungsansätze dazu in PHP?
- Wir ergänzen weitere Aspekte von PHP dabei jeweils bei Bedarf.

# Struktur von Web-Applikationen

- **Typische Kommunikationssequenz zwischen Client (Web-Browser) und Server (Web-Server)**
  1. Benutzer am **Client** klickt Link an oder schickt Formular ab
  2. **Client** stellt Anfrage (GET- oder POST-Request)
  3. **Server** antwortet mit HTML (Response mit HTML-Nutzlast)
  4. **Client** stellt HTML-Nutzlast dar → (1.)



# Client-Server-Informationsaustausch

- **Transaktionsmodus von Webseiten-Zugriffen:**

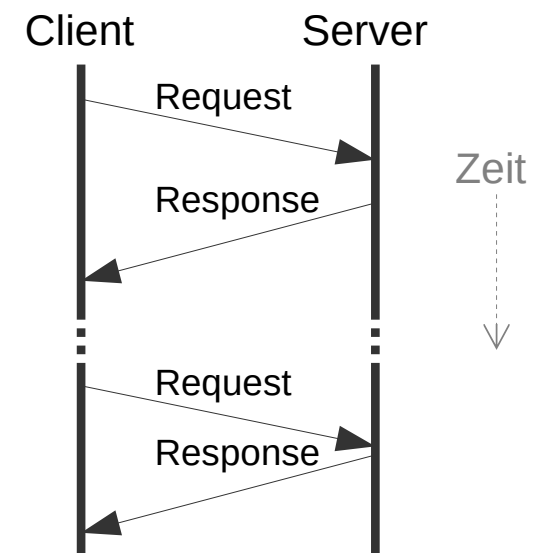
- **Request** vom Client an den Server (Anfrage)
- **Response** vom Server an den Client (Antwort)

- **Request**

- **Metadaten**
  - Was wollen wir? → URL
  - Wie wollen wir es? → Methode
  - **Request-Parameter** (z.B. Formulardaten)
- **Nutzlast** (evtl.)

- **Response**

- **Metadaten**
- **Nutzlast**
  - z.B. HTML-Seite
  - Im Body



# Client-Server-Informationsaustausch

---

- **Zur Erinnerung** (Kaptitel 1, HTTP):

```
Request = Request-Line
 *((general-header
 | request-header
 | entity-header) CRLF)
 CRLF
 [message-body]
```

```
Request-Line = Method SP
 Request-URI SP
 HTTP-Version CRLF
```

```
Request-URI = "*" | absoluteURI | abs_path | authority
```

```
Method = "GET" | "HEAD" | "POST"
```

- Bei Method POST werden im **message-body** (Nutzlast) Daten übertragen

# Requests auslösen

---

- **Requests** sind die Auslöser aller Aktivitäten

→ Fragen ...

- **Wie entstehen Requests?**
  - Was sind die Auslöser
  - Wer legt die Methode fest?
  - Wo kommen die Parameter her?
- **Wie werden Request-Parameter **kodiert**?**
  - Und warum muss ich das überhaupt wissen?

# Requests auslösen

- ... ausgelöst durch **Ereignis im** (oder beim) **Client**

1) Benutzer klickt einen **Link** an (**a**-Element mit **href**-Parameter)

- ```
<a href="/login.html">zum Login</a>
```

→ Erzeugt ein **GET-Request** mit der URL

2) Benutzer schickt **Formular** ab (**form**-Element mit **action**-Param.)

- ```
<form action="/login.html" method=get>
 Name: <input type=text name=name >

 Passwd: <input type=password name=password >

 <input type=submit name=login value="Login">
</form>
```

→ Erzeugt ein **GET-Request** (bei **method=get** )

→ ... oder ein **POST-Request** (bei **method=post** )

The diagram illustrates the relationship between the HTML form code and its visual representation. A yellow arrow points from the code block above to a visual form box. A white arrow points down from the top of the form box. The form box contains two input fields labeled 'Name:' and 'Passwd:', and a yellow 'Login' button.

3) Browser-extern ausgelöst

- Benutzer tippt URL in Browser-Adressleiste ein
- Externes Programm (z.B. Email-Client) übergibt URL an Browser



# Requests auslösen

---

- ... oder ausgelöst durch den vorherigen **Response**

- 1) Bei Empfang eines HTTP-Redirect-Headers

- **Location:** `http://myserver/mypath`
- Meist zusammen mit den Status-Codes
  - **301** Moved Permanently
  - **307** Temporary Redirect
- Die angegebene URL wird per GET abgerufen
  - *Interessant Rande:* Noch nicht lange ([RFC7231](#), Juni 2014) darf es sich beim Ziel um eine relative URI handeln. Als Fragment-Angabe („#...“) der resultierenden URL wird die angegebene oder ansonsten die der Basis-URL übernommen.

- 2) Ein meta-Element *refresh* im HTML-head-Element

- `<meta http-equiv=refresh content="5; URL=/login.html" >`
- *Hier wird die Webseite zunächst angezeigt und nach 5 Sekunden die angegebene URL per GET aufgerufen*
- *(Ohne URL-Angabe wird die selbe URL regelmäßig immer wieder geladen.)*

# Requests auslösen

- ... oder natürlich durch **Javascript**

Das ist einfach, Javascript automatisiert ja sozusagen den Client, man kann also u.a. Benutzeraktivitäten simulieren.

- 1) Einen GET-Request durch setzen der Dokument-URL auslösen

- ```
window.location = "http://myserver/mypath";
```

- 2) Eine Formular erzeugen und abschicken

- Idee: Man nutzt ein (in der HTML-Seite vorhandenes oder mit JS dynamisch aufgebautes) **Formular** und ruft die Methode **submit()** auf.

- ```
var form = document.querySelector('form#xyz');
form.submit();
```

**Vorhandenes** abschicken

- ```
var form = document.createElement('form');  
form.attr("method", "post");  
form.attr("action", "/login.html");  
var input = document.createElement('input');  
form.appendChild(input); // ggf. weitere Attribute für input  
form.submit();
```

Neu konstruiertes abschicken


- entsprechend ist GET oder POST möglich

GET- und POST-Parameter: query-strings

- Bei GET- und POST-Requests werden **Parameter** übermittelt

- z.B. die Inhalte von **Formularfeldern**:

- Name: Peter
- Password: 12345



A screenshot of a login form. It has a light gray background. On the left, there are labels 'Name:' and 'Passwd:'. To the right of 'Name:' is a text input field containing the text 'Peter'. To the right of 'Passwd:' is a text input field containing the text '12345'. Below these two input fields is a yellow button with the text 'Login' in black.

- **GET-Requests:**

- Parameter werden **in URL** einkodiert
- Format: **query-string**
 - **http://<host>/<path>?<query-string>**

- **POST-Requests:**

- Parameter werden **in message-body** abgelegt
- Format: **query-string**

GET- und POST-Parameter: query-strings

- Nochmal zur Erinnerung aus Kapitel 1: **URL-Syntax**

HTTP URL Scheme (1)

(gemäß [RFC 1738](#), „Uniform Resource Locators (URL)“)

http://<host>:**:**<port>**/**<path>**?**<searchpart>

- <searchpart> is a **query string**
- ein paar verbotene Zeichen
- keine weitere Struktur ...

- Hier wird keine Struktur für den *query string* festgelegt.

GET- und POST-Parameter: query-strings

- Was ist denn nun ein **Query-String**?

- Zunächst ist die Struktur im Standard nicht genauer spezifiziert
- Man könnte hier (fast) beliebige Strukturen nutzen
 - z.B. `Login`
- Man müsste dann nur dafür sorgen, dass mein Server das versteht.

Bitte nicht merken
(unrealistisches
Beispiel)!

- Wie nutzt man das in PHP (also auf Serverseite)?

- Die Variable `$_SERVER` enthält den kompletten Query-String:
 - `$_SERVER['QUERY_STRING'] == 'name*peter!password*12345'`
- Das müssten wir dann aber selber interpretieren und zerlegen
 - z.B. mit PHP-Funktionen `explode` oder `split` oder mit Regulären Ausdrücken
- Es gibt aber zum Glück einfachere Lösungen ...

GET- und POST-Parameter: query-strings

- Aber es gibt ja auch query-strings, die vom Browser automatisch erzeugt werden:

- **Formulare** erzeugen ihre query-strings automatisch, z.B.

```
http://myhost/login.html?name=Peter&password=12345
```

- Siehe HTML5-Standard 4.10.21.3 „**Form Submission Algorithm**“:
<https://html.spec.whatwg.org/multipage/form-control-infrastructure.html#...>
- Query-String-Encoding: <https://url.spec.whatwg.org/#concept-urlencoded-serializer>

- **Nutzen wir einfach ebenfalls im HTML-Quelltext**

- Dadurch muss der Server nur dieses eine Format unterstützen
- Wir erzeugen also auch mit allen anderen (nicht-Formular) Methoden query-strings nach diesem Muster
 - falls wir Parameter übergeben wollen
 - z.B. HTML-a-Element

```
<a href="/login.html?name=Peter&password=12345">zum Login</a>
```

- *Diskussion: In welchen Szenarien dieses a-Element mit Passwort real verwenden?*

GET- und POST-Parameter: query-strings

- **Beispiel (GET):**

```
<form action="/login.html" method=get>
  <input type=text      name=name      >
  <input type=password  name=password >
  <input type=submit    name=login     value="Login" >
</form>
```

Das 3. Input-Element ist „nur“ der Submit-Button, dennoch wird Sein value(Beschriftung) im Parameter übergeben.

- Eingabe in Formular „Peter“ für name und „12345“ für Passwort
- Erzeugte GET-URL:

```
http://.../login.html?name=Peter&password=12345&login=Login
```

- Es entsteht folgender Request (im Wesentlichen)

```
GET /login.html?name=Peter&password=12345&login=Login HTTP/1.1
Host: ...:80
```

GET- und POST-Parameter: query-strings

- **Beispiel (POST):**

```
<form action="/login.html" method=post>
  <input type=text      name=name      >
  <input type=password  name=password >
  <input type=submit    name=login    value="Login" >
</form>
```

- Eingabe in Formular „Peter“ für name und „12345“ für Passwort
- Erzeugte GET-URL:

```
http://.../login.html
```

- Es entsteht folgender **Request** (im Wesentlichen)

```
POST /login.html HTTP/1.1
Host: ...:80
Content-Type: application/x-www-form-urlencoded

name=Peter&password=12345&login=Login
```

} Header

} Leerzeile

} Payload

GET- und POST-Parameter: query-strings

- **Erzeugung (Format und Kodierung) von query-strings:**
 - Sequenz vom Name-Wert-Paaren der Form `<name> "=" <value>`
 - Jeweils getrennt durch ein `"&"`
 - ASCII-Sonderzeichen in Name und Wert werden als `%xx` kodiert
 - wobei die Hexadezimalzahl `xx` den ASCII-Code des Zeichens angibt.
 - Nicht kodiert werden müssen
Buchstaben `[A-Z, a-z]`, Ziffern `[0-9]` und `"-"`, `"_"`, `"."` und `"~"`
 - Leerzeichen können auch als `"+"` kodiert werden.
 - Das entstammt dem HTML-Standard,
 - entspricht nicht dem URI-Standard (ist aber unkritisch)
 - Um die Kodierung müssen wir uns immer selber kümmern, wenn wir die Query-Strings selber erzeugen
 - z.B. im Parameter `href` im `a`-Element eines von uns erzeugten HTML-Textes
 - In Formularen macht das der Browser für uns

GET- und POST-Parameter: query-strings

- **Wichtige-Codes (Auszug)**

SPACE	!	"	#	\$	%	&	'	()	
%20	%21	%22	%23	%24	%25	%26	%27	%28	%29	
*	+	,	/	:	;	=	?	@	[]
%2A	%2B	%2C	%2F	%3A	%3B	%3D	%3F	%40	%5B	%5D

- **Beispiele**

- Name „x“ mit Wert „2 * 3“ und „y“ mit „4“ ergibt kodiert:

`x=2%20%2A%203&y=4` *oder*

`x=2+%2A+3&y=4`

- Name „Price&Tax“ mit Wert „20\$ + 7.3%“ ergibt kodiert:

`Price%26Tax=20%24%20%2B%207.3%25` *oder*

`Price%26Tax=20%24+%2B+7.3%25`

Query-String-Encoding mit PHP

- **Zum Glück hat PHP dazu Funktionen**
 - `urlencode($str)` **kodiert** Parameter-Strings (→ php.net)
 - `urldecode($str)` **dekodiert** sie (→ php.net)
 - Beispiel:
 - `<?php echo urlencode('20$ + 7.3%') . "\n"; ?>`
 - Ergebnis: `20%24+%2B+7.3%25`
 - Wenn man es strikter haben will: `rawurlencode` und `rawurldecode`
 - Kodiert robuster (fast alle alphanumerischen Zeichen → php.net)
 - Beispiel:
 - `<?php echo rawurlencode('20$ + 7.3%') . "\n"; ?>`
 - Ergebnis: `20%24%20%2B%207.3%25`
 - Mit diesen Funktionen können wir also URL-Parameter kodieren
 - z.B. für ein a-Element

Query-String-Encoding mit PHP

- **Wo braucht man das?**

- Szenario: Wir haben (z.B. per Formular von Benutzer) eine Namens-Eingabe bekommen. Der Wert liegt in `$name`.
- Wir möchten jetzt die URL `/login.php` aufrufen und den Namen übergeben.

```
<?php
    $name = ... ; // stammt irgendwo her
    $url = '/login?name=' . urlencode($name);
    echo '<a href="' . $url . '>Login</a>';
?>
```

- Ergebnis ist im erzeugten HTML-Text ein A-Element, das bei Aufruf den Namen korrekt als GET-Parameter übergibt.
 - Enthält `$name` z.B. `"Firma Schmitt&Partner"`, so entsteht
`Login`
 - Ohne `urlencode` wäre ein Fehlerhafter Query-String entstanden:
`Login`

Verarbeitung von GET- und POST-Daten

- **Wie verarbeitet man die Query-String-Daten?**

- Zum Glück dekodiert PHP die per GET oder POST übergebenen Daten für uns. Sie liegen in

- `$_GET` alle per **GET** übertragenen Name-Wert-Paare
- `$_POST` alle per **POST** übertragenen Name-Wert-Paare
- `$_REQUEST` = `array_merge($_GET , $_POST)` // Vereinigung der Mengen

- Wenn gleichgültig ist, wie die Daten übertragen wurden, kann man `$_REQUEST` benutzen.

- Alle drei sind assoziative Arrays

- um z.B. auf den GET-Parameter „name“ zuzugreifen, dient der Ausdruck

- `$_GET['name']`

- Siehe <https://www.php.net/manual/de/language.types.array.php>

- Alle drei Variablen sind **superglobal**

- d.h. man von überall auf sie zugreifen (also ohne „`global $_GET;`“)

- Siehe <https://www.php.net/manual/de/language.variables.superglobals.php>

Verarbeitung von GET- und POST-Daten

- **Sicherheit** (aus Sicht des Servers)
 - Die in \$_GET und \$_POST enthaltenen Werte stammen **von außen** (vom Benutzer des Webdienstes)
 - Es können **Fehler** passieren oder Sie können sogar zu **Angriffen** dienen
 - Man kann ihren **Inhalten** allgemein **nicht vertrauen!**
 - Parameter **könnten ungesetzt** sein, obwohl wir sie gesetzt erwarten
 - Sie könnten Inhalte enthalten, die zu **unerwünschten Effekten** führen
- **Also ...**
 - Benutzergenerierte Daten müssen allgemein immer so behandelt werden, als würden sie von einem **Angreifer** stammen.
 - Ausnahme: Daten die von einem vertrauenswürdigen Nutzer (z.B. Admin).
 - Angriffe über benutzergenerierte Daten können **auch indirekt** erfolgen
 - Z.B. durch Daten, die zwischenzeitlich in der Datenbank abgelegt wurden.

Verarbeitung von GET- und POST-Daten

- **Wie prüfen, ob Parameter (nicht) gesetzt sind?**

- Wir erwarten, dass ein Parameter gesetzt wurde

- ... z.B. da wir die URL `/login.php` nur aus einem Formular aufrufen, in dem Name und Passwort abgefragt werden:

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

- In `/login.php` werten wir diese Daten aus:

```
<?php
  if ( $_POST['name']      == 'Tom'  &&
       $_POST['password'] == '1234' ) {
    $user = 'Tom'; // erfolgreich eingeloggt ...
  }
?>
```

- Der Benutzer hat aber die URL manuell eingegeben oder das Formular manipuliert. So sei z.B. `$_POST['password']` undefiniert.

- Der Zugriff auf `$_POST['password']` erzeugt eine Warnung

Verarbeitung von GET- und POST-Daten

- **Lösung 1:** Vorher prüfen

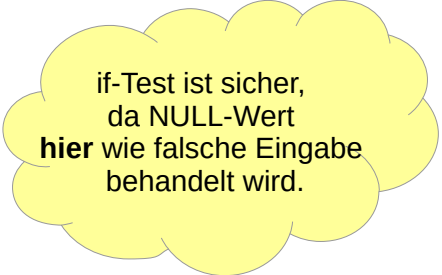
- mit `isset()` vorher prüfen

```
if (isset($_POST['name'], $_POST['password']) &&
    $_POST['name'] == 'Tom' &&
    $_POST['password'] == '1234' )
{
    $user = 'Tom'; // erfolgreich eingeloggt ...
}
```

- **Lösung 2:** Sicherer Default-Wert

- Ist der Wert nicht gesetzt ist das Ergebnis NULL → sicher?
- Optional: `@`-Präfix → Keine Warnung wenn nicht gesetzt

- ```
if (@$_POST['name'] == 'Tom' &&
 @$_POST['password'] == '1234')
{
 $user = 'Tom'; // erfolgreich eingeloggt ...
}
```



if-Test ist sicher,  
da NULL-Wert  
hier wie falsche Eingabe  
behandelt wird.



# Schutz vor gefährlichen Inhalten

---

- **Wie mit gefährlichen Inhalten von Daten umgehen?**
  - z.B. werden übergebene Parameter oft in Ausgaben verwendet ...
    - Als Parameter für URLs
      - Zum Schutz gegen unerwünschte Effekte kennen wir ja schon `urlencode` (s.o.)
    - Als Ausgabe im HTML-Text
    - Als Teil einer Datenbank-Anfrage („SQL-Injection“ → später)
  - Die Idee beim Angriff ist immer, einen Parameter zu übergeben, der beim Einbau in eine Ausgabe zu unerwünschten Effekten führt.
  - Beispiel:
    - Unser `/login.php` soll bei einem erfolglosen Login-Versuch schon einmal den Benutzernamen wieder ins Formular übernehmen:

```
<form action="/login.php" method=post>
 <input type=text name=
 value="<?php echo @$_POST['name']; ?>" >
 <!-- -->
</form>
```

# Schutz vor gefährlichen Inhalten

- **Unser (böser?) Nutzer versucht sich einzuloggen ...**

- 1. Versuch, Eintippen der URL (keine POST-Daten)

- `@$_POST['name']` liefert **NULL**, also konvertiert zu String den **leeren String**

```
<input type=text name=name value="" >
```

- 2. Versuch: Name „Peter“, (falsches) Password „4321“

- Wir liefern das Login-Formular mit `value="Peter"` zurück

```
<input type=text name=name value="Peter" >
```

- 3. Versuch: Name „`></html>`“

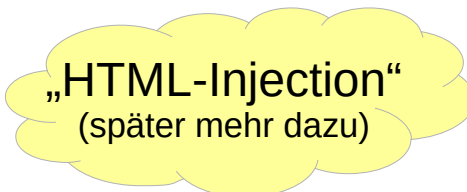
- Wir bauen das unbesehen in das Formular ein ...

```
<input type=text name=name value="" ></html>" >
```

- Das wollten wir sicher nicht!!!

- Derartiges passiert überall, wo *unsichere Inhalte* in HTML eingebaut werden

- `<p>Hallo <?php echo $name; ?>, dein Passwort war nicht richtig.</p>`



„HTML-Injection“  
(später mehr dazu)

# Schutz vor gefährlichen Inhalten

---

- **Wir wollen alle für uns gefährlichen Zeichen los werden ...**
  - „<“ und „>“, um zu verhindern dass Tags erzeugt werden (s.o.)
  - **Anführungszeichen**, um zu verhindern dass man HTML-Tag-Parameter-Strings vorzeitig beendet (s.o.)
    - u.U. genügt es, nur Doppelte „“ zu entfernen, wenn man in für Tag-Parameter-Strings nur solche benutzt. Einfache ‚‘ können dann bleiben.
  - **&-Zeichen**, um zu verhindern dass diese als HTML-Zeichencode interpretiert werden.
    - z.B. in der Eingabe „ich messe volt&“, die in HTML anders interpretiert würde als vom Formular-Benutzer erwartet.
- **Genau das tun `htmlspecialchars()` bzw. `htmlentities()`**
  - „&“ (Ampersand/kaufmännisches UND) wird zu `'&#38;'`.
  - „“ (doppeltes Anführungszeichen) wird zu `'&#34;'`
  - „<“ (kleiner als) wird zu `'&#60;'`
  - „>“ (größer als) wird zu `'&#62;'`

# Schutz vor gefährlichen Inhalten

---

- Wirkung von **htmlspecialchars()**

- Wandelt die o.g. HTML-Sonderzeichen um in HTML-Codes

- `<?php`

```
$dangerous = "Test";
$safe = htmlspecialchars($dangerous);
echo $safe;
```

`?>`

Ergebnis: `&lt;a href='test'&gt;Test&lt;/a&gt;`

- Bei Aufruf `htmlspecialchars($dangerous, ENT_QUOTES)` werden **auch einfache** Anführungszeichen umgewandelt

- Ergebnis: `&lt;a href='&#039;test&#039;&gt;Test&lt;/a&gt;`

- Diverse weitere Optionen (siehe php.net)

- Wirkung von **htmlentities()**

- Wandelt darüber hinaus z.B. Umlaute (,ä' → ,&auml;') um.

- Vorteilhaft bzgl. robustem Encoding – aber kein echter Sicherheitsgewinn
- Achtung: Auch hier ist ggf. `htmlentities(..., ENT_QUOTES)` nötig (s.o.)

# Schutz vor gefährlichen Inhalten

- **Anwendung**

- Diese sichere Kodierung sollte **immer** angewandt werden, wenn **potentiell gefährliche Daten** in HTML-Seiten eingebaut werden.

- **Probleme**

- **Mehrfachanwendung** führt zu unerwünschten Effekten: Die HTML-Codes werden in der Webseite nur einmal dekodiert.

- ```
<?php
    $dangerous = "&";
    $safe      = htmlspecialchars($dangerous);
    $doublesafe= htmlspecialchars($safe);
    echo $dangerous . "\n" . $safe . "\n" . $doublesafe;
?>
```

- | | | | |
|-----------|-----------|------------------|--------------------------|
| Ergebnis: | & | Browser-Anzeige: | <input type="checkbox"/> |
| | & | | & |
| | &amp; | | & |

Fehler (-Toleranz)

Schutz vor gefährlichen Inhalten

- **Man kann auch die HTML-Tag-Typen beschränken**

- `strip_tags($str [, $allowable_tags])`

- Entfernt alle Tags, außer sie sind explizit erlaubt

- `<?php`

```
$text = 'Das <b>"A&O"</b> ist <i>alles</i>.';
echo strip_tags($text) . "\n";
echo strip_tags($text, '<b><p>') . "\n";
?>
```

- Ergebnis: `Das "A&O" ist alles.`

- `Das "A&O" ist alles.`

- Nützlich, wenn man z.B. in Blog-Beiträgen Fettschreibung (also b-Tags) erlauben will, aber nicht z.B. Bilder (img-Elemente) oder Links (a-Elemente).

- **Achtung:** Das ist nicht in allen Situationen ausreichend um Stabilität und Sicherheit zu garantieren

(Debug-) Ausgabe von PHP-Variablen

- **print und echo**

- ... geben einen (**print** x) oder mehrere (**echo** x, y, z) Strings aus.
→ ggf. Stringkonvertierung → strukturierte Daten werden nicht ausgegeben

```
$a = array('rot', 3=>'grün', 'b'=>'blau');  
print $a;
```

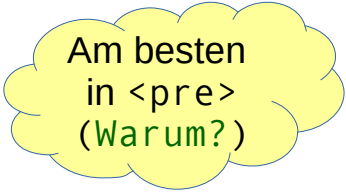
```
Array
```

- **print_r** (→ php.net)

- ... gibt strukturierte Daten **rekursiv** aus

```
print_r($a);
```

```
Array  
(  
    [0] => rot  
    [3] => grün  
    [b] => blau  
)
```



Am besten
in <pre>
(Warum?)

- Optional auch zum Abspeichern in eine Variable

```
$a_as_text = print_r($a, true); // keine Ausgabe
```

(Debug-) Ausgabe von PHP-Variablen

- **var_dump** (→ php.net)

- Gibt strukturierte Daten **rekursiv** und mit **Typangaben** aus

```
$a = array('rot', 3=>'grün', 'b'=>'blau');  
var_dump($a);
```

```
array(3) {  
  [0]=>  
    string(3) "rot"  
  [3]=>  
    string(5) "grün"  
  ["b"]=>  
    string(4) "blau"  
}
```

- Abfangen der Ausgabe von `var_dump` als Wert

- **Trick:** Ausgabe abfangen mit `ob_start` + `ob_get_clean` (→ php.net)

```
$a = array('rot', 3=>'grün', 'b'=>'blau');  
  
ob_start();  
var_dump($a);  
$a_as_text = ob_get_clean(); } // Ausgabe der 3  
// Zeilen abfangen  
// und zurück liefern
```


(Debug-) Ausgabe von PHP-Variablen

- **print_r** oder **var_dump** in HTML-Ausgaben benutzen
 - Wunsch:
 - Ausgabestruktur zu erhalten (Einrückung, Umbruch)
 - Sichere Ausgabe (keine Interpretation von HTML-Steuerzeichen)
 - Lösung:

```
<pre>
  <?php
    $a = array('rot', 3=>'grün', 'b'=>'<b>bold</b>');
    echo htmlspecialchars(print_r($a, true));
  ?>
</pre>
```

```
<pre>
  Array
  (
    [0] => rot
    [3] => grün
    [b] => &lt;b&gt;bold&lt;/b&gt;
  )
</pre>
```

Ausgabe aller PHP-System-Variablen

- Funktion **phpinfo()**

- Gibt diverse Systemvariablen von PHP in Tabellenform aus
 - → php.net
- Ausgabe ist sicher bzgl. Injections
- Vorsicht aber vor öffentlicher Ausgabe
 - da viele Informationen Angreifern helfen können



| | |
|--|---|
| System | Linux scilab-0100 5.15.152-1-pve #1 SMP PVE 5.15.152-1 (2024-04-29T07:31Z) x86_64 |
| Build Date | Feb 23 2023 12:43:23 |
| Server API | Apache 2.0 Handler |
| Virtual Directory Support | disabled |
| Configuration File (php.ini) Path | /etc/php/7.4/apache2 |
| Loaded Configuration File | /etc/php/7.4/apache2/php.ini |

.....

- **Demo:** 5... in http://scilab-0100.cs.uni-kl.de/1_basics/

Sichere Ausgabe von PHP-System-Variablen

- **Beispiel:** Wir wollen zum Testen Variablen anzeigen
 - Sie sollen auf der Webseite sicher angezeigt werden
 - Beispiel: `$_GET`
 - ```
<?php
 echo "<h2>_GET</h2>\n";
 $printed = print_r($_GET, true);
 echo '<pre>' . htmlspecialchars($printed) . '</pre>';
?>
```
    - Wir benutzen `print_r($var, true)` um die **Ausgabe als String** zu erzeugen
    - Wir benutzen `htmlspecialchars()` um Sonderzeichen abzufangen (s.o.)
    - Analog: `$_POST`, `$_SERVER`, `$_COOKIES`, ... (s.u.)

# Sichere Ausgabe von PHP-System-Variablen

- Verbesserung: **Test-Ausgabe der Variablen in Schleife**

- Diverse Variablen sollen sicher angezeigt werden

- ```
<?php
    foreach (array('_GET', '_POST', '_COOKIE',
                  '_SESSION', '_SERVER') as $varname) {
        if (isset($$varname)) {
            echo "<h2>" . $varname . "</h2>\n";
            $printed = print_r($$varname, true);
            echo '<pre>' . htmlspecialchars($printed) . '</pre>';
        }
    }
?>
```

- Mit `$$varname` bekommen wir den Inhalt der Variablen mit Namen `$varname`

- *Wenn einige Teile nicht erscheinen ist vielleicht folgender Workaround nötig:*

- Vor die foreach-Schleife die folgende Zeile einfügen:

- `@$_SERVER; @$_REQUEST; // Workaround, macht ggf. Variablen sichtbar`

- **Demo:** 6... + 7... in http://scilab-0100.cs.uni-kl.de/1_basics/

Applikationsstruktur

- **Struktur einer PHP-Applikation**

- Eine HTML- oder PHP-Seite anlegen, die
 - Inhalte und Daten ausgibt
 - **Requests** (ggf. mit Parametern) **erzeugt**
 - Links auf andere URLs (GET-Requests)
 - **Formulare** (GET- oder POST-Requests)
- Eine PHP-Seite anlegen, die solche **Requests verarbeitet**
 - Prüft ob Daten übergeben wurden
 - Daten sicher verwendet,
 - z.B. in eine Webseite ausgibt, ohne dass sie Schaden verursachen können

- **Günstige Applikationsstruktur**

- Ziel: Lösungen **kompakt** und **übersichtlich** realisieren
- Jeweilige Funktionen als separate PHP-Dateien realisieren
- Idee: Formular und verarbeitenden Code bündeln (**Postback**)

Applikationsstruktur

- **Beispiel: Login-Seite**

- Funktion:

- Bei erstem Aufruf: Login-Formular (POST) anbieten
- Nach Abschicken des Formulars:
 - Einloggen (wenn Daten korrekt) *oder*
 - erneut Formular anbieten (wenn Daten nicht korrekt)

- Grober Ablauf:

- Aufruf **mit** POST-Daten: Login-Daten prüfen
 - Wenn erfolgreich: \$user setzen
 - Wenn nicht erfolgreich: Fehler anzeigen
- Aufruf **ohne** POST-Daten (*oder* Login-Versuch nicht erfolgreich)
 - Login-Formular ausgeben (ggf. vor-ausgefüllt)

- Beides wird **von der selben PHP-Seite** realisiert

- **Postback**, das POST des Formulars geht also an die selbe URL
 - im Form-Tag gilt `action="" method="post"`

Beispiel: Login-Seite

- Prüfen, ob Login erfolgreich

- ```
<?php
 $user_id = NULL;
 if (@$_POST['name'] == 'Tom' &&
 @$_POST['password'] == '1234') {
 $user_id = 'Tom'; // erfolgreich eingeloggt ...
 }
?>
```

- Login-Formular ausgeben, wenn kein Login erfolgt ist

```
<?php if (!$user_id) { ?>
 <h1>Login</h1>
 <form action="" method=post>
 Name: <input type=text name=name >

 Passwd.: <input type=password name=password >

 <input type=submit name=login value="Login" >
 </form>
<?php } ?>
```

HTML,  
wird nur  
ausgegeben  
wenn  
if-Bedingung  
erfüllt.

- Willkommensmeldung ausgeben, wenn Login erfolgreich war

```
<?php if ($user_id) { ?>
 Willkommen, <?php echo htmlspecialchars($user_id); ?>!
<?php } ?>
```

# Beispiel: Login-Seite

- **Verbesserung: Login flexibler realisieren**

- Login-Test flexibler und in separate Funktion

- ```
<?php
    function get_userdata($id) {
        $user_list = array(
            'Tom' => array('password' => '1234', 'name' => 'Tom Jones'),
            'Peter' => array('password' => '2345', 'name' => 'Peter Deng'),
            'John' => array('password' => '3456', 'name' => 'John Doe'),
        );
        return @$user_list[$id]; // liefert NULL bei unbekanntem User
    }

    function get_login($id, $password) {
        $u = get_userdata($id);
        if ($u && $u['password'] == $password )
            return $u; // erfolgreich eingeloggt
        return NULL; // nicht erfolgreich eingeloggt
    }
?>
```

- Prüfen, ob Login erfolgreich

- ```
<?php
 $user_data = get_login(@$_POST['name'], @$_POST['password']);
?>
```



# Beispiel: Login-Seite

- **Verbesserung: Login separat, setzt selbst Variablen**

- Login-Test in separater Include-Datei, setzt globale Variablen

In externe-Datei 'inc\_\_get\_login.php' auslagern

```
• <?php // This is the Include-File 'inc__get_login.php'
 function get_userdata($id) {
 // ... wie bisher ...
 }

 function get_login($id = NULL, $password = NULL) {
 global $user_data, $user_id;
 $user_data = $user_id = NULL; // Fallback-Werte
 $u = get_userdata($id);
 if ($u && @$u['password'] == $password)
 $user_data = $u;
 $user_id = $id;
 }
 }
?>
```

- Prüfen, ob Login erfolgreich

- <?php **include** 'inc\_\_get\_login.php';  
    **get\_login**(@\$\_POST['name'], @\$\_POST['password']); ?>
- Die Variablen `$user_data` und `$user_id` werden dadurch gesetzt

Include bindet externe Dateien ein (→ [php.net](http://php.net))

# Beispiel: Login-Seite

---

- **Verbesserung: Benutzer-Daten verwenden**

- Willkommensmeldung mit dem Klartext-Namen ausgeben

```
<?php if ($user_data) { ?>
 Willkommen
 <?php echo htmlspecialchars(@$user_data['name']); ?>
 !
<?php } ?>
```

- Besser: Von separater Login-Seite bei Erfolg **weiterleiten** auf Inhaltsseite für Nutzer (z.B. auf persönliche „Homepage“)

- ```
<?php
    if ($user_data) {
        header("Location: /user_home.php", TRUE, 307);
        // Setzt Location Header
        // Setzt Status-Code 307: Temporary Redirect
    }
?>
```

- `header()` muss vor allen Ausgaben aufgerufen werden (→ php.net)

Beispiel: Login-Seite

- **Verbesserung: Passwörter *gehasht* speichern (1)**

- Dadurch kann ein Angreifer die originalen Passwörter nicht erhalten, wenn er an die Benutzer-Datenbank bekommt.

- **Hash-Funktionen**

- Berechnen zu einem beliebigen String einen **Hash** (ein String fester Länge)
- Ein Hash ist eine Art **Prüfsumme**
- Kleine Änderungen an der Eingabe erzeugen große Änderungen am Hash

- Beispiele für verbreitete Hash-Funktionen: [md5](#), [sha1](#), [sha256](#)

- In php berechnen **md5(\$x)** und **sha1(\$x)** Hashes zu Strings

```
<?php
    foreach (array('1234', '1235') as $x)
        echo "md5('$x') = " . md5($x) . "\n";
?>
```

- Ergebnis:

```
md5('1234') = '81dc9bdb52d04dc20036dbd8313ed055'
```

```
md5('1235') = '9996535e07258a7bbfd8b132435c5962'
```

Beispiel: Login-Seite

- **Verbesserung: Passwörter *gehasht* speichern (2)**

- Wir legen die gespeicherten Passwörter im Server nur *gehasht* ab

```
function get_userdata($id) {
    $user_list = array(
        'Tom'    => array('pw_md5' => 'e7df7cd2ca07f4f1ab415d457a6e1c13',
                        'name'    => 'Tom Jones'),
        'Peter' => array('pw_md5' => 'c47abe049e90cd2d285fd697ca4a8c6a',
                        'name'    => 'Peter Deng'),
    );
    return @$user_list[$id];
}
```

- Beim Prüfen des übergebenen Passworts hashen wir dieses und vergleichen

```
function get_login($id, $password) {
    $u = get_userdata($id);
    if ($u && @$u['pw_md5'] == md5($password) )
        // ... erfolgreich eingeloggt
    // ... sonst nicht erfolgreich eingeloggt
}
```

- Sind die Hashes gleich, sind auch die Passwörter gleich.
- Der Server kennt (außer während des Prüfens) nur Passwörter-Hashes.
- *Am Rande:* Idealerweise sollte man den Hash noch mit einem *Salt* versehen.

Beispiel: Login-Seite

• Hintergrund: Kryptographische Hash-Funktionen (1)

Wozu?

1) Anforderung: Falltüreigenschaft

Vorsicht: Für eine **Teilmenge** der Eingaben könnte man aber die Hashwerte voraus berechnen und wiedererkennen (Rainbow-Tables).

aus dem Hashwert kann die Eingabe nicht (effizient) berechnet werden

- **Einsatz-Beispiel:** Ein Angreifer kann aus dem obigen gehashten Passwort nicht (effektiv) das originale Passwort bestimmen.

Wozu?

2) Anforderung: Hash-Kollisionen (praktisch) nicht zu finden

- Zu einer Hashfunktion haben alle Hash-Werte eine feste Länge
- Die Eingabe kann eine beliebige Länge haben
 - Es gibt mehr Eingabe-Werte als Hash-Werte
 - Es gibt daher mehrere Eingaben, die den selben Hash haben (Hash-Kollision)
- **Aber:** Hash-Kollisionen können nicht (effizient) gefunden werden
 - Zu einer Eingabe kann man nicht (effektiv) andere mit gleichem Hashwert finden.
 - **Einsatz-Beispiel:** Wenn der Hash eines Strings unverändert ist, dann ist auch der String (höchstwahrscheinlich) unverändert.
- Und: Hash-Kollisionen sind allgemein astronomisch selten
 - Bei einer **sicheren** Hash-Funktion sind sie praktisch auch nicht gezielt zu finden

MD5 und SHA-1 gelten insbes. bzgl. provozierten Kollisionen nicht mehr für alle Anwendungen als ausreichend sicher.

Beispiel: Login-Seite

- **Hintergrund: Kryptographische Hash-Funktionen (2)**

- MD5 und SHA1 erfüllen mittlerweile die o.g. Anforderungen nicht mehr ausreichend sicher.
 - Es sind u.a. Verfahren entdeckt worden, mit denen **Hash-Kollisionen** unter bestimmten Bedingungen gezielt gefunden werden können.
 - Sie sollten daher für kritische Anwendungen nicht mehr eingesetzt werden.

- **Bessere Alternativen: z.B. SHA256**

```
<?php
    foreach (array( '1234', '1235' ) as $x)
        echo "sha265('$x') = " . hash('sha256', $x) . "\n";
?>
```

Ergebnis:

```
sha265('1234') = '03ac674216f3e15c761ee1a5e255f067953623c8b388b4459e13f978d7c846f4'
sha265('1235') = '310ced37200b1a0dae25edb263fe52c491f6e467268acab0ffec06666e2ed959'
```

- Die PHP-**hash**-Funktion unterstützt noch weitere Hashes
 - Siehe <https://www.php.net/manual/de/function.hash.php>

Beispiel: Login-Seite

- **Hintergrund: Kryptographische Hash-Funktionen (3)**

- Alle (effizient) umkehrbaren Funktionen sind **keine** Hash-Funktionen

- Beispiel: **base64-Kodierung** ist keine Hash-Funktion

- `<?php`

```
$s = '1234';
```

```
$a = base64_encode($s);
```

```
echo "base64_encode('$s') = '$a'\n";
```

```
unset($s); // original-String gelöscht
```

```
$b = base64_decode($a);
```

```
echo "base64_decode('$a') = '$b'\n";
```

```
?>
```

- Ergebnis:

```
base64_encode('1234') = 'MTIzNA=='
```

```
base64_decode('MTIzNA==') = '1234'
```

Zudem: lokale Änderung
→ lokaler Effekt:

'1234' → 'MTIzNA=='

'1235' → 'MTIzNQ=='

- Zur Erinnerung (Kapitel 1):

- Die Base64-Kodierung wurde zur Verschlüsselung des Passworts in der **Basic-Authentication** eingesetzt

- Diese kann aber leicht dekodiert werden → Schein-Sicherheit („Snakeoil“)

Beispiel: Login-Seite

- **Ausblick: Sichere Passwort-Hash-Verfahren**

- Problem: Passwörter oft **kombinatorisch schwach**
 - Vom Benutzer gewählte Passwörter sind oft **relativ kurz** (6-10 Zeichen)
 - Es werden oft nur wenige Zeichen genutzt (nur Kleinbuchstaben, nur Ziffern)
 - 10^6 (6 Ziffern) ... $26^{10} \approx 1,4 \cdot 10^{14}$ (10 Kleinbuchstaben) Varianten
- Angriff: Man berechnet alle Hashes voraus („**Rainbow-Table**“)
 - Eine Festplatte mit 8TB ($8 \cdot 10^{12}$ Zeichen) kostet ca. 100€
 - Für einige 10.000€ kann man alle Hashes für erwartete Passwörter speichern
- Lösung: **Salt** (oder **Pepper**)

- Das Passwort wird um einen Zufalls-Zeichenkette **erweitert** (**Salt**)
- Der Salt wird zusammen mit dem Passworthash abgelegt

$\$salt$ $hash(\$salt . \$passwd)$

`$a34df2d$81dc9bdb52d04dc22036dbd8313ed055`

Und/oder **Pepper**:
Vom Server **fest** gewählte
(geheime) Erweiterung
hinzufügen

- Ziel: Rainbow-Tables werden zu groß → nicht realisierbar (exponentiell!)
- Siehe [https://de.wikipedia.org/wiki/Salt_\(Kryptologie\)](https://de.wikipedia.org/wiki/Salt_(Kryptologie))

Beispiel: Login-Seite

- **Ausblick: Sichere Passwort-Hash-Verfahren**

- Problem: Passwörter haben oft **geringe Entropie** oder sind **errätbar**
 - Entropie ist ein Maß, wie ungeordnet ein System ist
 - Konkret: Bestimmte Passwörter sind beliebt
 - Angreifer sammeln „beliebte“ **Passwörter** und probieren diese (**Wörterbuch**)
 - Aber auch: Passwörter sind nicht völlig gleichverteilt
 - Worte, Silben oder Zifferngruppen kommen oft vor
 - Häufig Muster in Passwörtern (z.B. „xyz123#“)
 - Erfolgswahrscheinlichkeit bei **geschicktem Ausprobieren** besser als erwartet
- Angriffsform: **Wörterbuchattacke**, geschickte **Brute-Force-Attacke**
- Lösung: Das **Ausprobieren** eines Passworts „teuer“ machen
 - Hash-Funktion mehrmals (z.B. 10.000 **Runden**) hintereinander anwenden
 - Dadurch kostet ein Passwort-Test z.B. eine Sekunde → Massentests unmöglich
 - Das erschwert auch den Aufbau einer Rainbow-Table extrem
 - Beispiel: **Bcrypt** (<https://de.wikipedia.org/wiki/Bcrypt>)
 - Salting, parametrierbarer Rechenaufwand, nicht ASIC-/SIMD-optimierbar

Beispiel: Login-Seite

- **Ausblick: Sichere Passwort-Hash-Verfahren**

- Passwort-Hashing ist ein komplexes Thema

- Man kann leicht Fehler bzgl. Verwundbarkeiten verursachen

- Besser eine dafür vorgesehene und überprüfte Lösung nutzen

- Erweiterte Passwort-Hashing-Funktionen von PHP

- <https://www.php.net/manual/de/book.password.php>

- `$hash = password_hash($password, $algo [, $options])`

- `$algo` z.B. `PASSWORD_BCRYPT` oder `PASSWORD_ARGON2I`

- Liefert so etwas wie

- `$argon2i$v=19$m=1024,t=2,p=2$YzJBMzc3d3laeg$zqUsdWLaw3sYY2i2jT0 ...`

- Incl. `Algorithmus`, Anzahl Runden, Parameter, `Salt`, Hash

- `$ok = password_verify($password, $hash)`

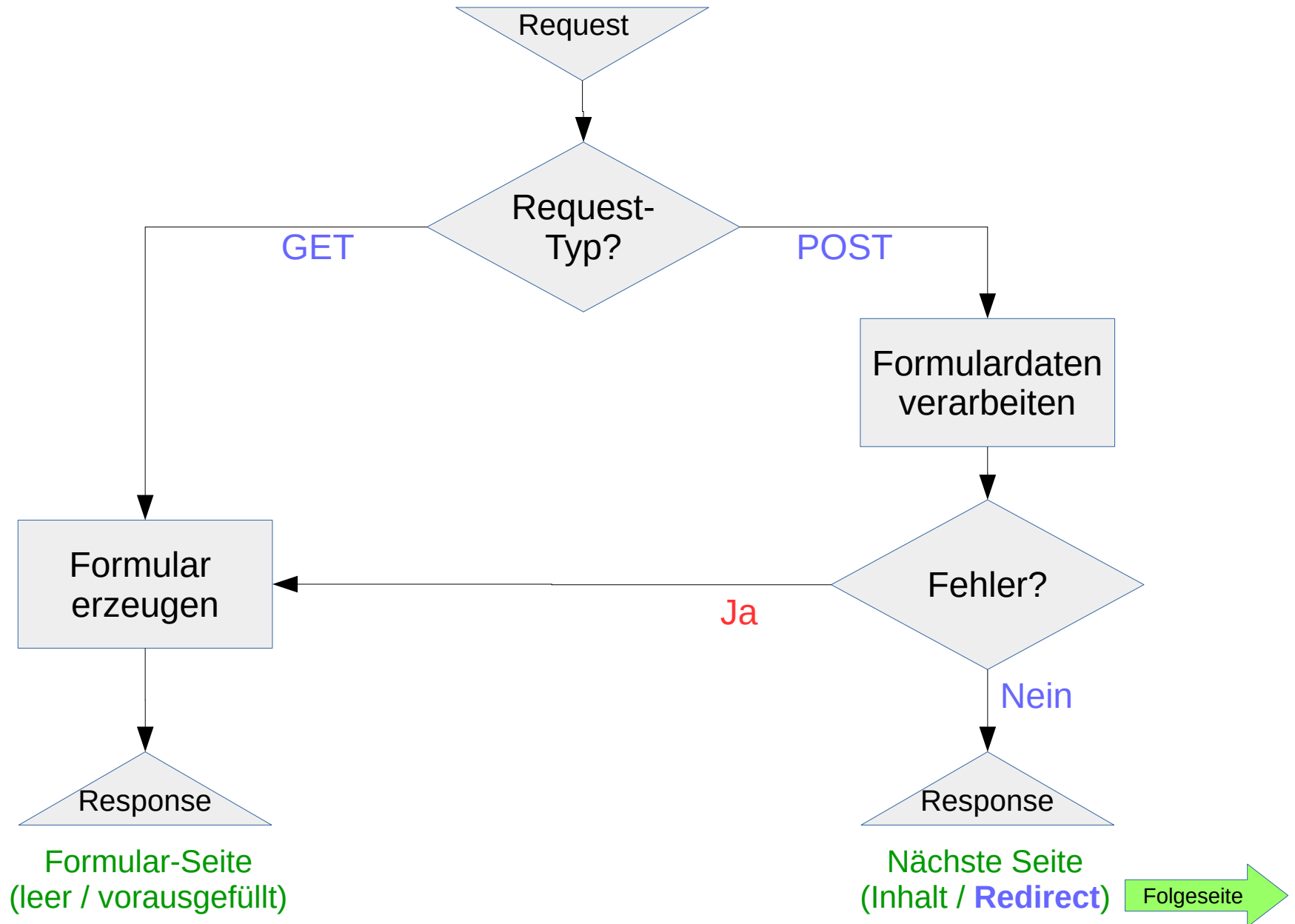
- Siehe

- <https://www.php.net/manual/de/faq.passwords.php>

Postback

- **Ein und die selbe PHP-Seite ...**
 - liefert das Formular und
 - verarbeitet dessen Eingaben
- **Vorteil:**
 - Kompakt, alle Formularaspekte in einer Datei behandelt
- **Schema:**
 - 1) GET (ersten Aufruf): Webseite liefert leeres Formular
 - 2) POST: Webseite wertet ausgefülltes Formular aus
 - Wenn fehlerfrei: **Positive Antwort** erzeugen
 - Wenn fehlerhaft: Auf (1) zurückfallen, ggf. mit Fehlermeldung

Postback



Postback

- **Positive Antwort (nach POST)**

- Unmittelbar Webseite mit Inhalt für den User liefern
- oder **HTTP-Redirect** auf Zielseite (z.B. Homepage nach Login)
 - Zur Erinnerung: **HTTP-Redirect**
 - Status-Code 307 (Temporary Redirect)
 - Location-Response-Header `Location = "Location" ":" absoluteURI`
 - Setzen mit PHP-Funktion `header()`
 - ```
<?php
 if ($user_data) {
 header("Location: /user_home.php", TRUE, 307);
 // Setzt Location Header
 // Setzt Status-Code 307: Temporary Redirect
 }
?>
```
- Frage: Was passiert in beiden Fällen wenn Nutzer **Reload** drückt?

# Design von GET- und POST-Requests

---

## Fragen:

- Wann GET, wann POST?
  - Warum?
- Wie werden Parameter übertragen?
  - Zur Erinnerung: Der Webserver selbst ist zustandslos!
  - Woher weiß er dann, alles, was er über die Anfrage wissen muss?

# Design von GET- und POST-Requests

---

- **Typische GET-Requests**

- *„Zeige eine Liste aller Lehrveranstaltungen“*
  - Parameter: In welchem Semester? Zu welchem Studiengang?
- *„Zeige meine neuen Emails“*
  - Parameter: Zu welchem Benutzer?
- *„Zeige den Inhalt des Warenkorb“*
  - Parameter: Zu welchem Kunden?
  - Wenn Kunde noch nicht bekannt (eingeloggt / registriert): Zu welchem (noch anonymen) Warenkorb?

- **Typische POST-Requests**

- *„Melde mich zu der Prüfung an!“*
- *„Lösche die markierte Email (endgültig)!“*
- *„Bestelle den Inhalt des Warenkorb verbindlich!“*

# Design von GET- und POST-Requests

---

## Regel:

- **GET-Requests** für **nicht-ändernde** (wiederholbare) Anfragen
  - Requests können problemlos **wiederholt** werden
    - (Anzahl der durchgeführten Requests)  $\geq$  (Anzahl der nachgefragten)
  - Antworten können oft auch aus (Client- / Server-) **Cache** geliefert werden
    - (Anzahl der durchgeführten Requests)  $\leq$  (Anzahl der nachgefragten)
      - *Hier spielt es bzgl. Caching eher eine Rolle, ob die Antwort noch aktuell ist.*
- **POST-Requests** für **ändernde** Anfragen
  - Requests sollten im Zweifelsfall **nicht wiederholt** werden
    - Sonst hat man vielleicht ungewollt zwei Waschmaschinen gekauft
    - Clients fragen deshalb bei **Reload** (meist F5-Taste) beim Benutzer nach
  - Request kann meist **nicht aus Client-Cache** bedient werden
    - Sonst kommt die (gewollte) ändernde Anfrage nicht beim Server an
      - *Und man bekommt den zweiten Drink nicht trotz erneuter Bestellung ...*



# Parameter-Austausch

---

- **Übergabe von Parametern** (Client ↔ Server)
  - Als GET- oder POST-Parameter
    - z.B. `http://www.google.de/search?q=HTML+5`
    - In PHP dann in `$_GET`, `$_POST`, `$_REQUEST` (hier z.B. in `$_GET['q']`)
  - Als Cookie-Parameter (*TODO*)
    - In PHP dann in `$_COOKIE`
  - Als URL-Komponente
    - z.B. `https://vlu.informatik.uni-kl.de/auswertung/11/227/`
      - Die beiden Zahlenwerte werden hier als **Parameter** (z.B. Datenbank-ID) benutzt
      - Sie geben typischerweise **nicht** wie gewohnt einen Dateisystem-Pfad an, in dem z.B. ein PHP-Script oder eine fertige HTML-Datei liegt
      - Es wirkt aber so ... und soll es auch!
    - Idee: **Semantic URL**
  - *Ausblick*: Als frei definierter **X-Header** (mit Javascript im Client)

# Parameter-Austausch

---

- **Semantic URLs** („User-Friendly URLs“, „Search Engine-Friendly URLs“)
  - **Grundidee:** Die URL erklärt sich selbst
    - Pfadstruktur, die den Inhalt **hierarchisch** und **semantisch** beschreibt
    - Sie enthält keine GET-Parameter
      - Alles sieht so aus, als ob die Webseiten gar nicht erzeugt würden, sondern als ob sie schon fertig als statische Dateien in einer Verzeichnis-Hierarchie auf dem Server liegen würden
  - **Beispiel:**
    - Z.B. die URL einer Wikipedia-Seite zum Thema „Semantic URL“  
[http://en.wikipedia.org/wiki/Semantic\\_URL](http://en.wikipedia.org/wiki/Semantic_URL)
  - **Gegenbeispiel:**
    - `http://www.kis.uni-kl.de/campus/all/event.asp?...`  
`gguid=0xED9E620955AF4E36AB19239CA3CFB41E&...`  
`tguid=0xFABFCA6FD5CF47949C85F333100B85D4`
      - Das ist „*offensichtlich*“ die KIS-Seite dieser Vorlesung (INF-00-32-V-3) im SS 2023
      - **Übungsfrage:** Wie würde die **Semantic-URL** idealerweise aussehen?

# Parameter-Austausch

---

- **URL-Zerlegung** – Realisierung in Apache + PHP

- In **Apache** (*leider etwas technisch – nur Grundidee relevant*)

- Ziel: Wir wollen alle URL-Zugriffe unterhalb eines Pfades (z.B. `/blog/`) auf das selbe PHP-Script lenken

- Also Zugriff auf `http://myserver/blog/2024/05/01/` führt zu `/blog/index.php`

- Dazu in Apache z.B. die Option **FallbackResource** benutzen

- In der Apache-Konfiguration:

```
<Directory "/htdocs/blog">
 FallbackResource /blog/index.php
</Directory>
```

- Oder in der Datei `/htdocs/blog/.htaccess` die Zeile

```
FallbackResource /index.php
```

- Alternative: Rewrite-Regeln (Apache-Modul „mod\_rewrite“)

- Danach wird bei Zugriffen unterhalb von `/blog/` immer auf das PHP-Script `/blog/index.php` zugegriffen

z.B. `http://myserver/blog/2024/05/01/`  
    `≈ http://myserver/blog/index.php`  
    `≈ http://myserver/blog/`

# Parameter-Austausch

---

- **URL-Zerlegung – Realisierung in Apache + PHP**
  - In **PHP** (*leider etwas technisch – nur Grundidee relevant*)
    - **Ziel:** Wir müssen den URL-Pfad zerlegen und die **Parameter extrahieren**
    - Der URL-Pfad ist in `$_SERVER['REQUEST_URI']` enthalten
      - Genauer ist es (siehe gemäß RFC 2616): `abs_path [ "?" query ]`
      - Z.B.: Wird im obigen Beispiel auf `http://myserver/blog/2015/05/01/` zugegriffen ...
        - So gilt `$_SERVER['REQUEST_URI'] = '/blog/2024/05/01/'`
        - Mit GET-Parametern ggf. `'/blog/2024/05/01/?a=b&c=d'`
    - Den String kann man mit der PHP-Funktion „explode“ zerlegen
      - `explode ( string $delimiter , string $string [, int $limit ] )`
        - Gibt ein Array aus Strings zurück, die jeweils Teil von string sind.  
Die Abtrennung erfolgt dabei an der mit delimiter angegebenen Zeichenkette.
      - `$param = explode ( '/' , '/blog/2024/05/01/test'`) liefert in `$param` den Wert `array( ' ' , 'blog' , '2024' , '05' , '01' , 'test'`)
        - Dem Array könnten wir nun unsere Parameter entnehmen, z.B. `$year = $param[2]`
    - *Für Interessierte:* Dokumentiertes Beispiel
      - <http://forum.codecall.net/topic/74170-clean-urls-with-php/>

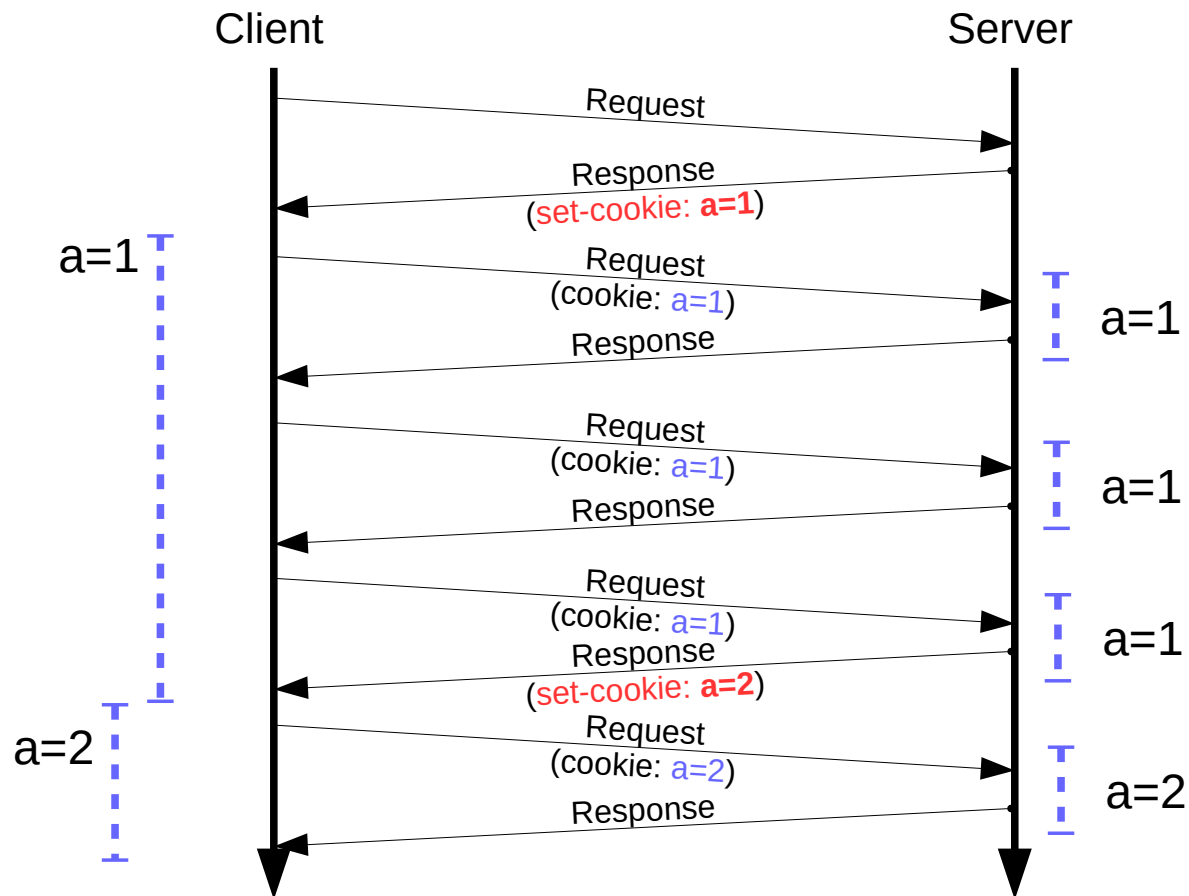
# Rückblick: Das HTTP-Protokoll – Cookies

---

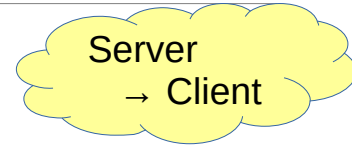
- **Cookies** sind Zeichenketten, die
  - der Server mit einem Response im Client setzen kann und
  - der Client mit jedem Request an den Server zurück überträgt.
- **Durch Cookies kann der Server der Sitzung zwischen Client und Server Attribute zuordnen**
  - Beispiel: Benutzerspezifische Einstellungen
    - „language=de“ oder „sort\_messages=date“
  - Da HTTP zustandslos ist, kann der Server durch die Übertragung der Cookies mit jedem Request solche Einstellungen berücksichtigen ohne sie selbst zu speichern
- **Cookies können über Javascript auch im Client gesetzt werden.**
  - Sie werden beim nächsten Request an den Server übertragen

# Rückblick: Das HTTP-Protokoll – Cookies

- Server: Setzen durch **Set-Cookie**-Response Header
- Client: Rück-Übertragung durch **Cookie**-Request-Header



# Rückblick: Das HTTP-Protokoll – Cookies



- **Semantik des Set-Cookie Response Headers**
  - Der Client speichert unter dem Namen **Name** den Wert **VALUE**.
    - Der Client interpretiert beides nicht! (Ausnahme: mit Javascript-Code)
  - **Comment**: Optionale Information, die der Nutzer lesen könnte
    - Z.B. wozu das Cookie dient. Kann vom Nutzer ggf. in spezieller Funktion des Browsers gelesen werden. Keine technische Nutzung.
  - **Domain**: An welche Server wird das Cookie zurück geliefert
    - Default: Nur an den setzenden Server
  - **Max-Age**: Lebensdauer des Cookies in Sekunden
  - **Path**: Begrenzt den Teilbaum, an den das Cookie geschickt wird
    - Bsp '/test/': Cookie wird bei Request von '/test/daten/' geliefert, nicht aber bei '/x/'
  - **Secure**: Cookie darf nur sicher (über HTTPS) übertragen werden
  - **Version**: Version der Cookie-Grammatik
    - (verpflichtend, „1“ ist aktuell)

# Cookies in PHP

---

- **Wie arbeitet man in PHP mit Cookies?**

- Cookies sind in PHP ganz ähnlich zu GET- und POST-Parametern

- `$_COOKIE` enthält alle vom Client beim Request gelieferten Cookies als Name-Wert-Paare

- Es ist ein assoziatives Array

- um z.B. auf das Cookie „`language`“ zuzugreifen, dient der Ausdruck

- `$_COOKIE[ 'language' ]`

- Diese Variable ist **superglobal**

- d.h. man kann von überall auf sie zugreifen (also ohne „`global $_COOKIE;`“)

- Wie setzt man Cookies vom Server aus?

- `setcookie($name, $value);`

- Die Funktion muss **vor der ersten Ausgabe** aufgerufen werden.

- Sowohl des statischen Webseiteninhalts als auch des dynamischen PHP-Codes.

- Cookie-Werte werden automatisch mit `urlencode` und `urldecode` behandelt.

- Es muss kein Encoding mehr erfolgen



# Cookies in PHP

---

- **Wie arbeitet man in PHP mit Cookies?**

- Es gibt noch diverse weitere **optionale Parameter** ( → [php.net](https://www.php.net))

- `bool setcookie (`  
    `string $name`  
    `[, string $value`  
    `[, int $expire = 0`  
    `[, string $path`  
    `[, string $domain`  
    `[, bool $secure = false`  
    `[, bool $httponly = false`  
    `]]]] )`

- `$expire`: Zeitliche Lebensdauer des Cookies

- `0` (d.h. Cookie verfällt am Ende der Browser-Sitzung)

- Unix-Zeitstempel (Sekunden seit 1.1.1970 in `UTC`), `time()` = jetzt-Zeit, also z.B. für 30 Tage: `time()+60*60*24*30`

- **Cookie Löschen**

- `setcookie($name, "", 1)`

- Verfallsdatum auf Vergangenheit setzen (hier 1.1.1970, 0:00 Uhr + 1 Sekunde)

# Cookies in PHP

---

- **Wie arbeitet man in PHP mit Cookies?**

- Der Aufruf von `setcookie` hat **keine direkte Auswirkung** auf `$_COOKIE` – es setzt nur den Response-Header

- Also so gestalten, dass es ggf. egal ist, ob der Wert aus `$_COOKIE` stammt oder gerade mit `setcookie` neu gesetzt wurde

```
if ($we_want_to_set_the_cookie) {
 $cookievalue = ...;
 setcookie("cookiename", $cookievalue);
}
else
 $cookievalue = @$_COOKIE["cookiename"];
// ab hier $cookievalue benutzen
```

- Oder: nach `setcookie` auch `$_COOKIE` einfach (für diese Requestbehandlung) modifizieren:

```
if ($we_want_to_set_the_cookie) {
 $cookievalue = ...;
 setcookie("cookiename", $cookievalue);
 @$_COOKIE["cookiename"] = $cookievalue;
}
// ab hier kann man @$_COOKIE["cookiename"] benutzen
```

- Die Änderung von `$_COOKIE` selbst hat **keinen dauerhaften Effekt!**

# Cookies in PHP

---

- **Wie arbeitet man in PHP mit Cookies?**

- `setcookie` immer vor der ersten Ausgabe aufrufen.

Also ganz am Anfang die **Applikationslogik** ausführen (z.B. Formulare verarbeiten), ggf. Cookies setzen, etc.

- Erst **danach** Ausgaben machen.

```
<?php
 if (isset($_COOKIE['show_details']))
 // Wert wurde schon mal gesetzt → Cookie-Wert benutzen
 $show_details = @$_COOKIE['show_details'];
 else
 // Wert wurde noch nie gesetzt → Default benutzen
 $show_details = FALSE;
 if (isset($_REQUEST['new_value_show_details'])) {
 // Benutzer will Wert ändern (Formular-Post)
 $show_details = $_REQUEST['new_value_show_details'];
 setcookie('show_details', $show_details);
 }
?>

<!DOCTYPE html>
<html> <head> <!-- ... --> </head> <body> <!-- ... -->
<?php if ($show_details) { ... } ?>
```

# Cookies in PHP

- **Beispiel: Anzahl der Seiten-Aufrufe zählen**

```
<?php
// Wir zählen die Zahl der Aufrufe dieses Clients in Cookies
if (!isset($_COOKIE['number_of_calls']))
 // cookie war ungesetzt → setze Startwert
 $number_of_calls = 0;
else
 $number_of_calls = $_COOKIE['number_of_calls'] + 1;
setcookie('number_of_calls', $number_of_calls);
?>

<!DOCTYPE html>
<html> <head> <!-- ... --> </head>
<body>

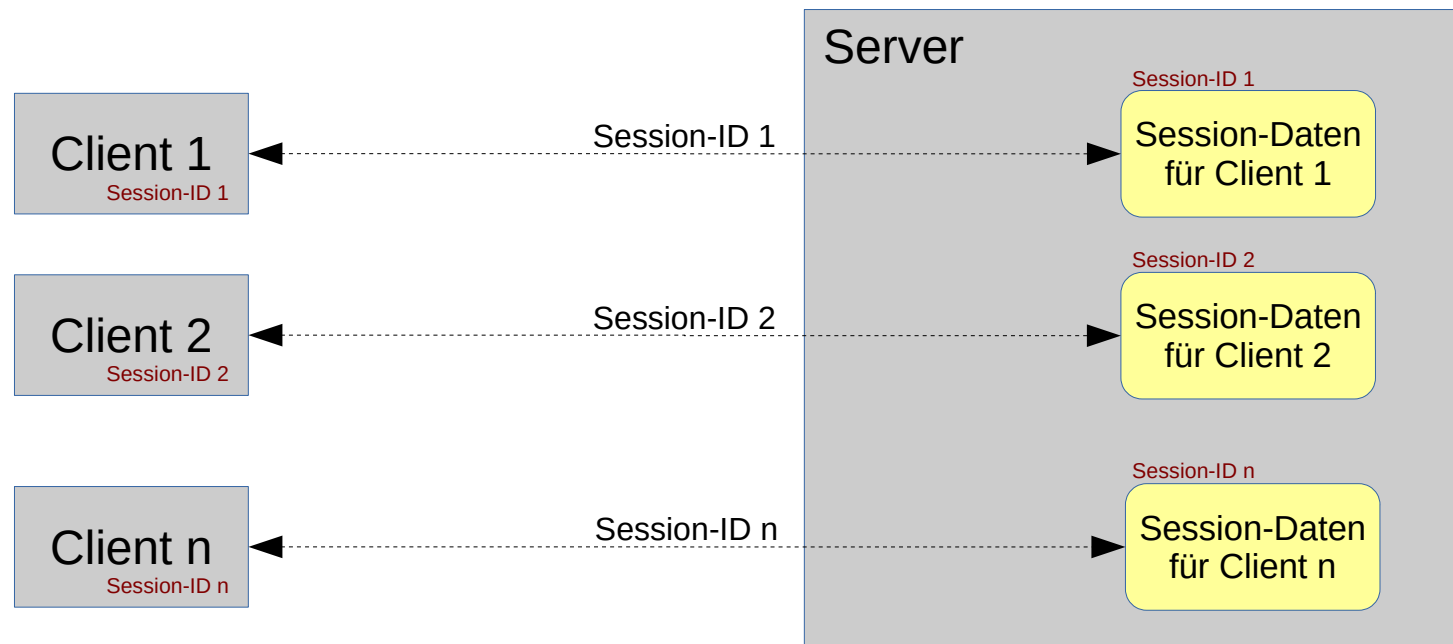
<p>Sie haben
 <?php echo $number_of_calls; ?>
 Aufrufe gemacht.

</body>
</html>
```

- Diesen Zähler kann der Webseitennutzer manipulieren.
  - **Frage:** Wie? Was genau kann man?

# Sessions

- **Session-Daten** („Sitzungsdaten“)
  - Client-spezifische Parameter, die nur im Server zugänglich sind
    - Parameter sind **nur im Server zugänglich**, nicht im Client
    - Parameter werden **im Server** mittelfristig unter einer **Session-ID** gespeichert
      - z.B. in einer **Datei** oder einer **Datenbank**
    - Die **Session-ID** muss vom Client bei (allen) Requests übermittelt werden



# Sessions und Authentifizierungs-Tokens

---

- **Session-IDs in PHP**

- PHP enthält bereits einen Mechanismus, der diese Verwaltung durchführt
  - Ordnet neuen Clients eine Session-ID zu
  - Bietet Interface, um bequem Server-interne Daten zur Session-ID zuzuordnen
  - Speichert diese Daten **dauerhaft**
    - z.B. in einer Datei unter „`/var/lib/php*`“ auf dem Server
- Dazu muss am Anfang die Funktion **`session_start()`** aufgerufen werden (→ [php.net](http://php.net))
  - Dadurch wird ein Cookie „**PHPSESSID**“ mit der generierten Session-ID gesetzt.
  - Zudem wird die superglobale Variable **`$_SESSION`** aktiviert
    - Hier kann man von PHP aus Werte zuweisen, die dauerhaft **im Server** gespeichert werden
    - d.h. auch beim nächsten Request zu dieser Sitzung (Cookie PHPSESSID) sind diese Werte verfügbar
    - Der Client kann diese Werte nicht manipulieren!

Übungsfrage: Warum?

# Sessions und Authentifizierungs-Tokens

- **Sessions nutzen – Beispiel: Anzahl Seitenaufrufe**

- Session-Daten sind auch weitaus handlicher zu ändern als Cookies, da sie auch nach der ersten Ausgaben modifiziert werden können. (Frage: Warum?)

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

<!DOCTYPE html>
<html> <head> <!-- ... --> </head> <body>

<?php
 // Wir zählen die Zahl der Aufrufe in dieser Session
 if (!isset($_SESSION['number_of_calls'])) {
 $_SESSION['number_of_calls'] = 0;
 } else {
 $_SESSION['number_of_calls']++;
 }
 // Die Änderung an der Variablen wird sofort dauerhaft wirksam
?>

<p>Sie haben
 <?php echo $_SESSION['number_of_calls']; ?>
 Aufrufe in dieser Session gemacht.
</body>
</html>
```

# Sessions

---

- **Die Session-ID ...**

- ist für jeden Client unterschiedlich
  - **Identifiziert** aus Sicht des Servers **den Client** eindeutig (über längere Zeit)
- dient zur Zuordnung der im Server gespeicherten **Session-Daten** zur Sitzung (und somit zum Client)
  - Der Server kann viele solche Sessions mit verschiedenen Clients zugleich haben
- muss vom Client bei Requests mitgeliefert werden
  - Meist als **Cookie** („**Session-Cookie**“)
  - Alle Parameterübergabeverfahren (GET, POST, Pfad) sind aber möglich
- fungiert als **Authentifizierungs-Token**
  - Wenn ein Client sich **authentifiziert** hat (also z.B. gültige Login-Daten in einem **Login-Formular** eingegeben wurden), kann der Server die bewiesene Identität auch der **Session** zuordnen, die **Session-ID** wird zum Authentifizierungs-Token.
- darf also nicht erratbar sein oder Dritten offenbart werden
  - Sonst könnte ein Angreifer eine fremde **Session** (und damit Identität) „**stehlen**“



# Sessions und Authentifizierungs-Tokens

---

- **Cookies als Authentifizierungs-Tokens**

- Session-Cookies sind Authentifizierungs-Tokens

- Sie identifizieren den Client (Browser-Instanz) → Nutzer
- Es kann ein **Account/Benutzer** zugeordnet werden, wenn dieser eingeloggt ist
- Aber **auch ohne Login** kann man die Folge von Requests dem Client zuordnen (**anonyme Session**)
  - Beispiel: Warenkorb in einer Shopping-Plattform:  
Man kann Gegenstände in den Warenkorb legen, ohne eingeloggt zu sein

- Wie sollte so ein Session-Cookie aussehen

- (Schlechte) Idee: Benutzername
  - Problem: Manipulierbar (der Nutzer könnte einen anderen Benutzernamen erraten und das Cookie ändern)
  - Problem: Keine anonyme Session mit späterem Login möglich
- Idee: **Zufallszahl** (ausreichend komplex, „**Session-ID**“)
  - Manipulationssicher, da andere Zufallszahl nicht erratbar
  - anonyme Sitzung möglich
  - Session-Daten müssen im Server unter der Session-ID gespeichert werden

# Sessions und Authentifizierungs-Tokens

- Login-Session setzen und nutzen

<?php

```
session_start(); // Session wieder aufnehmen oder ggf. neu erzeugen
```

```
function get_login($id = NULL, $password = NULL) {
```

```
 global $user_data, $user_id;
```

```
 ▶ $user_data = $user_id = NULL;
```

```
 $u = get_userdata($id); // liefert assoz. Array u.a. mit Passwort
```

```
 if ($u && @$u['password'] == $password) { ◀
```

```
 // neuer Login erfolgreich
```

```
 $user_id = $id;
```

```
 $user_data = $u;
```

```
 @$SESSION['user_id'] = $id; // User der Session zuordnen:
 // Zuweisung ist dauerhaft!
```

```
 }
```

```
 elseif (@$SESSION['user_id']) {
```

```
 // Bestehenden User-Login aus Session lesen:
```

```
 $user_id = @$SESSION['user_id'];
```

```
 $user_data = get_userdata($user_id);
```

```
 }
```

```
}
```

```
// ...
```

```
get_login(@$_POST['name'], @$_POST['password']);
```

?>

Default: nicht  
eingeloggt

Hiermit neu  
eingeloggt

Oder  
zuvor bereits  
eingeloggt

# Sessions und Authentifizierungs-Tokens

---

- **Login-Session setzen und nutzen** (Fortsetzung)
  - Die in den Session-Daten gespeicherten Daten gehen beim Ende der Session verloren ...
    - Wenn das Cookie abläuft (Lebensdauer)
    - Wenn das Cookie (z.B. am Ende einer Browser-Sitzung) gelöscht wird
  - Die Eigenschaften des Cookies können gesteuert werden:
    - in der [PHP-Konfigurationsdatei php.ini](#)
      - z.B. `session.cookie_lifetime` int
        - Lebensdauer in Sekunden oder 0 (bis zum Ende der Browser-Sitzung – Default)
        - z.B. für 1 Stunde: „`session.cookie_lifetime 3600`“
    - oder dynamisch mit ...
      - void `session_set_cookie_params` (
        - int \$lifetime
        - [, string \$path
        - [, string \$domain
        - [, bool \$secure = false
        - [, bool \$httponly = false ]]] )
      - vor `session_start()` aufrufen (Verständnisfrage: Warum?)

# PHP-Schnittstelle zu MySQL

---

- **PHP-Schnittstelle zu MySQL**

- Es gibt mehrere Prozedurale und OO-Schnittstellen
- Wir benutzen `mysqli` in prozeduraler Form
  - Siehe <http://php.net/manual/de/book.mysqli.php>
  - Siehe <http://php.net/manual/de/mysqli.quickstart.dual-interface.php>

- **Erste Schritte ...**

- Verbindung zum Server aufbauen:

```
$mysqli = mysqli_connect($mysql_server,
 $mysql_user,
 $mysql_password,
 $mysql_database
);
```

- Mit dem Objekt `$mysqli` kann man dann auf die DB zugreifen
  - Am Ende des PHP-Scripts sollte die Verbindung geschlossen werden

```
mysqli_close($mysqli);
```

oder

```
$mysqli->close();
```

# PHP-Schnittstelle zu MySQL

- Verbindungsaufbau als PHP-Datei

```
<?php
// Zugangsdaten zur DB aus Datei lesen
include('/home/lamp/.mysql_credentials');

// Verbindung aufbauen
$mysqli = mysqli_connect(
 $mysql_server,
 $mysql_user,
 $mysql_password,
 $mysql_database);

/* hier mit $mysqli Queries stellen und verarbeiten */

mysqli_close($mysqli); // oder $mysqli->close();
?>
```

Importiert Inhalt der  
PHP-Datei hierher

- Wir nehmen an, dass in der Require-Datei die benötigten Zugangsdaten liegen:

```
<?php // Zugangsdaten zur DB
$mysql_server = 'localhost';
$mysql_user = 'lamp';
$mysql_password = 'asdf3asdf3asdf';
$mysql_database = 'wikipedia_sql_example';
?>
```

Idee: Zentral  
konfiguriert  
+ nicht im  
htdocs-Baum

# PHP-Schnittstelle zu MySQL

- Query stellen und Daten empfangen

```
<?php
 $mysqli = mysqli_connect(...);

 // Query stellen ...
 $res = mysqli_query($mysqli, "SELECT * FROM Student");

 // Datensätze abrufen und verarbeiten
 while ($row = mysqli_fetch_assoc($res)) {
 print "<pre>";
 print_r($row);
 print "</pre>";
 }

 mysqli_close($mysqli);
?>
```

– Ergebnis:

```
Array
(
 [MatrNr] => 25403
 [Name] => Jonas
)

Array
(
 [MatrNr] => 26120
 [Name] => Fichte
)
```

```
Array
(
 [MatrNr] => 27103
 [Name] => Fauler
)

Array
(
 [MatrNr] => 27104
 [Name] => Peter
)
```

```
Array
(
 [MatrNr] => 27106
 [Name] => Neumann
)
```

# PHP-Schnittstelle zu MySQL

- **Attributzugriff in Datensätzen aus Queries**

```
<?php
 $mysqli = mysqli_connect(...);

 // Query stellen ...
 $res = mysqli_query($mysqli, "SELECT * FROM Student");

 // Datensätze abrufen und verarbeiten
 print "<table>\n<tr><th>MatrNr <th>Name";
 while ($row = mysqli_fetch_assoc($res)) {
 print "\n<tr>";
 print "<td>" . $row['MatrNr'];
 print "<td>" . $row['Name'];
 }
 print "\n</table>";
?>
```

– Ergebnis:

<b>MatrNr</b>	<b>Name</b>
25403	Jonas
26120	Fichte
27103	Fauler
27104	Peter
27106	Neumann

# PHP-Schnittstelle zu MySQL

- Fehlerbehandlung: Verbindungsaufbau

- Fehler bei `mysqli_connect()`

- Liefert False zurück
- `mysqli_connect_errno()` liefert den Fehlercode
- `mysqli_connect_error()` liefert die Fehlerbeschreibung

```
<?php
 $mysqli = mysqli_connect(...);
 if (!$mysqli) {
 print 'Connect Error ('
 . mysqli_connect_errno() . ') '
 . mysqli_connect_error() . "\n";
 exit(1);
 }

 // Hier ist die Verbindung in $mysqli erfolgreich hergestellt ...

?>
```



# PHP-Schnittstelle zu MySQL

- Fehlerbehandlung: Queries

- Fehler bei `mysqli_query()`
  - Liefert False zurück
  - `mysqli_errno($mysqli)` liefert den Fehlercode
  - `mysqli_error($mysqli)` liefert die Fehlerbeschreibung

```
<?php
 $mysqli = mysqli_connect(...);
 if (!$mysqli) { /* ... */ }

 $res = mysqli_query($mysqli, "SELECT * FROM Student");
 if (!$res) {
 print 'Query Error ('
 . mysqli_errno($mysqli) . ') '
 . mysqli_error($mysqli) . "\n";
 exit(1);
 }

 // Hier kann das Query-Ergebnis in $res verarbeitet werden
?>
```

# PHP-Schnittstelle zu MySQL

- Fehlerbehandlung: **Warnungen** bei Queries

- Warnungen bei `mysqli_query()`

- Auch wenn kein Fehler gemeldet wird, können **relevante Probleme** aufgetreten sein, die MySQL nur als **Warnungen** behandelt.
- `mysqli_warning_count($mysqli)` liefert die Anzahl
- `mysqli_get_warnings($mysqli)` Warnungen als interaktives Objekt

```
<?php
 $mysqli = mysqli_connect(...);
 if (!$mysqli) { /* ... */ }

 $res = mysqli_query($mysqli, "SELECT * FROM Student");
 if (!$res) { /* ... */ }
 if (mysqli_warning_count($mysqli)) {
 $warn = mysqli_get_warnings($mysqli);
 do {
 print 'Query Warning ('
 . $warn->errno . ') '
 . $warn->message . "\n";
 } while ($warn->next());
 }
 // Auch bei Warnungen kann das Query-Ergebnis verarbeitet werden
```

# PHP-Schnittstelle zu MySQL

- Fehlerbehandlung: **Warnungen** bei Queries

- Die gelieferten Warnungen entsprechen den Ausgaben des SQL-Kommandos „**SHOW WARNINGS**“
- **Beispiel:** Beim Einfügen eines Datensatzes wird ein Wert nicht explizit angegeben, der NOT NULL ist und keinen DEFAULT hat.
  - z.B. **Student.Name** im obigen Beispiel

```
INSERT INTO Student (MatrNr) VALUES (34567);
Query OK, 1 row affected, 1 warning (0.09 sec)
SHOW WARNINGS;
+-----+-----+-----+
| Level | Code | Message |
+-----+-----+-----+
| Warning| 1364 | Field 'Name' doesn't have a default value |
+-----+-----+-----+

SELECT * FROM Student;
+-----+-----+
| MatrNr | Name |
+-----+-----+
| ... | ... |
| 27105 | Fauler |
| 34567 | |
+-----+-----+
```

# PHP-Schnittstelle zu MySQL

---

- **Fehlerbehandlung und -Meldung allgemein**

- Die obige Art der Fehlerausgabe ist **nur für Testsysteme** sinnvoll.
  - Der Anwender kann mir der Fehlermeldung meist nichts anfangen.
  - Der Administrator erfährt evtl. gar nichts von dem Problem.
  - Es werden durch die Fehlertexte evtl. Informationen an außenstehende geliefert, die für Angriffe genutzt werden können.
- Produktivsysteme sollten dem Endanwender nur (freundlich) anzeigen, dass es überhaupt ein Problem gab.
  - Fehler sollten in ein Logdatei oder eine Log-Datenbank geschrieben werden.
  - Kritische Fehler sollten ggf. aktiv an den Administrator gemeldet werden.
    - Zentrale Systemüberwachung oder Email
  - **Zur Erinnerung:** *Fehlerbehandlung* im Kapitel PHP in Teil 1 der Vorlesung
    - `set_error_handler(my_callback_function, E_ALL);`
      - Eigene Behandlung von Fehlermeldungen
    - `error_log('Etwas schlimmes ist passiert', 1, 'admin@my.domain');`
      - Schreibt Log-Text und sendet Email

# PHP-Schnittstelle zu MySQL

---

- **Queries mit Daten aus Parametern**

- Das PHP-Script erzeugt einen Query-String, der von der Eingabe abhängt
  - Wurde kein Suchausdruck angegeben, werden alle Studenten angezeigt
  - Wurde ein Suchausdruck angegeben, werden alle Datensätze angezeigt, bei denen der Ausdruck im Namen vorkommt.

```
<?php
 $search = @$_GET['name'];
 // ...
 // Query stellen ...
 $query = "SELECT * FROM Student";
 if ($search)
 $query .= " WHERE Name LIKE '%$search%'";
 $res = mysqli_query($mysqli, $query);
```

- **Verständnisfrage:** ist Ihnen klar, wie diese Suche funktioniert?
  - Überlegen Sie ggf. wie der Query-String aussieht und wie LIKE funktioniert

# PHP-Schnittstelle zu MySQL

- Queries mit Daten aus Parametern

```
<?php $search = @$_GET['name']; ?>

<h1>Search for Student</h1>
<form action="" method=get>
 <input type=text name=name value="<?php print $search; ?>">
 <input type=submit name=search value="Search" >
</form>
<h1>Search Result
 <?php if ($search) print "for '$search'"; ?>
</h1>
<?php
 $mysqli = mysqli_connect(...);

 // Query stellen ...
 $query = "SELECT * FROM Student";
 if ($search)
 $query .= " WHERE Name LIKE '%$search%'";
 $res = mysqli_query($mysqli, $query);

 // Datensätze abrufen und verarbeiten
 print "<table>\n<tr><th>MatrNr <th>Name";
 while ($row = mysqli_fetch_assoc($res)) {
 print "\n<tr>";
 print "<td>" . $row['Name'];
```

# PHP-Schnittstelle zu MySQL

- **Queries mit Daten aus Parametern (Angriffe)**

- **Beispiel:** Suchausdruck „au“

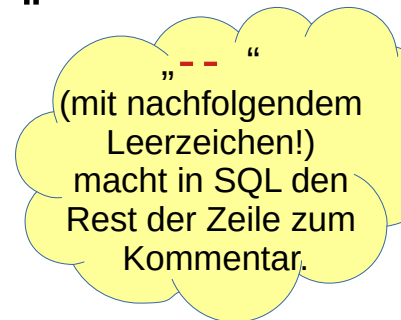
- → `SELECT * FROM Student WHERE Name LIKE '%au%'`
  - Korrekt, liefert alle Studenten, deren Name „au“ enthält

- **Beispiel:** Suchausdruck „'“ (das Apostroph-Zeichen)

- → `SELECT * FROM Student WHERE Name LIKE '%''%`
  - Der Query-String ist offensichtlich nicht syntaktisch korrekt
  - Es entsteht eine Fehlermeldung aus der Datenbank

- **Beispiel:** Suchausdruck „' AND MatrNr > 27000 --“

- → `SELECT * FROM Student WHERE  
Name LIKE '%' AND MatrNr > 27000 -- %'`
  - Der String ist syntaktisch korrekt, bedeutet aber etwas anderes

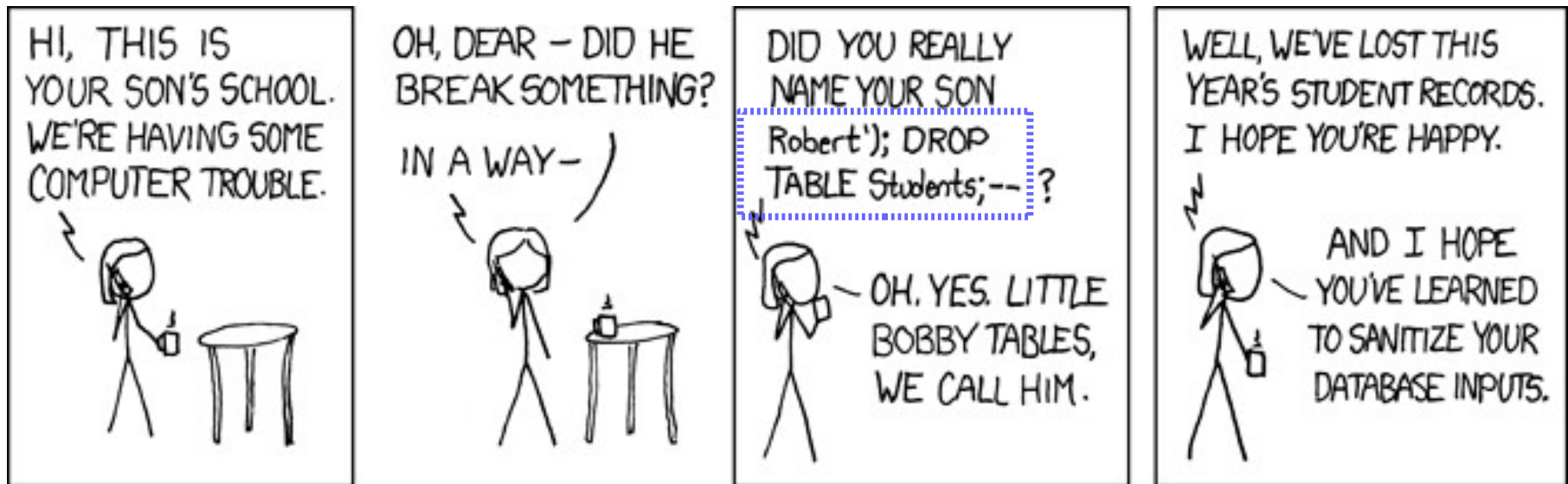


→ Ein Teil der **SQL-Instruktion** stammt **aus Nutzerdaten!**

- **Das nennt man SQL-Injection**

# Injections

- **XKCD-Comic:** „*Little Bobby Tables*“



<https://xkcd.com/327/>

## – Übung:

- Wie sah das gesamte SQL-Statement hier also (vermutlich) aus?
- Was soll es bewirken?



# Injections

---

- **SQL-Injection**

- Bei einer **SQL-Injection stammen** nicht nur (wie beabsichtigt) Daten im Query-String, sondern auch **SQL-Statements, Bedingungen, etc. aus unzuverlässigen Quellen** (z.B. vom Endanwender)
- Ursache: Bei der Bildung des Query-Strings werden SQL-Statements und Benutzer-Daten kombiniert.
  - Wenn bei der Zerlegung des entstandenen Strings eine andere Interpretation entsteht als ursprünglich beabsichtigt, können Benutzerdaten fälschlich als SQL-Statements interpretiert werden
  - Siehe auch <http://de.wikipedia.org/wiki/SQL-Injection>

- **Mögliche Folgen**

- Syntaktische Fehler beim Datenbankzugriff
- Manipulation von Auswahl-Bedingungen (WHERE-Bedingungen)
  - Dadurch kann man z.B. fremde Datensätze sehen
- Einschleusung kompletter Datenbank-Anweisungen
  - Dadurch kann man volle Kontrolle über die Datenbank erlangen

# Injections

- **Mögliche Folge: Ausweitung von Ergebnis-Menge**

- Dadurch kann man z.B. fremde private Datensätze sehen

- **Beispiel:**

- Ein eingeloggter Student soll die von ihm selbst (MatrNr aus Session) belegten Vorlesungen nach Titeln durchsuchen können.

```
<?php
$search = @$_GET['titel'];
$matrnr = @$_SESSION['MatrNr']; // MatrNr des eingeloggten Stud.
// ...
$query = "SELECT * FROM hört JOIN Vorlesung USING (VorlNr) WHERE";
if ($search)
 $query .= " Titel LIKE '%$search%' AND";
$query .= " MatrNr=$matrnr";
```

- Beabsichtigte Query (Suchausdruck „ET“):

- `SELECT * FROM hört JOIN Vorlesung USING (VorlNr) WHERE Titel LIKE '%ET%' AND MatrNr=25403`

Liefert nur eigene Vorlesungen

- Query durch Injection-Suchausdruck „' --“

- `SELECT * FROM hört JOIN Vorlesung USING (VorlNr) WHERE Titel LIKE '%' -- '%' AND MatrNr=25403`

Liefert Vorlesungen aller Studenten

# Injections

- **Mögliche Folge: Einschleusen neuer SQL-Anweisungen**

- Dadurch kann man beliebige Manipulationen in der Datenbank vornehmen (im Rahmen der Rechte des Benutzers)
- **Beispiel** (zu Studenten-Suchfunktion von oben):

- **Injection-Suchausdruck**

```
„ ' ; DELETE FROM Professor -- '“
```

- **Resultierende Anfrage:**

```
SELECT * FROM Student WHERE Name LIKE '% ' ;
DELETE FROM Professor -- '% '
```



- Löscht alle Professor-Datensätze

- **Analog:** ändern von Passwörtern, exmatrikulieren von Studierenden, ...

- Dieses Beispiel funktioniert bei [mysqli\\_query](#) nicht, da hier keine **mehrteiligen SQL-Queries** erlaubt sind (→ dann Syntax-Fehler)

# Injections

- **Gegenmaßnahmen zu SQL-Injections (1)**

- Daten im Query-String vor Einbettung geeignet „**escapen**“

- Vermeidet die Unterbrechung des umgebenden ' ... ' -Strings
- Funktion: **mysqli\_real\_escape\_string** ( mysqli \$link , string \$escapestr )
  - Behandelt **NUL** (ASCII 0), **\n**, **\r**, **\\**, **'**, **"**, **Control-Z**.
  - Wird auf jeden Parameter einzeln angewandt

- Beispiel:

```
<?php
$search = @$_GET['name'];
// ...
// Query stellen ...
$query = "SELECT * FROM Student";
if ($search) {
 $xsearch = mysqli_real_escape_string($mysqli, $search)
 $query .= " WHERE Name LIKE '%$xsearch%'";
}
$res = mysqli_query($mysqli, $query);
```

- **Verständnisfrage**: Kann ich den ganzen Query-String damit behandeln?

# Injections

---

- **Gegenmaßnahmen zu SQL-Injections (2a)**
  - Idee: Daten nicht in Query-String einbetten, sondern **separat** an das DBMS übergeben
  - **Technik: Prepared Statements**
    - Ich formuliere zunächst den Query-String ohne Daten.
      - Er enthält überall ein „?“, wo Daten eingefügt werden sollen.
      - Beispiel: „**SELECT \* FROM Student WHERE a=? AND b=?**“
      - Ziel: Ein „prepared Statement“
    - Erst in einem späteren Aufruf „binde“ ich Parameter an das prepared Statement
      - Beides wird getrennt an das DBMS übergeben und verarbeitet.
      - Es es gibt also keine Vermischung von Code (SQL-Statements) und Daten

# Injections

- **Gegenmaßnahmen zu SQL-Injections (2b)**

- Daten nicht in Query-String einbetten, sondern **explizit** übergeben

- **Technik: Prepared Statements**

- Query-String enthält überall ein „?“ , wo Daten eingefügt werden sollen.
- mit `mysqli_prepare()` wird ein Prepared Statement erzeugt
- mit `mysqli_stmt_bind_param()` werden die Parameterwerte gebunden
- mit `mysqli_stmt_execute()` wird der Query ausgeführt
- mit `mysqli_stmt_get_result()` wird das Ergebnis geholt

- **Beispiel:**

```
<?php
$a = @$_GET['a']; $b = @$_GET['b'];
$stmt = mysqli_prepare($mysqli,
 "SELECT * FROM Student WHERE a=? AND b=?");
mysqli_stmt_bind_param($stmt, 'ss', $a, $b);
mysqli_stmt_execute($stmt);
$res = mysqli_stmt_get_result($stmt);

while ($row = mysqli_fetch_assoc($res)) {
 // ...
}
```

- Der 2. Parameter in `mysqli_stmt_bind_param()` gibt die Typen an
  - 's'=String, 'i'=integer, 'd'=Fließkommazahl, 'b'=BLOB

# Injections

---

- **Gegenmaßnahmen zu SQL-Injections (3)**
  - Keine mehrteiligen SQL-Statements in Query-Strings zulassen
    - Dies ist bei [mysqli\\_query](#) ja gesichert (s.o.)
    - Es bietet aber keine vollständige Sicherheit
  - Filterung von Eingaben
    - z.B. bekannte Injection-Muster erkennen und ablehnen oder bereinigen
    - Problem: **False Positives**
      - Zulässige Eingaben werden abgelehnt oder verändert
    - Problem: **False Negatives**
      - Unbekannte (neue / modifizierte) Angriffe werden nicht sicher erkannt
- **SQL-Injections können sehr vielseitig ablaufen**
  - Siehe auch <http://de.wikipedia.org/wiki/SQL-Injection>
    - Überschreiben von Server-Dateien mit INTO OUTFILE
    - Überlasten des Servers, löschen von Tabellen, Ändern von Rechten, ...

# Injections

---

- **Injections**

- Allgemein bezeichnet man mit „**Injection**“ jede Form von **fremdbestimmten Daten**, die als **Code** interpretiert werden
- „**Code**“ in diesem Sinne sind **alle Kontrollstrukturen**
  - also nicht nur klassische Programmiersprachen
  - **Beispiele:**
    - SQL-Statements,
    - Javascript-Code,
    - HTML-Tags und HTML-Tag-Parameter
- „**fremdbestimmte Daten**“ sind alle Daten, die von nicht vertrauenswürdiger Seite beeinflussbar sind
  - **Beispiele:**
    - Benutzereingaben (GET- oder POST-Parameter)
    - Cookies
    - HTML-Header
    - hochgeladene Dateien



# Injections

---

- **Injections (Beispiele)**

- Über einen POST-Parameter werden **SQL-Fragmente** übergeben, die eine Verfälschung der in der Folge erzeugten **SQL-Queries** verursachen
  - **SQL-Injection**
  - Gefahr: Fehler, Manipulation der Datenbank, Geheimnis-Enthüllung
  - Gegenmaßnahmen (s.o.):
    - Parameter-Escaping
    - sichere Query-Erzeugung
    - Daten-Filterung (evtl. unsicher)
- Das obige Script enthält eine solche Verwundbarkeit

# Injections

- Queries mit Daten aus Parametern

```
<?php $search = @$_GET['name']; ?>

<h1>Search for Student</h1>
<form action="" method=get>
 <input type=text name=name value="<?php print $search; ?>">
 <input type=submit name=search value="Search" >
</form>
<h1>Search Result
 <?php if ($search) print "for '$search'"; ?>
</h1>
<?php
 $mysqli = mysqli_connect(...);

 // Query stellen ...
 $query = "SELECT * FROM Student";
 if ($search)
 $query .= " WHERE Name LIKE '$search'";
 $res = mysqli_query($mysqli, $query);

 // Datensätze abrufen und verarbeiten
 print "<table>\n<tr><th>MatrNr <th>Name";
 while ($row = mysqli_fetch_assoc($res)) {
 print "\n<tr>";
 print "<td>" . $row['Name'];
```

# Injections

---

- **Injections (Beispiele)**

- Über einen GET-Parameter werden **HTML-Fragmente** übergeben, die eine Verfälschung der in der Folge erzeugten **HTML-Seite** verursachen.
- Diese HTML-Fragmente können auch **Javascript** enthalten, die im Browser eines Nutzers ausgeführt werden
  - **HTML-Injection**
  - **Javascript-Injection („XSS“ = Cross-Site-Scripting)**
  - Gefahr: Fehlerhafte Darstellung, Manipulation der Benutzer-Webseite, Geheimnis-Enthüllung durch Javascript (z.B. gefälschte Passwort-Dialoge)
  - Gegenmaßnahmen:
    - Parameter-Escaping (PHP-Funktion **htmlspecialchars**)
    - Daten-Filterung (z.B. PHP-Funktion **strip\_tags**) (allgemein evtl. unsicher)
- Das obige Script enthält zwei solcher Verwundbarkeiten

# Injections

- HTML-Injection, XSS

```
<?php $search = @$_GET['name']; ?>

<h1>Search for Student</h1>
<form action="" method=get>
 <input type=text name=name value="<?php print $search; ?>">
 <input type=submit name=search value="Search">
</form>
<h1>Search Result
 <?php if ($search) print "for '$search'"; ?>
</h1>
<?php
 $mysqli = mysqli_connect(...);

 // Query stellen ...
 $query = "SELECT * FROM Student";
 if ($search)
 $query .= " WHERE Name LIKE '$search'";
 $res = mysqli_query($mysqli, $query);

 // Datensätze abrufen und verarbeiten
 print "<table>\n<tr><th>MatrNr <th>Name";
 while ($row = mysqli_fetch_assoc($res)) {
 print "\n<tr>";
 print "<td>" . $row['Name'];
```

# Injections

---

- **HTML-Injection, XSS (Demo)**

- **Beispiel:** Demo-XSS-Injection String für Such-Feld:

`" onclick="alert('Hallo')"`

Liefert auf dem verwundbaren Input-Feld:

```
<input type="text" name="name" value="" onclick="alert('Hallo')">
```

Öffnet also beim Klick auf das Text-Eingabefeld einen Alert.

- **Übung**

- Überlegen Sie sich einen Injection-String, durch den Sie das Action-Feld des Formulars auf einen beliebige URL setzen können.  
(Zwei JS-Funktionsaufrufe genügen.)

- **Angriffsszenario** (bei nicht geschützten Servern)

- Sie ergänzen die Login-URL einer Bank mit einem GET-Parameter, durch den die Login-Daten an einen fremden Server geschickt werden. Dann schicken Sie die URL per gefälschter Email an einen Kunden mit der Aufforderung, sich „*ganz dringend*“ darüber einzuloggen.

# Injections

---

- **Persistente / Reflektierte Injections**
  - Injections müssen nicht immer unmittelbar durch einen Query erfolgen
  - Sie können auch über Daten erfolgen, die zunächst auf dem Server dauerhaft („**persistent**“) gespeichert werden und erst bei einer späteren Abfrage einen Effekt haben
    - Sie werden sozusagen vom Server „**reflektiert**“
  - Beispiel:
    - In der Datenbank liegt ein Fragment für eine Javascript-Injection (**XSS**)
    - In der Datenbank liegt ein Fragment für eine **SQL-Injection**
      - **Verständnisfrage**: Wie kann das sein? Sie wurde doch escaped beim Einfügen?
    - Über eine Upload-Funktion wird ein PHP-Script hochgeladen, das vom Server beim späteren Zugriff ausgeführt wird (**Script-Injection**)
      - Gegenmaßnahme: Im Upload-Bereich immer aktives Scripting deaktivieren!
  - Das obige Script enthält eine solche Verwundbarkeit

# Injections

- **Persistente HTML-Injection, XSS**

```
<?php $search = @$_GET['name']; ?>

<h1>Search for Student</h1>
<form action="" method=get>
 <input type=text name=name value="<?php print $search; ?>">
 <input type=submit name=search value="Search" >
</form>
<h1>Search Result
 <?php if ($search) print "for '$search'"; ?>
</h1>
<?php
 $mysqli = mysqli_connect(...);

 // Query stellen ...
 $query = "SELECT * FROM Student";
 if ($search)
 $query .= " WHERE Name LIKE '$search'";
 $res = mysqli_query($mysqli, $query);

 // Datensätze abrufen und verarbeiten
 print "<table>\n<tr><th>MatrNr <th>Name";
 while ($row = mysqli_fetch_assoc($res)) {
 print "\n<tr>";
 print "<td>" . $row['Name'];
```

# Injections

---

- **Persistente / Reflektierte Injections (Demo)**

- **Beispiel:** Demo-DB-XSS-Injection String für Studenten-Name:

```
INSERT INTO Student (Name) VALUES
(' <script>alert("DB")</script>');
```

Öffnet beim Auflisten des Studenten in jeder Such-Ausgabe einen Alert.

- **Übung**

- Die DB-Textfelder sind evtl. sehr kurz für komplexeren JS-Code (Student.Name hier 64 Zeichen). Wie könnte man das umgehen?
- Können Sie sich ein Szenario vorstellen, in dem man ausdrücklich Daten aus der Datenbank ungeschützt ausgeben will?

- **Angriffsszenario (bei nicht geschützten Servern)**

- Sie schreiben in ein Forum einen Kommentar. Auf der Einstiegsseite des Forums werden die neuesten Kommentare (ungeschützt) angezeigt. Der JS-Code im Kommentar sendet jeweils das Session-Cookie als GET-Parameter an einen fremden Server.



# Injections

- **Abgesichertes Beispiel** → `mysqli_04_search_safe` [Quelle](#) + [Ausführbar](#)

```
<!DOCTYPE html>
<html>
<head></head>
<body>
<?php $search = @$_GET['name']; ?>

<h1>Search for Student</h1>
<form action="" method=get>
 <input type=text name=name value="<?php print htmlspecialchars($search); ?>">
 <input type=submit name=search value="Search" >
</form>

<h1>Search Result
 <?php if ($search) print "for '".htmlspecialchars($search)."'"; ?>
</h1>
<?php
require('/home/lamp/.mysql_credentials');
$mysqli = mysqli_connect($mysql_server, $mysql_user, $mysql_password, $mysql_database);
if (!$mysqli) {
 echo "Failed to connect to MySQL: " . mysqli_connect_error();
 exit(1);
}
$query = "SELECT * FROM Student";
if ($search) {
 $search = mysqli_real_escape_string($mysqli, $search);
 $query .= " WHERE Name LIKE '%$search%'";
 // $query .= " OR MatrNr LIKE '%$search%'";
}
print '<div style="margin: 1em 0; padding: 0 1em; border: thin solid grey;">';
print "<pre>" . htmlspecialchars($query) . "</pre>";
$res = mysqli_query($mysqli, $query);
if (!$res) {
 echo "Failed to Query MySQL: " . mysqli_error($mysqli);
 exit(1);
}
print "</div>";
print "<table border=1>\n<tr><th>MatrNr <th>Name";
while ($row = mysqli_fetch_assoc($res)) {
 print "\n<tr>";
 print "<td>" . htmlspecialchars($row['MatrNr']);
 print "<td>" . htmlspecialchars($row['Name']);
}
print "\n</table>";
mysqli_close($mysqli);
?>
</body>
</html>
```