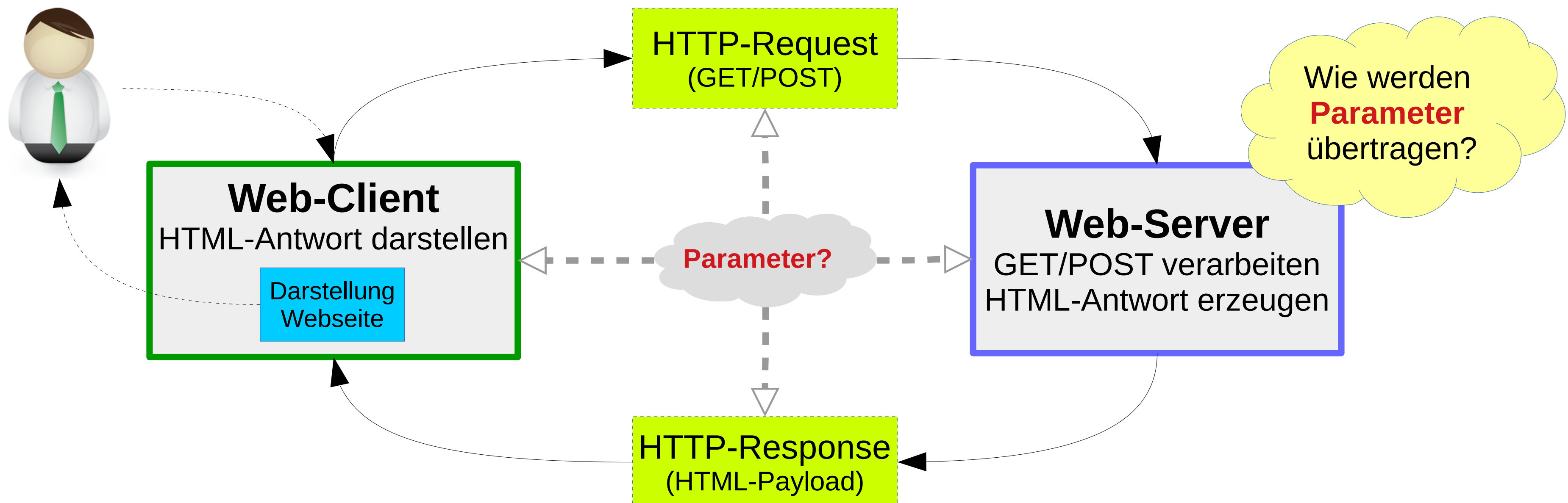
# Struktur von Web-Applikationen

- **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.

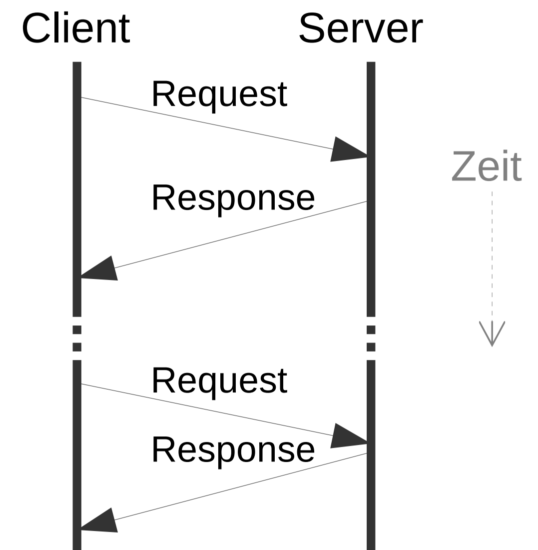
# Struktur von Web-Applikationen

- **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) Benutze schickt **Formular** ab (**form**-Element mit action-Param.)
     - ```
       <form action="/login.html" method=get>
           Name:   <input type=text     name=name     >           <br>
           Passwd: <input type=password name=password >           <br>
                   <input type=submit   name=login    value="Login">
       </form>
       ```

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

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

     Name:
     Passwd.:
     Login
  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

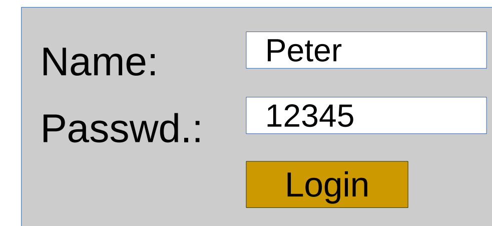
  - `Location: http://myserver/mypath`
  - Meist zusammen mit den Status-Codes
    - **301** Moved Permanently
    - **307** Temporary Redirect
  - Die angegebene URL wird per GET abgerufen
    - *Interessant Rande*: Noch nicht lange (RFC7231, Juni 2014) darf es sich beim Ziel um eine relative URI handeln. 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*<br>*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




- **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 <u>könnte</u> hier (fast) beliebige Strukturen nutzen
    - z.B. `<a href="/login?name*peter!passwort*12345">Login</a>`
  - Man müsste dann nur dafür sorgen, dass mein Server das versteht.

Bitte nicht merken (<u>unrealistisches</u> Beispiel)!

- **Wie nutzt man das in PHP (also auf Serverseite)?**
  - Die Variable **$_SERVER** enthält den kompletten Query-String:
    - `$_SERVER['QUERY_STRING'] == 'name*peter!passwort*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
```

# 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<br>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,<br>wenn <u>wir</u> 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: **raw**urlencode und **raw**urldecode
    - 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 brauch 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

```
<a href="/login.php?name=Firma%20Schmitt%26Partner">Login</a>
```

    - Ohne `urlencode` wäre ein Fehlerhafter Query-String entstanden:

```
<a href="/login.php?name=Firma Schmitt&Partner">Login</a>
```

# 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

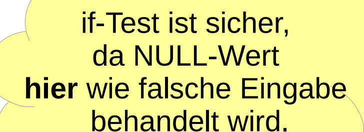
# Verarbeitung von GET- und POST-Daten

- **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 …
}
```



# 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 Datentbank-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=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 mir `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!!!

„HTML-Injection“
(später mehr dazu)

  - Derartiges passiert überall, wo *unsichere Inhalte* in HTML eingebaut werden
    - <p>Hallo <?php echo **$name**; ?>, dein Passwort war nicht richtig.</p>

# Schutz vor gefährlichen Inhalten

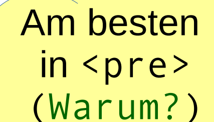
- **Wir wollen alle für uns gefährlichen Zeichen los werden …**
  - „<“ und „>“, um zu verhindern dass Tags erzeugt werden (s.o.)
  - Anführungszeichen, um zu verhindern dass man HTML-Tag-Parameter-Strings vorzeitig beendet (s.o.)
    - u.U. genügt es, nur Doppelte „"“ zu entfernen, wenn man in für Tag-Parameter-Strings 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 '&'.
  - „"“ (doppeltes Anführungszeichen) wird zu '"'
  - „<“ (kleiner als) wird zu '<'
  - „>“ (größer als) wird zu '>'

# Schutz vor gefährlichen Inhalten

- Wirkung von **htmlspecialchars()**
  - Wandelt die o.g. HTML-Sonderzeichen um in HTML-Codes
    - ```
      <?php
          $dangerous = "<a href='test'>Test</a>";
          $safe = htmlspecialchars($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=&amp;#039;test&amp;#039;&gt;Test&lt;/a&gt;
  - Diverse weitere Optionen (siehe php.net)
- Wirkung von **htmlentities()**
  - Wandelt darüber hinaus z.B. Umlaute (‚ä‘ → ‚&amp;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**:

      | Ergebnis | Browser-Anzeige |
      |---|---|
      | & | □ |
      | &amp;amp; | & |
      | &amp;amp;amp; | &amp;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
        $text = 'Das "A&O" ist alles.';
        echo strip_tags($text)            . "\n";
        echo strip_tags($text, '<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
)
```




  - 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();                        // Ausgabe der 3
var_dump($a);                      // Zeilen abfangen
$a_as_text = ob_get_clean();       // 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] =&gt; rot
    [3] =&gt; grün
    [b] =&gt; &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**
  - <u>Diverse</u> 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     > <br>
        Passwd.: <input type=password name=password > <br>
                 <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  }  ?>
```
```