Beispiel: Login-Seite

- **Verbesserung: Login flexibler realisieren**
  - Login-Test flexibler und in separate Funktion
    - ```php
      <?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
      <?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
<?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
<?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)

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
<?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

```php
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

```php
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.

Hintergrund: Hash-Funktionen

- **Hintergrund: Kryptographische Hash-Funktionen (1)**

1) Anforderung: **Falltüreigenschaft**

Wozu?

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.*

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

Wozu?

- 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)

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

- **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

Hintergrund: Hash-Funktionen

- **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
<?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

Hintergrund: Hash-Funktionen

- **Hintergrund: Kryptographische Hash-Funktionen (3)**
  - Alle (effizient) umkehrbaren Funktionen sind **keine** Hash-Funktionen
    - Beispiel: **base64-Kodierung** ist keine Hash-Funktion
    - ```php
      <?php
          $s = '1234';
          $a = base64_encode($s);
          echo "base64_encode('$s') = '$a'\n";
          unset($s);                  // original-String zur Demonstration 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“)
      ```

Hintergrund: Hash-Funktionen

- **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) … $26^{10} \approx 1{,}4 * 10^{14}$ (10 Kleinbuchstaben) Varianten
  - Angriff: Man berechnet alle Hashes voraus („**Rainbow-Table**“)
    - Eine Festplatte mit 8TB ($8 * 10^{12}$ 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
    - Ziel: Rainbow-Tables werden zu groß → nicht realisierbar (exponentiell!)
    - Siehe https://de.wikipedia.org/wiki/Salt_(Kryptologie)

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

Hintergrund: Hash-Funktionen

- **Ausblick: Sichere Passwort-Hash-Verfahren**
  - Problem: Passwörter haben oft geringe Entropie oder sind errratbar
    - Entropie ist ein Maß, wie ungeordnet ein System ist
    - Konkret: Bestimmte Passworte sind beliebt
      - Angreifer sammeln „beliebte“ Passworte und probierten diese (Wörterbuch)
    - Aber auch: Passworte sind nicht völlig gleichverteilt
      - Worte, Silben oder Zifferngruppen kommen oft vor
      - Häufig Muster in Passworten (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

Hintergrund: Hash-Funktionen

- **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

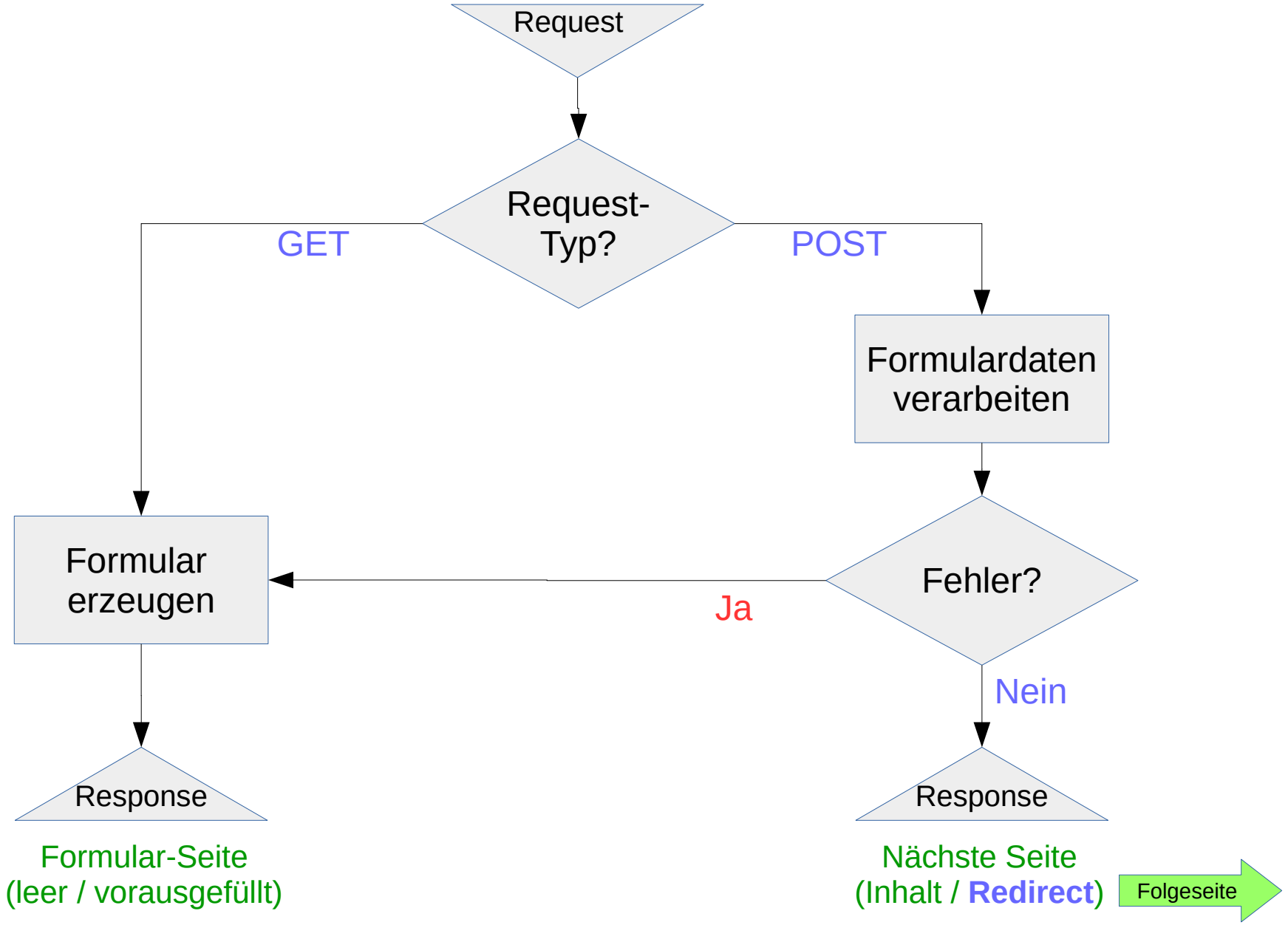
- **Vorteil:**
  - Kompakt, alle Formularaspekte in einer Datei behandelt

- **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


Request
Request-Typ?
GET
POST
Formulardaten verarbeiten
Formular erzeugen
Fehler?
Ja
Nein
Response
Response
Formular-Seite
(leer / vorausgefüllt)
Nächste Seite
(Inhalt / Redirect)
Folgeseite

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
<?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) >= (Anzahl der nachgefragten)

  - Antworten können oft auch aus (Client- / Server-) **Cache** geliefert werden
    - → (Anzahl der durchgeführten Requests) <= (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 (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`
  - **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 → $16^{32} \approx 3*10^{38}$ 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

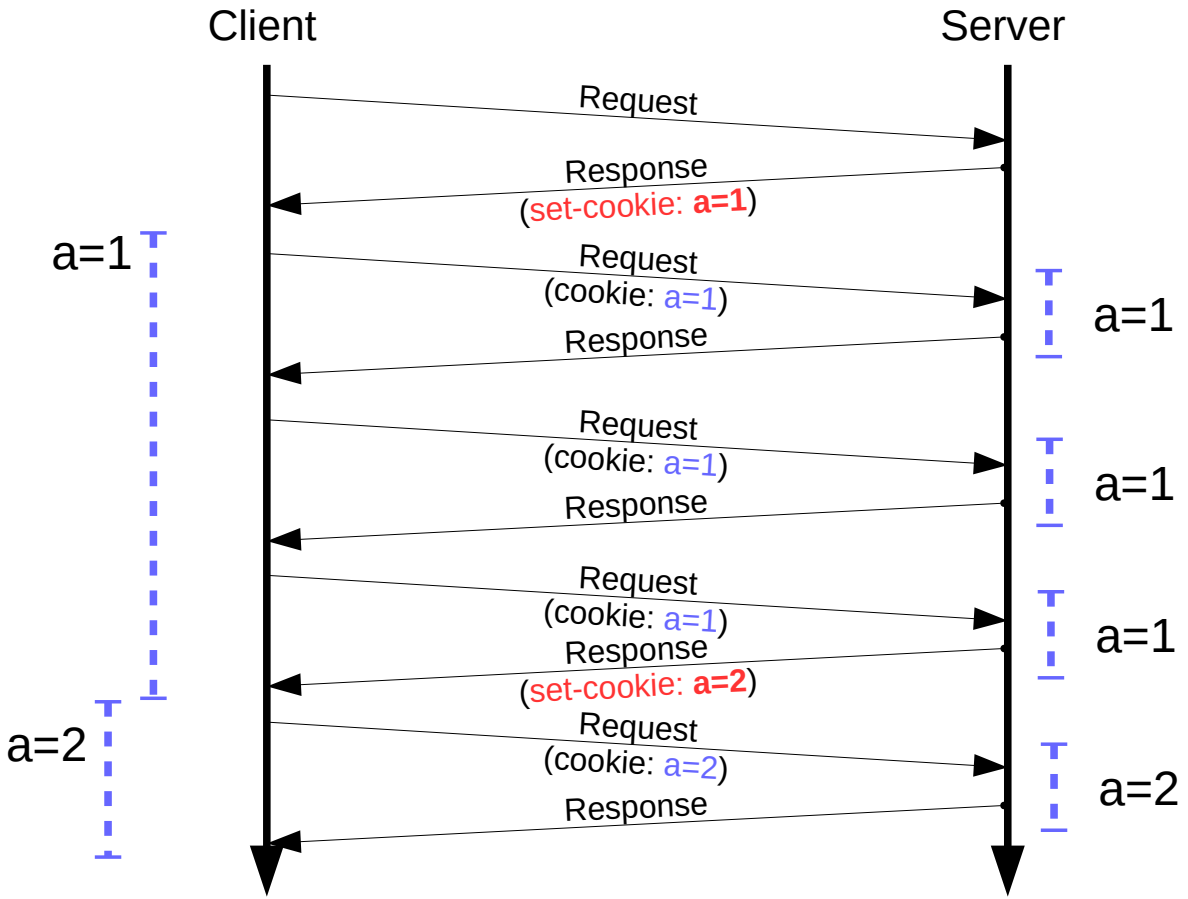
- **URL-Zerlegung – Realisierung in Apache + PHP**
  - In **PHP** *(leider etwas technisch – nur Grundidee relevant)*
    - **Ziel**: Wir müssen den URL-Pfad zerlegen und die **Parameter extrahieren**
    - Der URL-Pfad ist in $_SERVER['REQUEST_URI'] enthalten
      - Genauer ist es (siehe gemäß RFC 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 Paramter 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 Reqest 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)
    - ```
      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
<?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

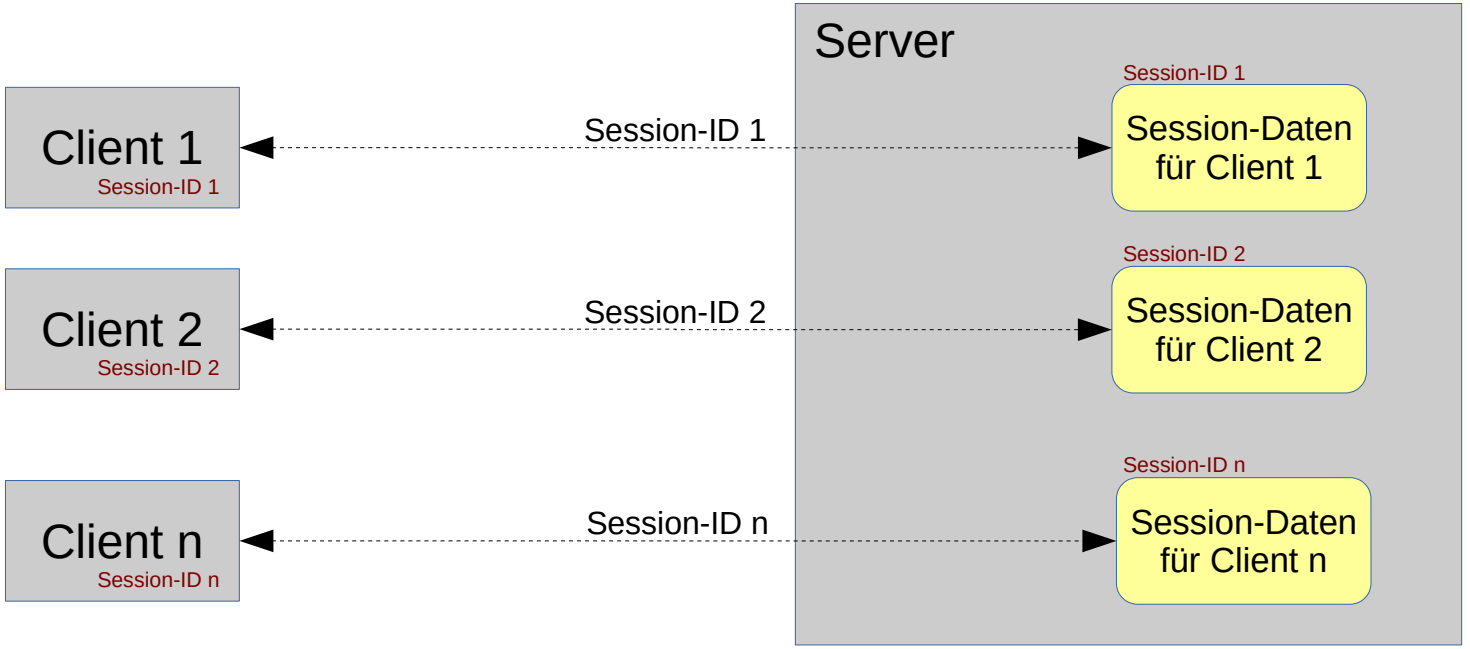
- **Beispiel: Anzahl der Seiten-Aufrufe zählen**

```php
<?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)
    - 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
<?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 Authentifzierungs-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
<?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?**)