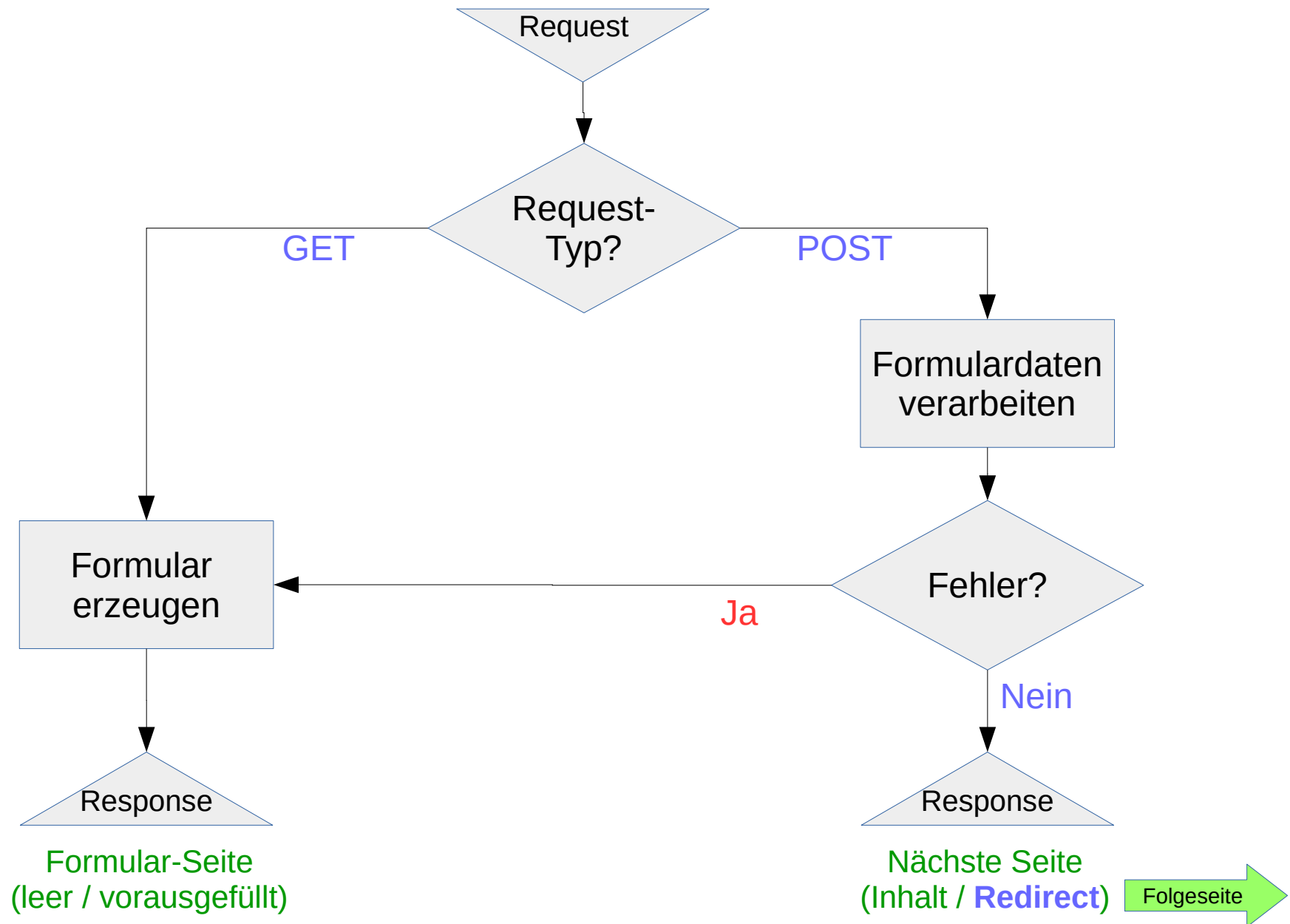


Postback

- **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 Warenkorbs“*
  - 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 Warenkorbs 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.*

mehr Requests ✓

weniger Requests ✓

### – 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 ...*

mehr Requests ✗

weniger Requests ✗

# 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** (verständlich) 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=0xF836A20564014AA9BFC1BD5A665B520D&tguid=0xE9A48F1EED9A4ECDB5A5775406C46C8D> ...
      - Das ist „*offensichtlich*“ die KIS-Seite dieser Vorlesung (INF-00-32-V-3) im SS 2025
      - Der Parameter „tguid“ gibt das Semester an (32 hex-Ziffern →  $16^{32} \approx 3 \cdot 10^{38}$  Werte)
      - **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/>

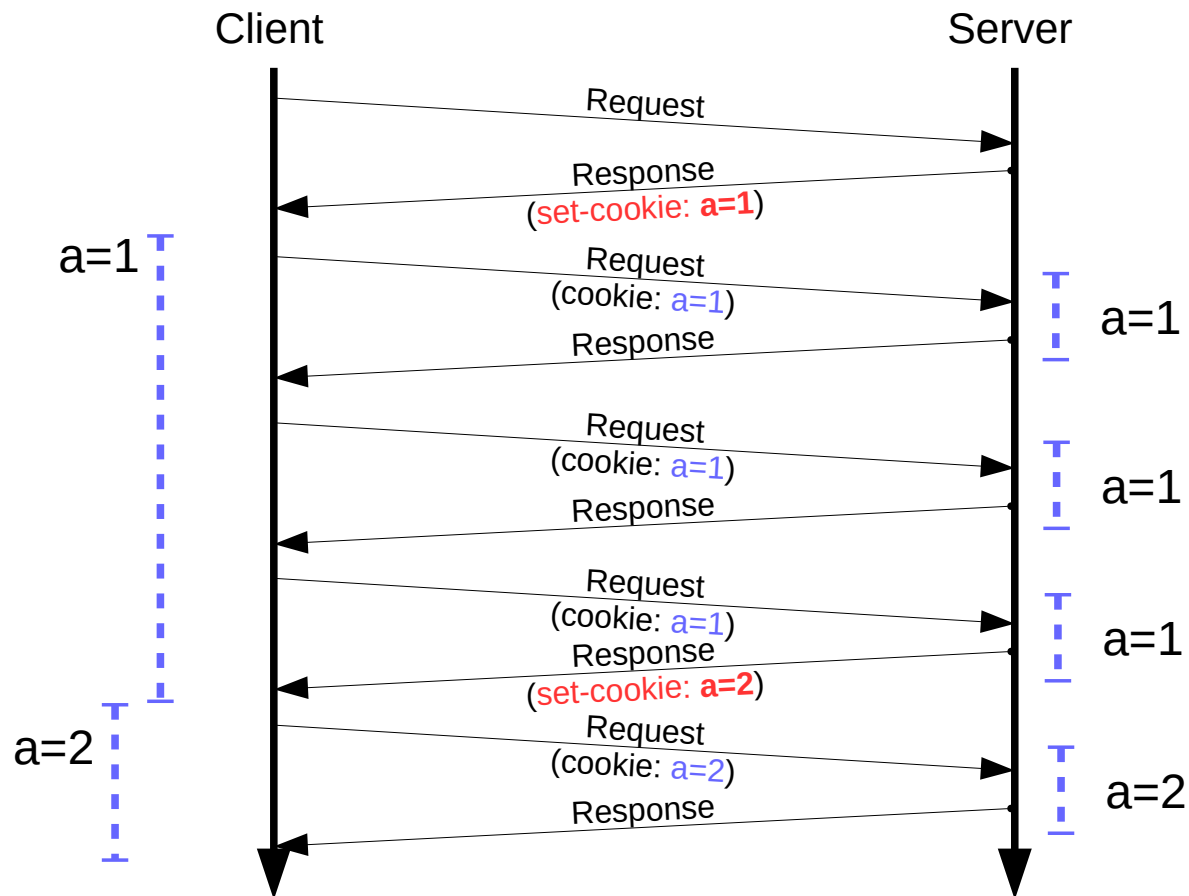
# Rückblick: Das HTTP-Protokoll – Cookies

---

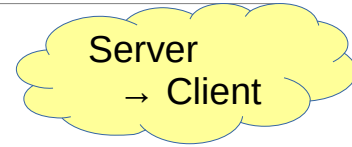
- **Cookies** sind Zeichenketten, die
  - der Server mit einem Response im Client setzen kann und
  - der Client mit jedem Request an den Server zurück überträgt.
- **Durch Cookies kann der Server der Sitzung zwischen Client und Server Attribute zuordnen**
  - Beispiel: Benutzerspezifische Einstellungen
    - „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**

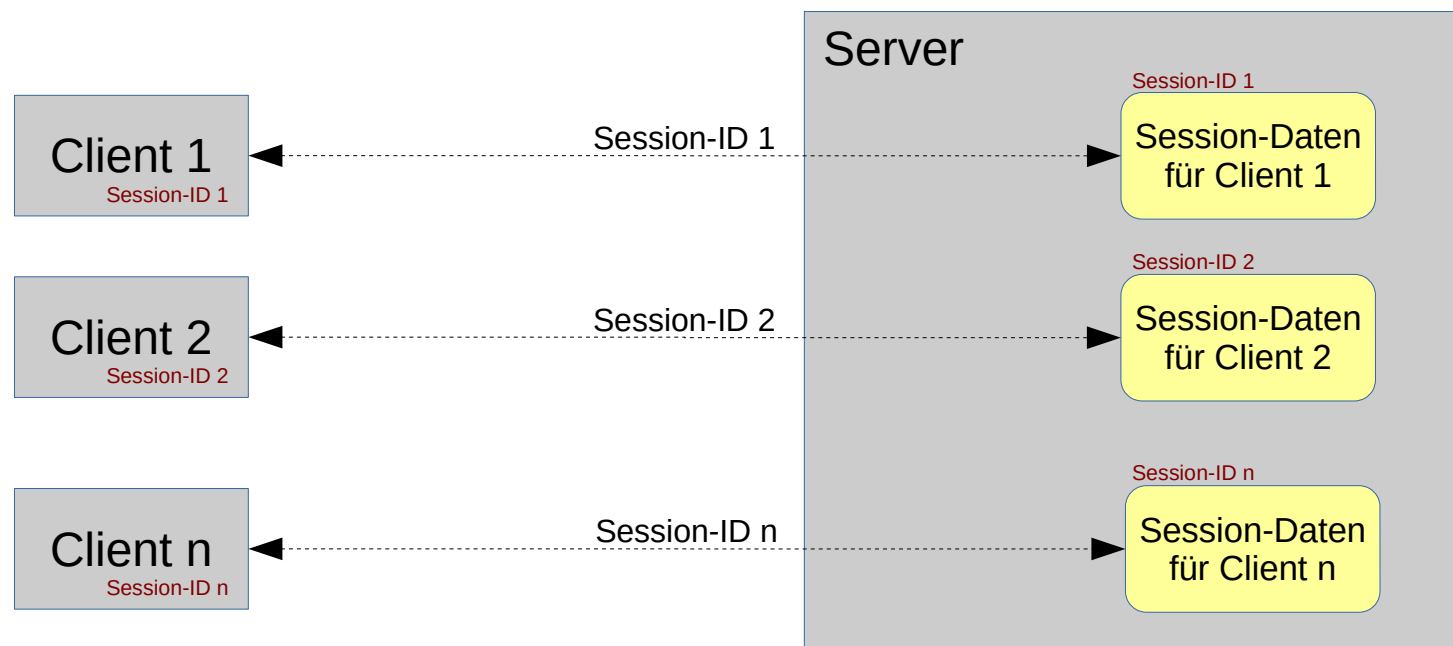
```
<?php
// Wir zählen die Zahl der Aufrufe dieses Clients in Cookies
if (!isset($_COOKIE['number_of_calls']))
 // cookie war ungesetzt → setze Startwert
 $number_of_calls = 0;
else
 $number_of_calls = $_COOKIE['number_of_calls'] + 1;
setcookie('number_of_calls', $number_of_calls);
?>

<!DOCTYPE html>
<html> <head> <!-- ... --> </head>
<body>
<p>Sie haben
 <?php echo $number_of_calls; ?>
 Aufrufe gemacht.
</body>
</html>
```

- Diesen Zähler kann der Webseitenutzer 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:

```
$mysqli = mysqli_connect($mysqli_server,
 $mysqli_user,
 $mysqli_password,
 $mysqli_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:

MatrNr	Name
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 mit 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 eine 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