

PHP-Grundlagen

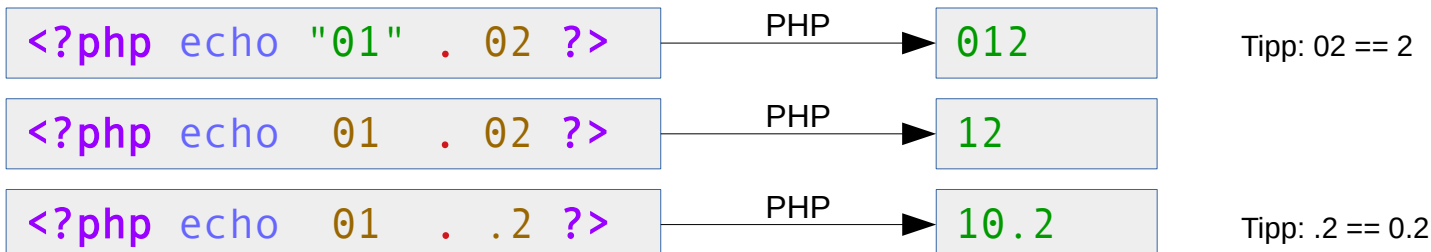
- **String-Werte können mit „.“ verkettet werden**

- 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. implizit 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

```
<?php echo "1" + "2" ?>
```

PHP

3

```
<?php echo "01" + "2" ?>
```

PHP

3

```
<?php echo 10 + ".2" ?>
```

PHP

10.2

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

```
<?php echo "1x" + "2y" ?>
```

PHP

3

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

```
<?php echo 10 + "" ?>
```

PHP

10

```
<?php echo "x" + "y" ?>
```

PHP

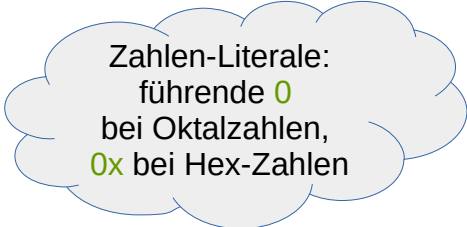
0

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"**



Zahlen-Literale:
führende **0**
bei Oktalzahlen,
0x bei Hex-Zahlen

- 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

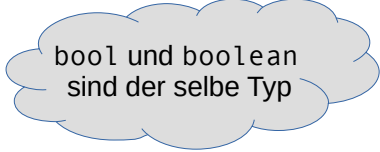
- Ressource
 - **NULL** Wert: **NULL**

- 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

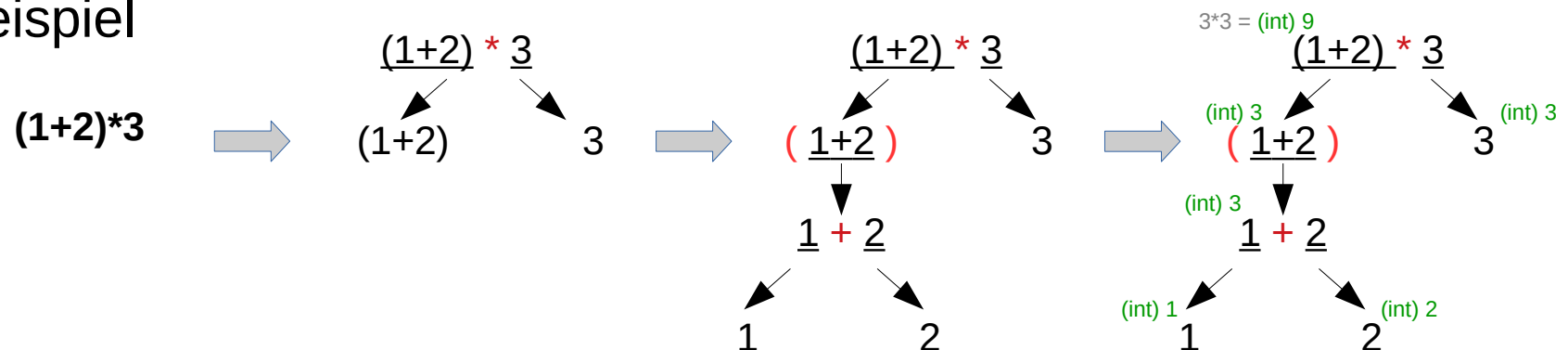
- z.B.: `$list = array(1,2,3); /* ... */ if ($list) { ... }`
- Diese Werte werden bei der Konvertierung zu **integer** entsprechend auch als 0 interpretiert.
 - Beachten Sie die Anomalie bei den beiden String-Werten!
- <https://www.php.net/manual/de/language.types.type-juggling.php>

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.

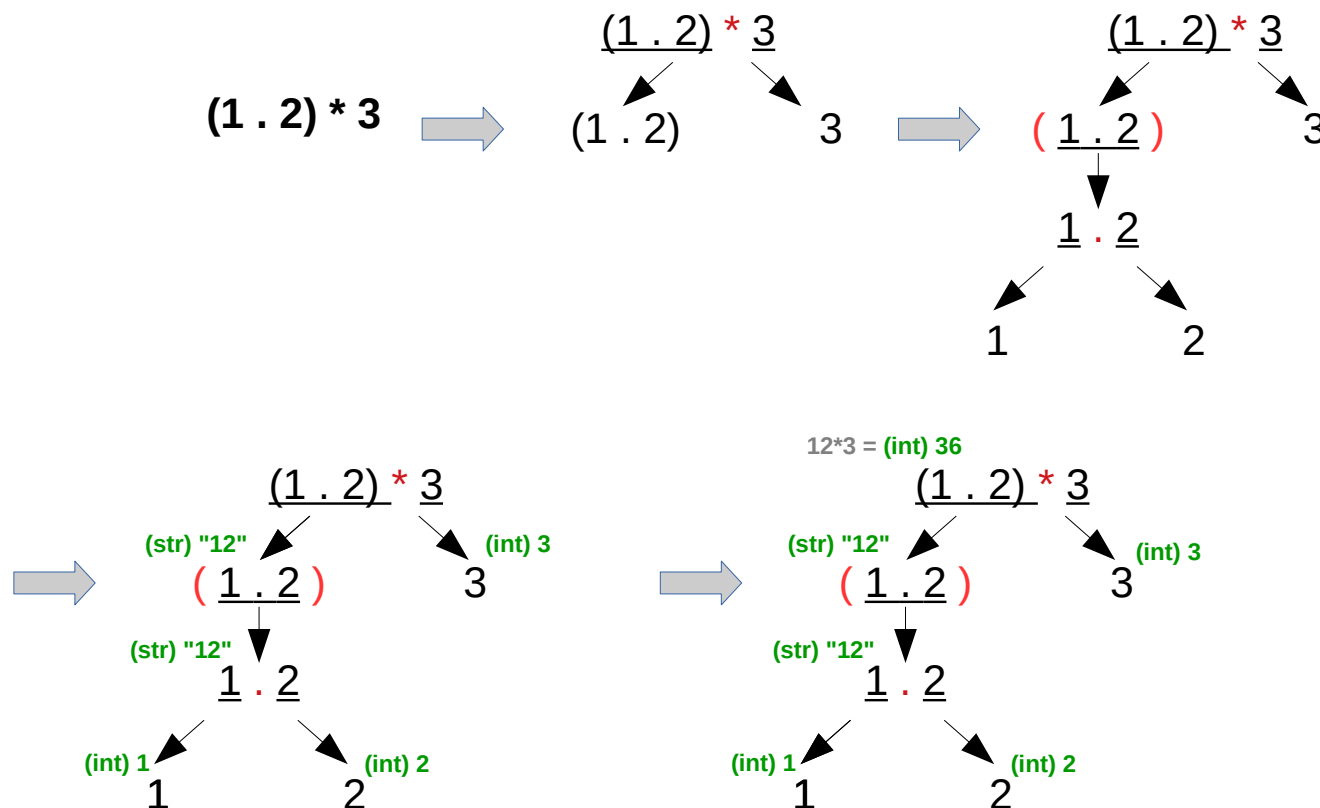
- **Beispiel**



PHP-Grundlagen

- **Ausdrücke (Expression): Typkonvertierungen**

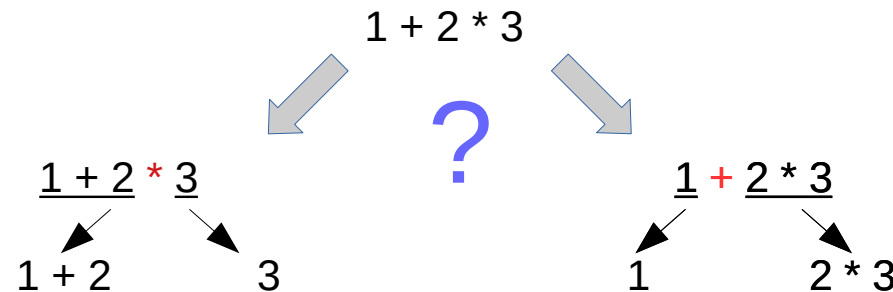
- Bei der Berechnung der Ergebnisse muss man auf implizite Typkonvertierungen achten
- Beispiel



PHP-Grundlagen

- **Ausdrücke (Expression): Präcedenzen**

- Die Zerlegung scheint manchmal nicht eindeutig zu sein



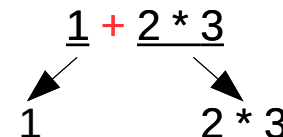
- Die Operatoren haben dazu eine **Rangfolge (Präzedenz)**
 - <https://www.php.net/manual/de/language.operators.precedence.php>
 - In der Tabelle höher stehende Werte „binden stärker“
→ die schwächeren Operatoren werden immer zuerst zerlegt

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

- Ergebnis im obigen Beispiel:

- $*$ bindet stärker als $+$

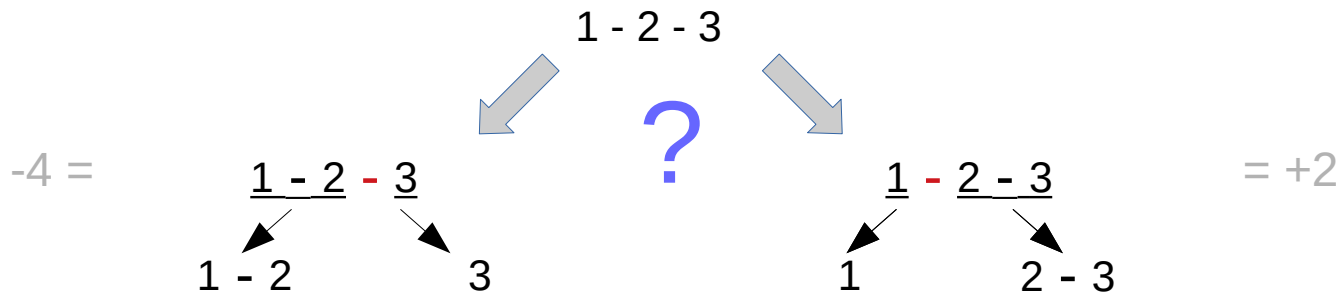
$1 + 2 * 3$



PHP-Grundlagen

- **Ausdrücke (Expression): Assoziativität**

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

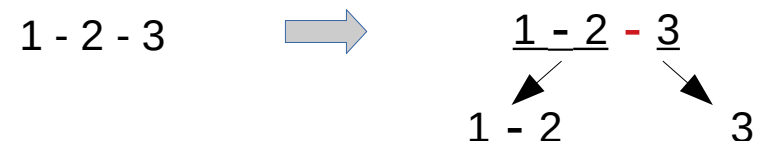


- Der Vorrang Operatoren gleicher Präzedenz (gleiche Zeile) wird über die **Assoziativität** (erste Spalte) geregelt

- <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

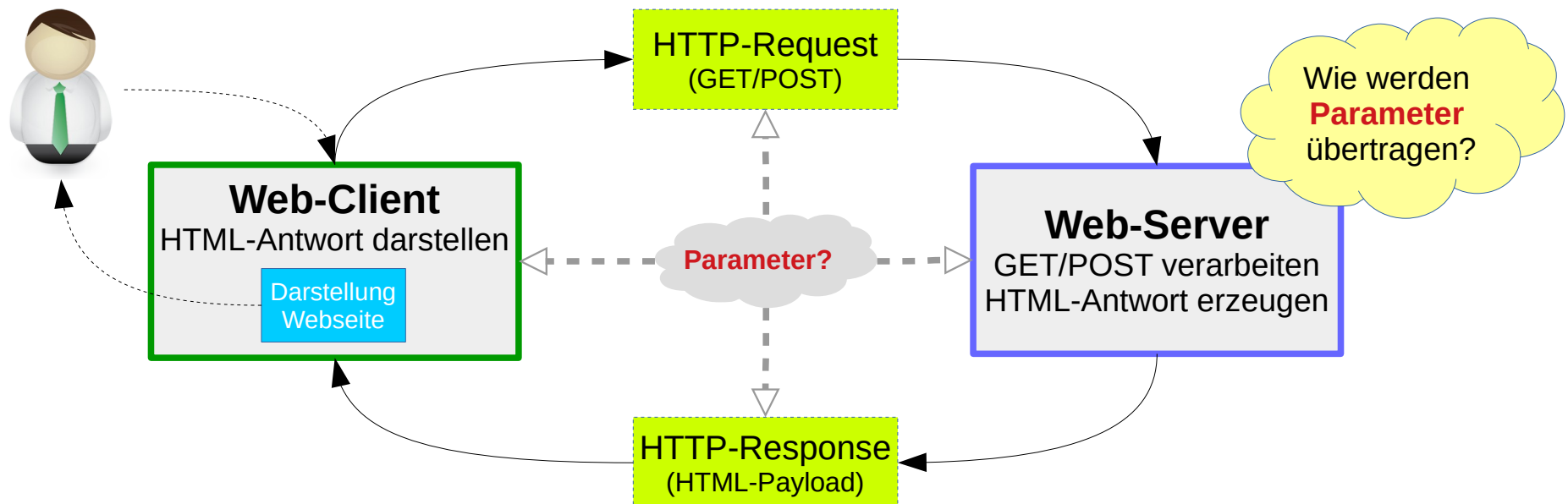


Struktur von Web-Applikationen

- **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:**

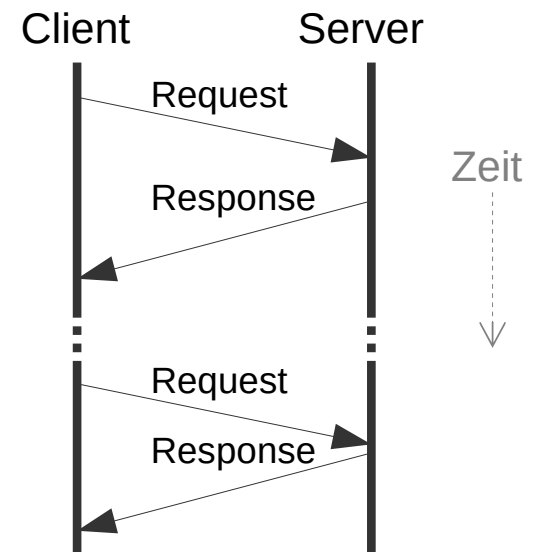
- **Request** vom Client an den Server (Anfrage)
- **Response** vom Server an den Client (Antwort)

- **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.)

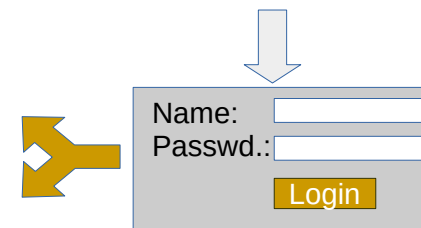
- ```
<form action="/login.html" method=get>
 Name: <input type=text name=name >

 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 representation of a login form. The form is a light gray box containing two text input fields. The first field is preceded by the label 'Name:' and the second by 'Passwd:'. Below these fields is a yellow button with the text 'Login' in black.

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 login form with a light gray background. It contains two input fields: the first is labeled 'Name:' and contains the text 'Peter'; the second is labeled 'Passwd.:' and contains the text '12345'. Below the second input field is a yellow button with the text 'Login'.

- **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. `<a href="/login?name*peter!passwort*12345">Login</a>`
- 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!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
```

} 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 `"-", "_", ".", "~"`
 - 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
`<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>`