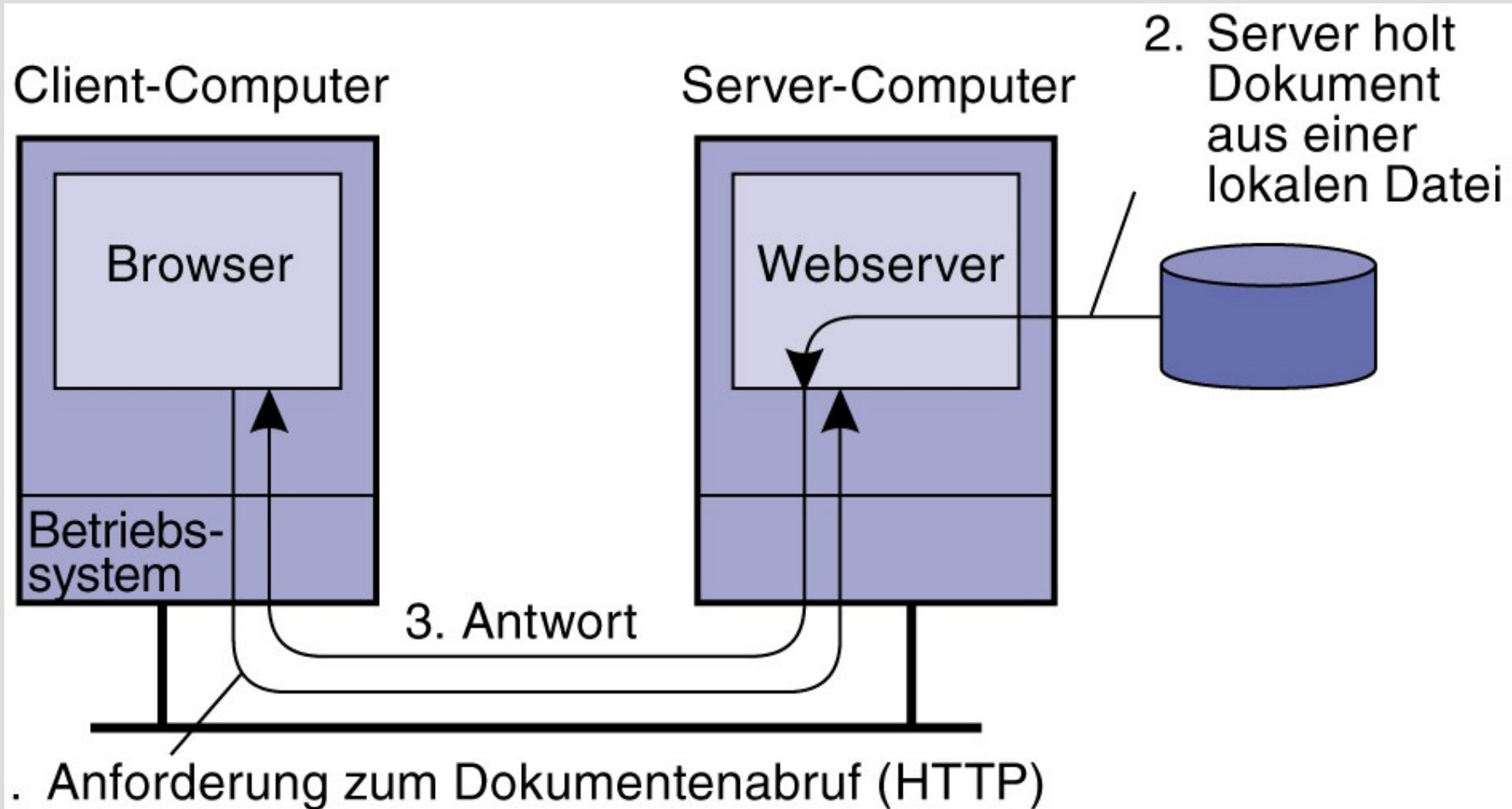


# Datenbank - API

Und wie nun zum Browser verbinden?

# Website



Bsp: GET /foo.html HTTP/1.0

# Website

| Layer | Protokoll |
|-------|-----------|
| 7     | HTTP      |
| 4     | TCP       |
| 3     | IP        |
| 1/2   | ETHERNET  |

GET /foo.html HTTP/1.1  
Host: foo.com

. Anforderung zum Dokumentenabruf (HTTP)

# Website

| Layer | Protokoll |
|-------|-----------|
| 7     | HTTP      |
| 4     | TCP       |
| 3     | IP        |
| 1/2   | ETHERNET  |

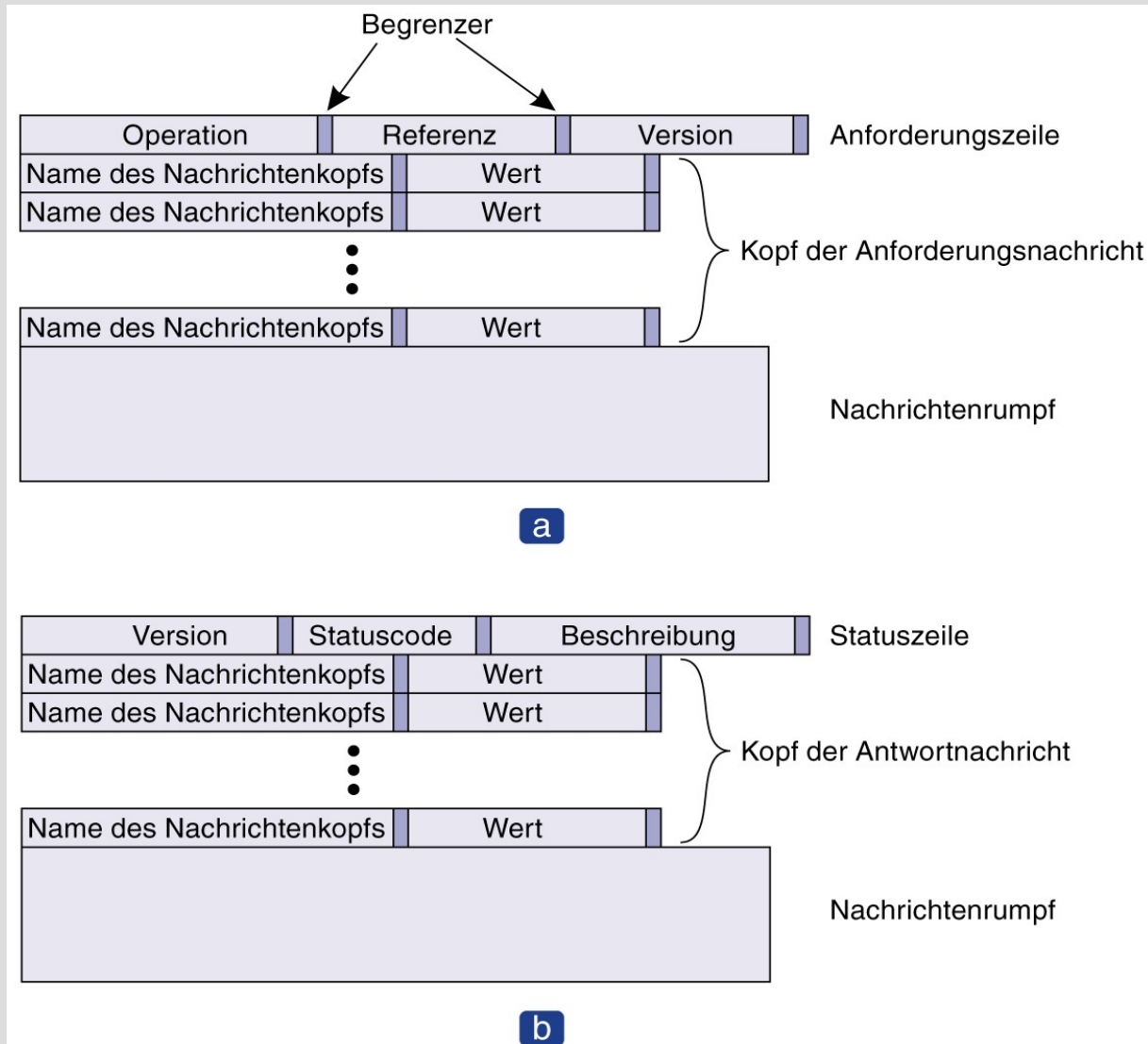
HTTP/1.1 200 OK  
Date: 09 Aug 2020 05:44:54 GMT  
Server: Apache/2.4.10 (Debian)  
Content-Length: 429  
**Content-Type: text/html**

```
<!doctype html>  
<html>..</html>
```

. Anforderung zum Dokumentenabruf (HTTP)

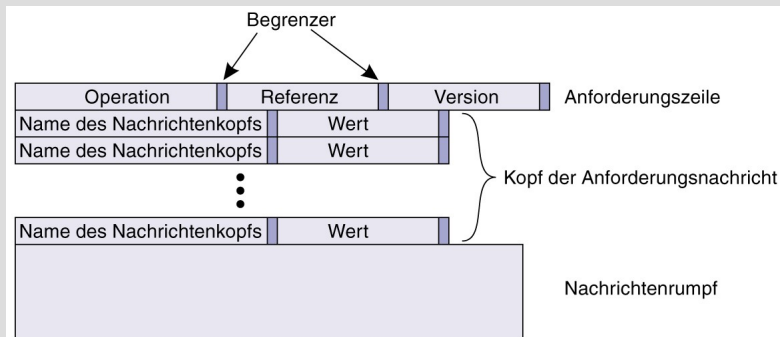
# HTTP-Protokoll

## Request(a) vs Rssponse(b)

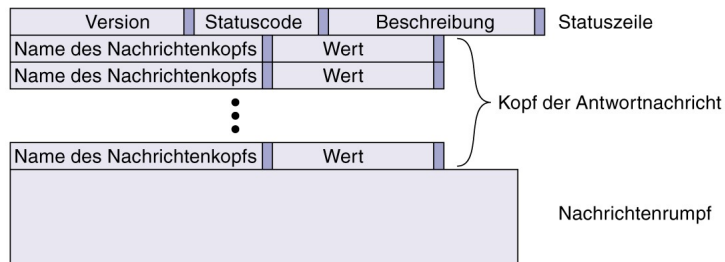


# HTTP-Protokoll

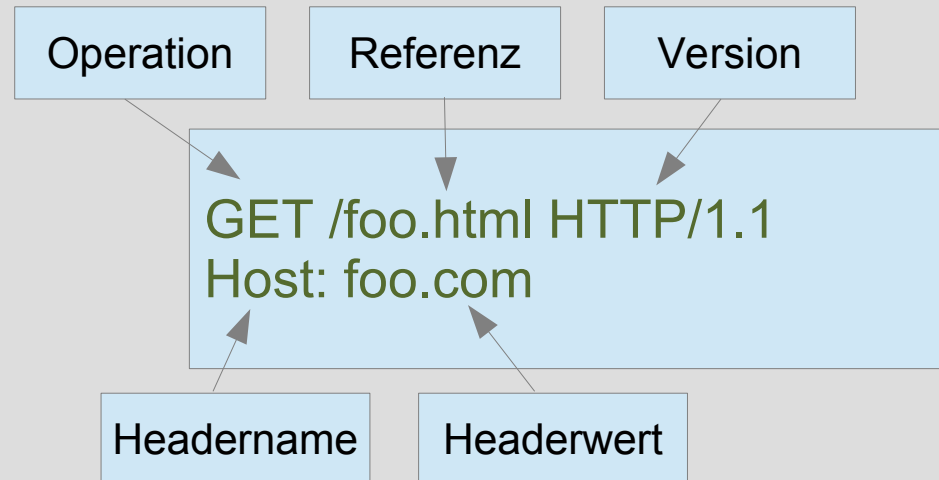
## Request vs Response



a



b

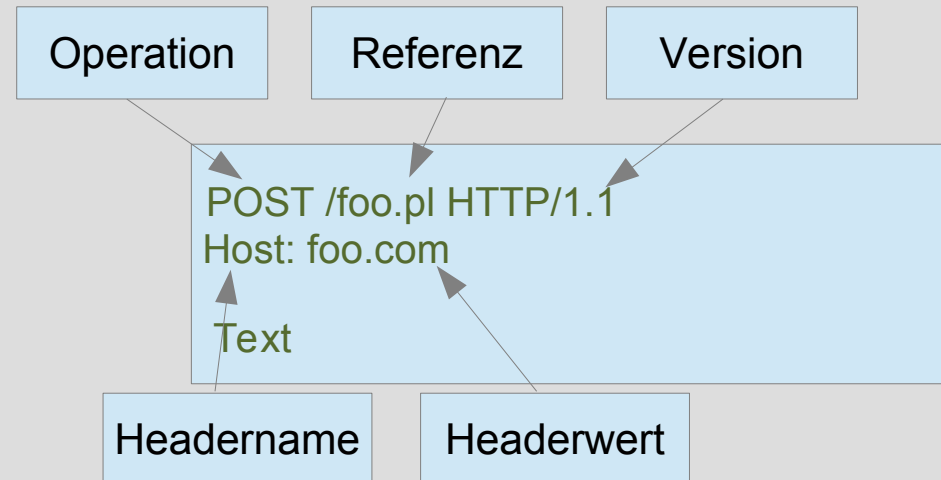
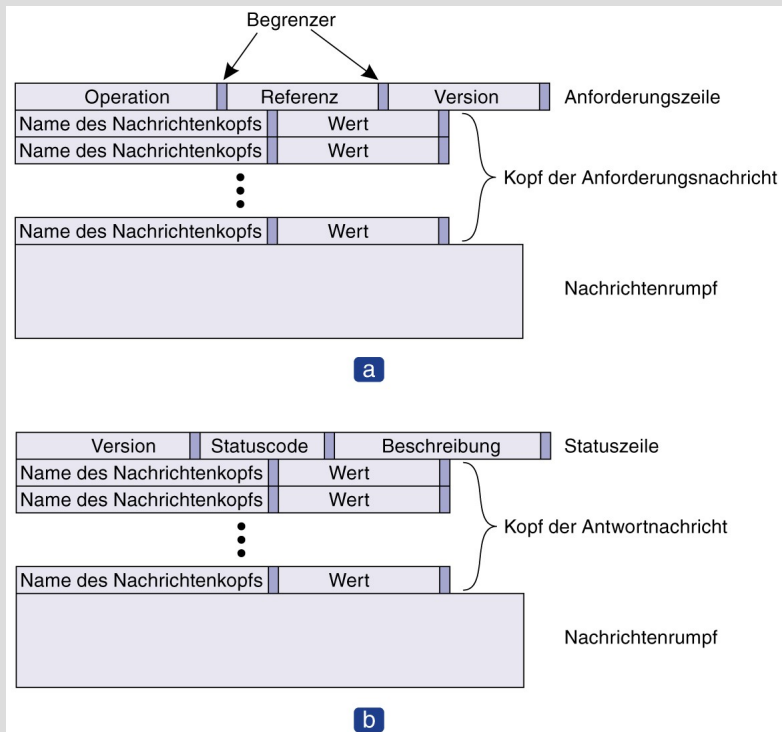


**HTTP/1.1 200 OK**  
**Server: Apache/2.4.10**  
**Content-Length: 429**  
**Content-Type: text/html**

**<!doctype html>**  
**<html>..</html>**

# HTTP-Protokoll

## Request vs Response



# HTTP

Das war der „Trivialfall“:

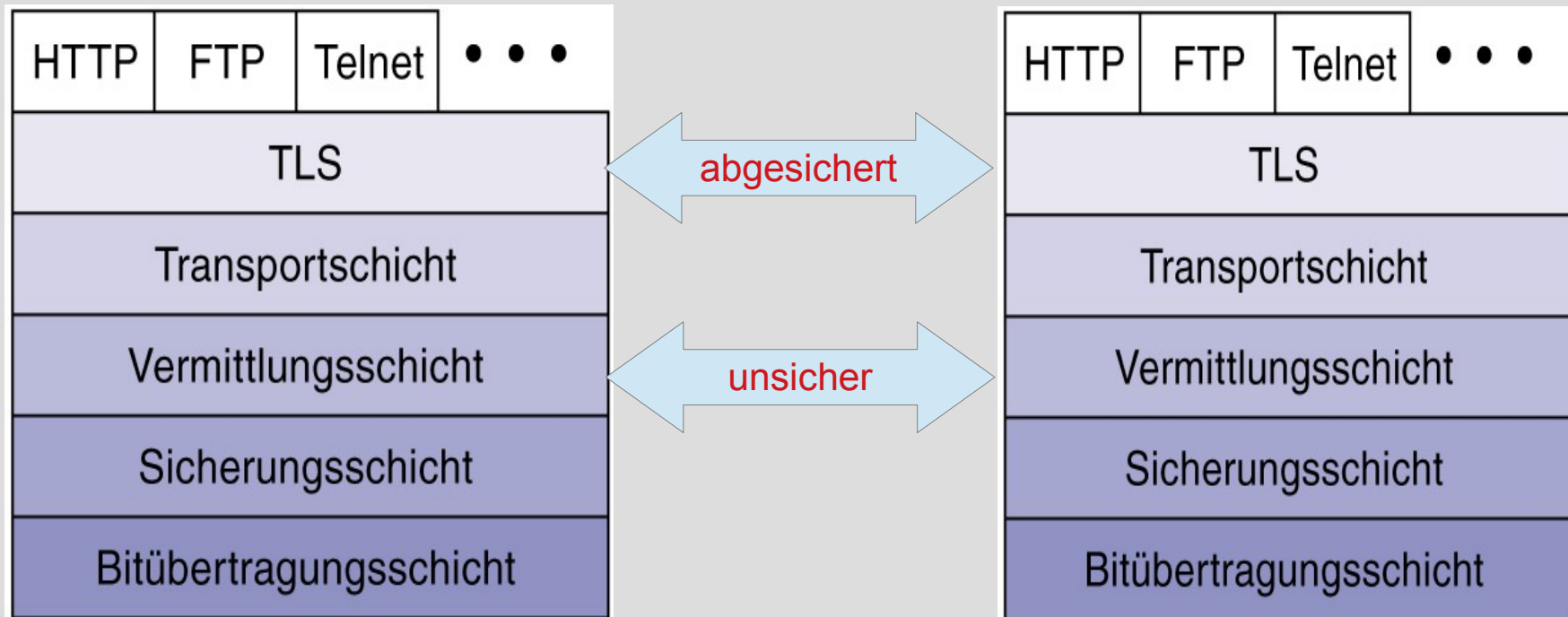
- Statische Dokumente abrufen
- Hochladen ginge auch mittels PUT

Interessanter: Dynamisches HTTP ergo Einbinden von Programmen, und damit unseren Skripten mit Datenbank-API.

HTTP wird zum Transportprotokoll jenseits statischer Webseiten.

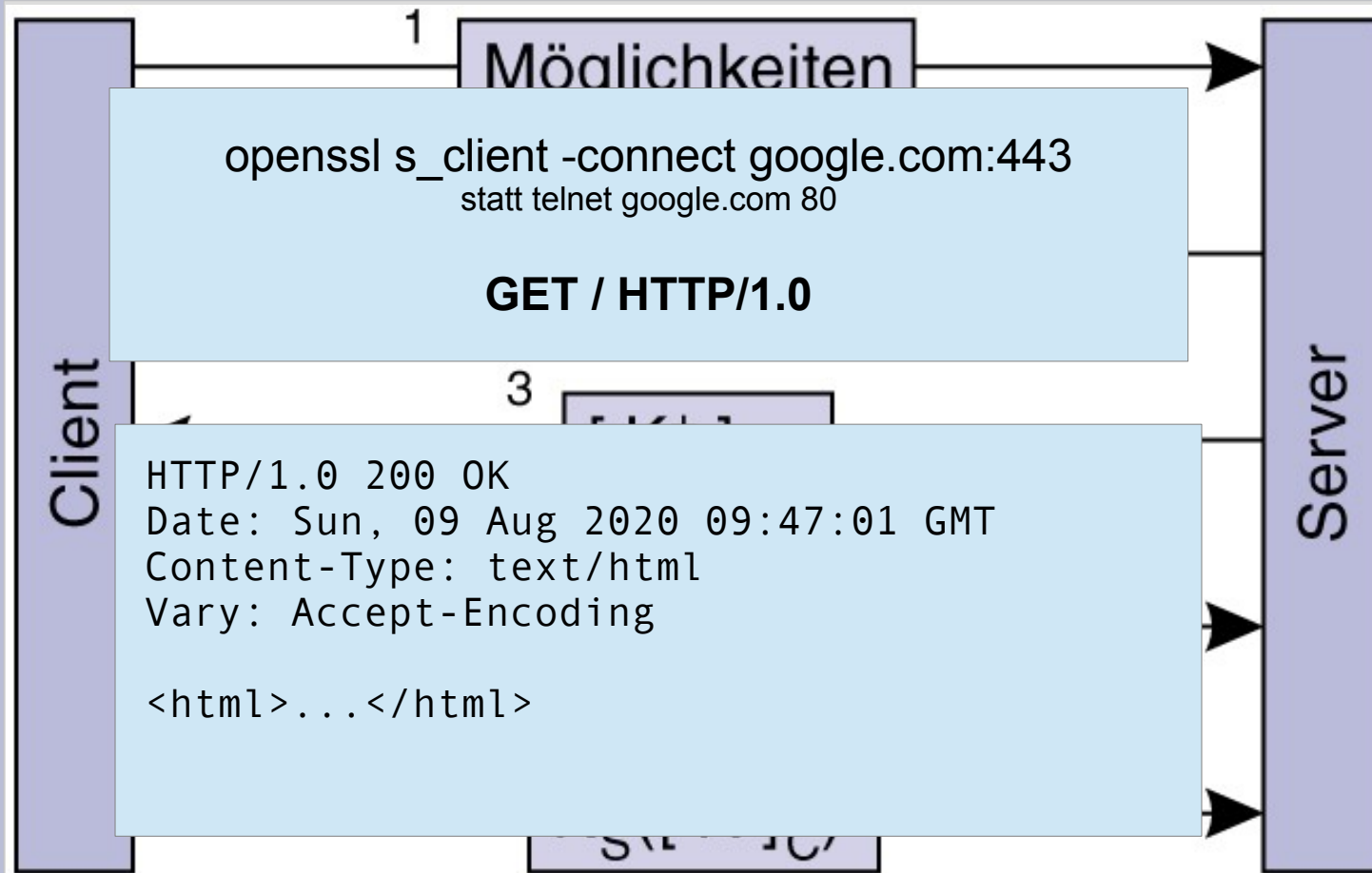


# Extra: Sicherheit mit TLS



# TLS Authentifizierung

## “Debugging“



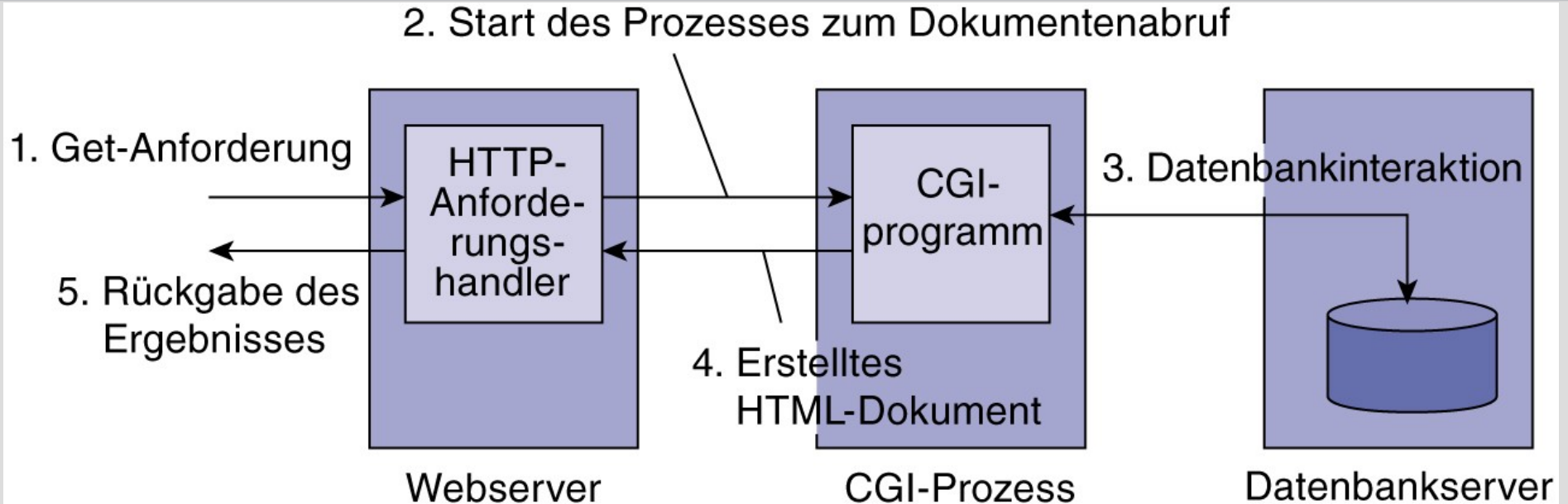
# HTTP

## Zwischenfazit

Wir haben mit HTTP / HTTPS ein Transportprotokoll, um Webapplikationen zu realisieren.

Schwerpunkt hier: Webapplikationen mit Datenbank-API

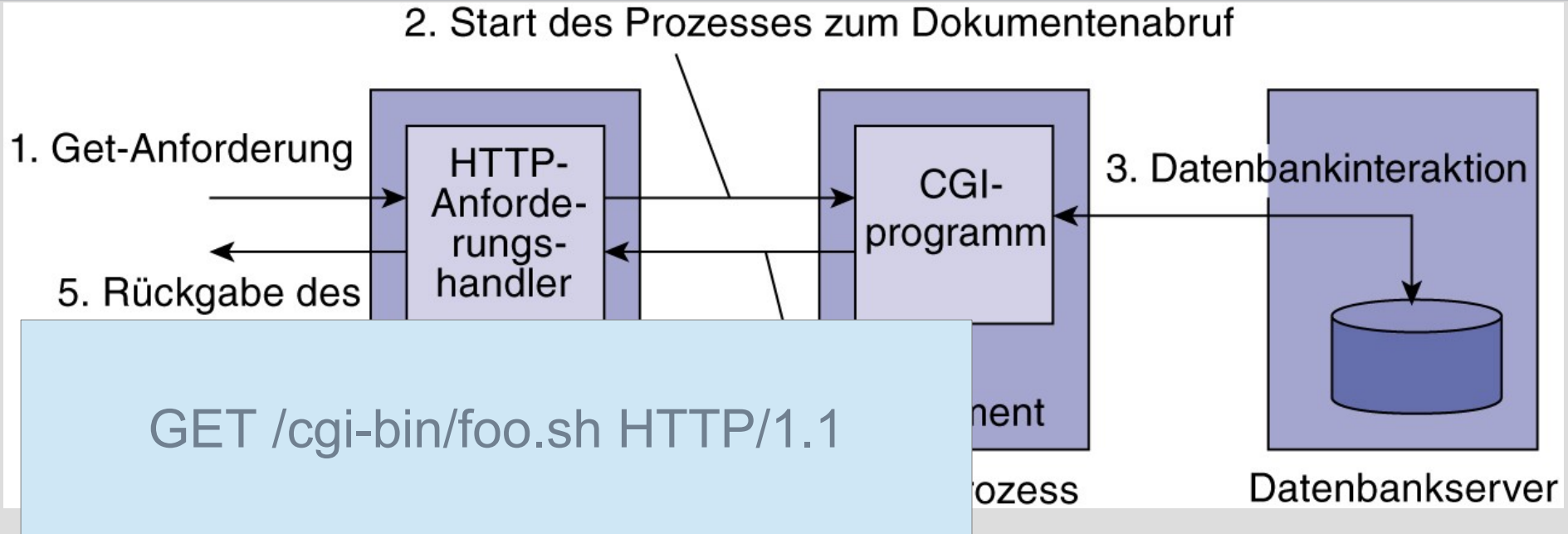
# CGI-Programme



```
GET /cgi-bin/foo.sh HTTP/1.1
```

# BASICS

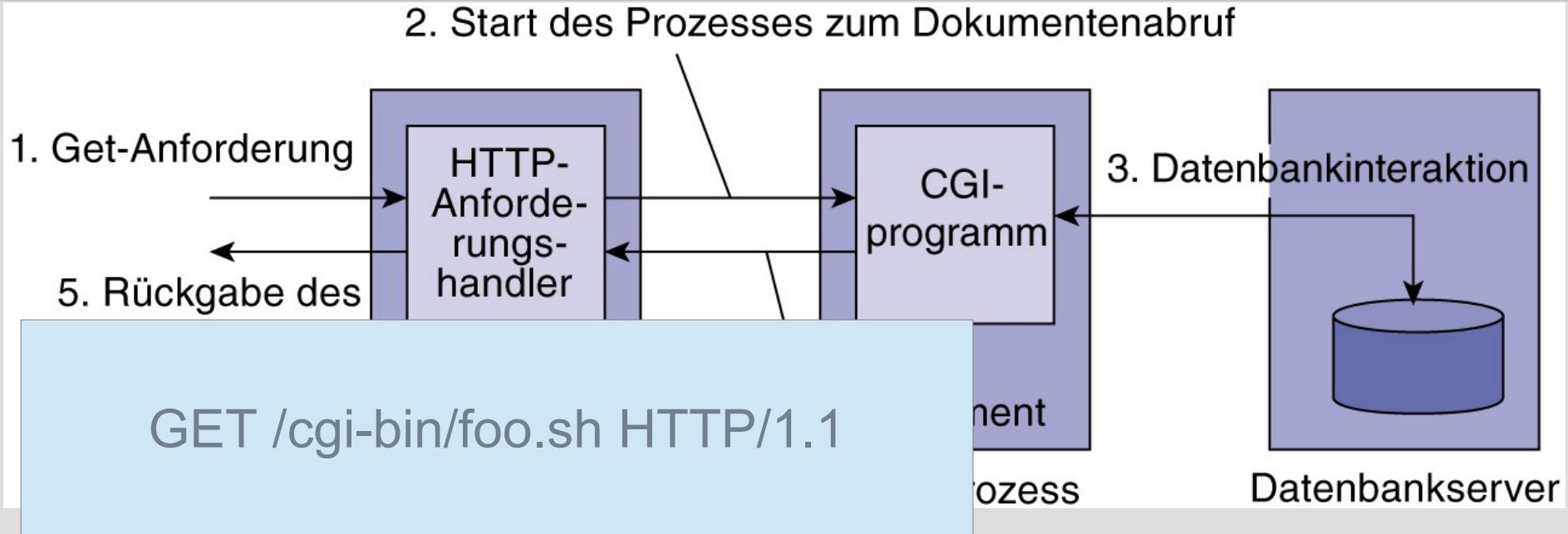
## CGI-Programme



foo.sh:

```
#!/bin/sh
echo content-type: text/html
echo
echo FOO
```

# CGI-Programme



foo.sh:

```
#!/bin/sh
echo content-type: text/html
echo
echo FOO
```

Stdout wird umgeleitet  
und im Browser angezeigt.

# CGI

Standard CGI ist unsicher und langsam

- besser Module / FastCGI
- Aber es zeigt das Prinzip aller Frameworks
  - Code aufrufen (mit Argumenten)
  - Ergebnis entgegennehmen

# CGI

Stdout zum Webbrowser umleiten? So geht das:

Standard-CGI kann man mit ein paar Zeilen C realisieren

```
. . .
newsock = accept(sock, . . .);
f = fdopen(newsock, "r+"); //Der Kanal zum Browser

p = popen("./foo.sh", "r");
fgets(l2, 255, p);
pclose(p);

fprintf(f, "HTTP 200 OK\ncontent-type: text/html\n\n%s", l2);
```



# CGI

## 1 stdout zum Webbrowser umleiten? So geht das:

Standard-CGI kann man mit ein paar Zeilen C realisieren

```
. . .  
p = popen("./foo.sh", "r");  
fgets(l2, 255, p);  
pclose(p);
```

foo.sh könnte das CGI-Skript sein.

Und die folgenden Codezeilen sind ein Mini-Webserver. Sie öffnen das Skript, führen es aus und leiten seine Ausgaben auf eine TCP-Socket weiter.

```
newsock = accept(sock, . . .);  
f = fdopen(newsock, "r+"); //Der Kanal zum Browser  
fprintf(f, "HTTP 200 OK\ncontent-type: text/html\n\n%s", l2);
```

# CGI

## Argumente übertragen: Basics



23 42  
ok

```
<form method="post" action="http://fert.de:42235">  
<input type="text" name="x" value="23">  
<input type="text" name="y" value="42">  
<input type="submit" value="ok"  
</form>
```

HTML

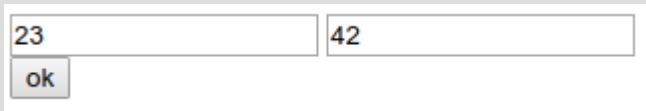
```
POST / HTTP/1.1  
Host: xxx.tld  
Content-Length: 9  
Content-Type: application/x-www-urlencoded
```

HTTP

```
x=23&y=42
```

# CGI

## Argumente übertragen



23 42  
ok

```
<form method="post" action="http://fert.de:42235">  
<input type="text" name="x" value="23"  
<input type="text" name="y" value="42">  
<input type="submit" value="ok  
</form>
```

HTML

```
POST / HTTP/1.1  
Host: xxx.tld  
Content-Length: 9  
Content-Type: application/x-www-urlencoded
```

HTTP

x=23&y=42

Server-Code

```
newsock = accept(sock, ...);  
f = fdopen(newsock, "r+"); //Der Kanal zum Browser  
// read header  
// read cgi content  
// parse cgi content x=23&y=42
```

# Datenbank - API: CGI - In/Out mit PHP

```
<?php
$servername      = "dbserv.ba-nitsch.de";
$username        = "uebung";
$password        = "ue_42235";
$dbname          = "db_uebung";

$column = $_POST["column"];
$sql = "select $column from test";

$conn = new mysqli($servername, $username, $password, $dbname);

$sql = "select $column from test";

$result = $conn->query($sql);

...

?>
```

# Datenbank - API

Das funktioniert im Prinzip,  
Aber...

**Gefahr von SQL-Injections**

# Datenbank - API: CGI - In/Out mit PHP

```
<?php
$servername      = "dbserv.ba-nitsch.de";
$username        = "uebung";
$password        = "ue_42235";
$dbname          = "db_uebung";

$column = $_POST["column"];
$sql = "select $column from test";

$conn = new mysqli($servername, $username, $password, $dbname);

$sql = "select $column from test";

$result = $conn->query($sql);

...

?>
```

Bei Nutzereingabe \$column="name" ist alles ok

# Datenbank - API Gefahren

dbserv.ba-nitsch.de/mariadb3.php - Chromium

dbserv.ba-nitsch.de/mari x +

Not secure | dbserv.ba-nitsch.de/mariadb3.php

## Maria-DB

name

clear

submit

SQL - Template: select **\$column** from test

Generated SQL : select **name** from test

| name    |
|---------|
| Kossäth |
| Puder   |
| Sobo    |

HTML-Formular mit Eingabefeld für den Spaltennamen,  
hier wurde „name“ eingegeben.

# Datenbank - API

## Gefahren

```
<?php
$servername      = "dbserv.ba-nitsch.de";
$username        = "uebung";
$password        = "ue_42235";
$dbname          = "db_uebung";

$column = $_POST["column"];
$sql = "select $column from test";

$conn = new mysqli($servername, $username, $password, $dbname);

$sql = "select $column from test";

$result = $conn->query($sql);

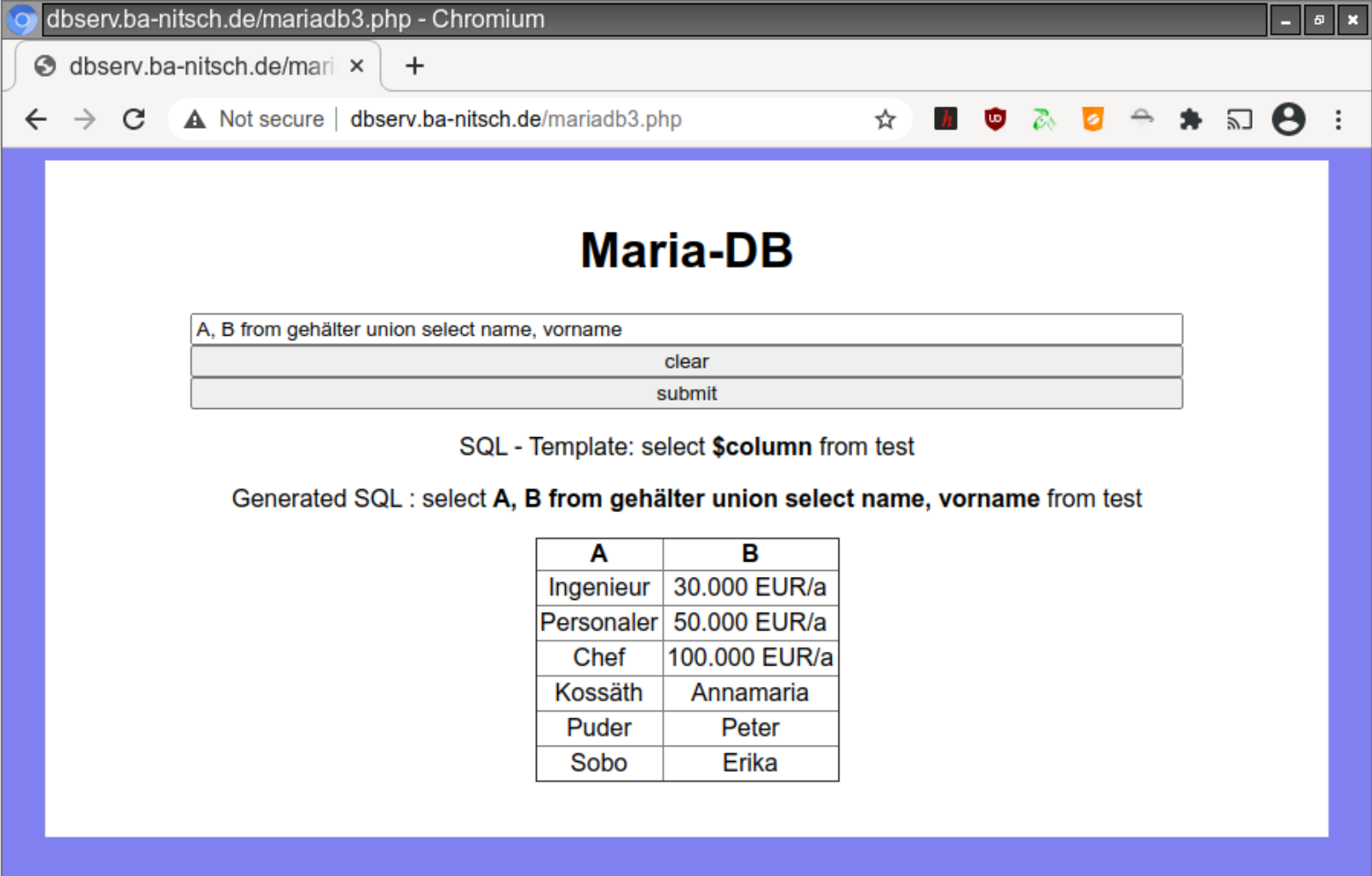
...

?>
```

Bei Nutzereingabe  
\$column="A, B from gehälter union select name, vorname"  
ist nichts mehr ok



# Datenbank - API Gefahren



dbserv.ba-nitsch.de/mariadb3.php - Chromium

dbserv.ba-nitsch.de/mari x +

Not secure | dbserv.ba-nitsch.de/mariadb3.php

## Maria-DB

A, B from gehälter union select name, vorname

clear

submit

SQL - Template: select **\$column** from test

Generated SQL : select **A, B** from **gehälter** union select **name, vorname** from test

| A          | B             |
|------------|---------------|
| Ingenieur  | 30.000 EUR/a  |
| Personaler | 50.000 EUR/a  |
| Chef       | 100.000 EUR/a |
| Kossäth    | Annamaria     |
| Puder      | Peter         |
| Sobo       | Erika         |

# Datenbank - API Gefahren

```
<?php  
  
$spalte = $_POST["column"];  
  
...  
  
$sql = "select ` $column ` from test";  
  
...  
?>
```

## Gegenmaßnahme1: Backquotes

Nun ergibt sich als SQL-String:

`select `A, B from gehälter union select name, vorname` from test`

Das liefert zumindest einen SQL-Fehler und wird nicht ausgeführt.

# Datenbank - API

## Gefahren

```
<?php

$spalte = $mysqli -> real_escape_string($_POST['column']);

...

$sql = "select ` $column ` from test";

...

?>
```

**Gegenmaßnahme2: SQL-Escapes**

# Datenbank - API

## Gefahren

„SQL-Escapes“ ersetzen alle „gefährlichen“ Eingaben, die als SQL-Befehle missbraucht werden könnten.

# Datenbank - API

## Gefahren

```
<?php

$column = $_POST['column'];

$stmt = $dbh->prepare("select :column from test;
$stmt->bindParam(':column', $column);
$stmt->execute();

?>
```

**Gegenmaßnahme 3:**  
**Prepared Statements mit Platzhaltern**

# Datenbank - API

Zwischenfazit:

Wir wissen nun, wie CGI im Prinzip funktioniert und worauf wir beim Prüfen der Argumente achten müssen.

Bleibt noch: Seitenaktualisierung ohne Reload-Button

# Webdienste ohne Nachladen der Seiten (synchrones AJAX)

//Der Browser ist der Client und die Clientsprache ist Javascript

```
function foo() {  
    var service = new XMLHttpRequest();  
    service.open(„GET“, „/cgi-bin/mathworker?fc=add&x=1&y=2“, false);  
    service.send(null);  
    var text = service.responseText;  
    document.getElementById(“res“).innerHTML = text;  
}
```

```
<body>  
<div id=“res“></div>  
</body>
```

# Webdienste ohne Nachladen der Seiten (asynchrones Ajax)

```
function foo() {  
    var service = new XMLHttpRequest();  
    service.open(„GET“, „/cgi-bin/mathworker?fc=add&x=1&y=2“, true);  
    service.onreadystatechange = foo_done;  
    service.send(null);  
}
```

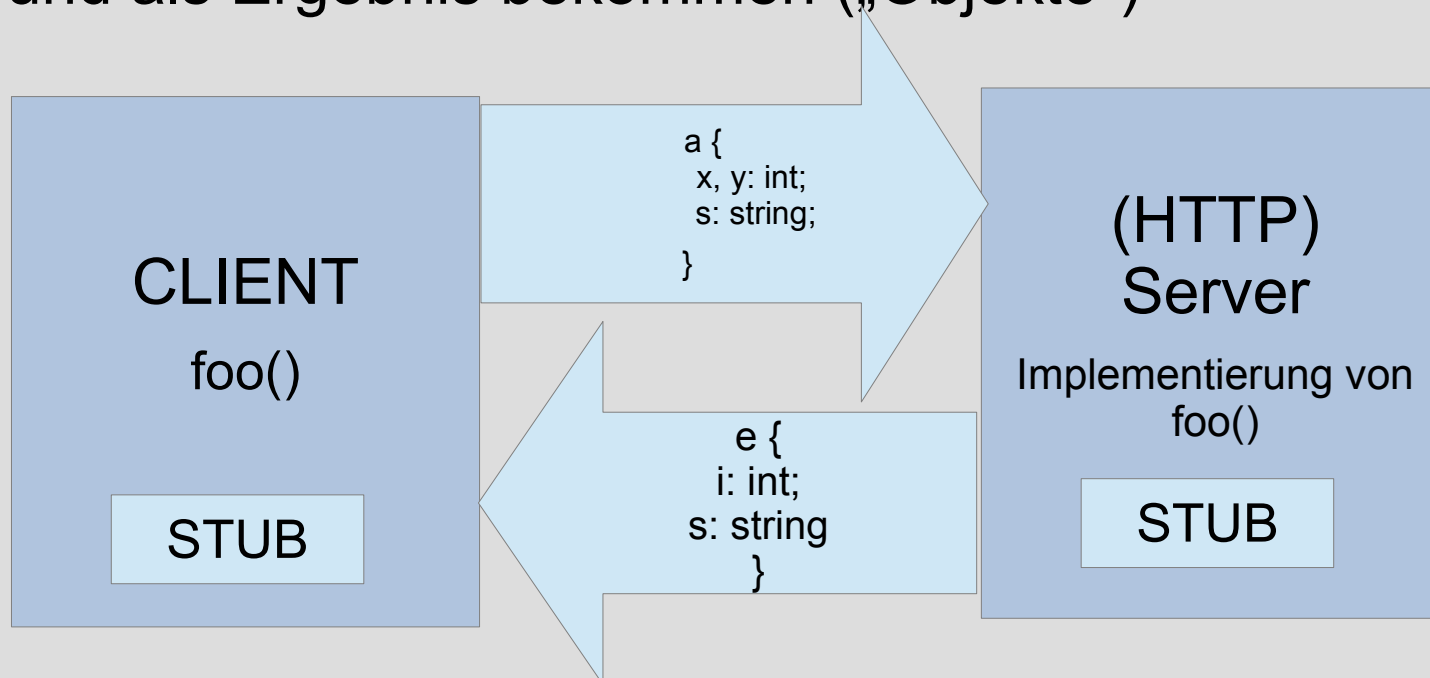
```
function foo_done() {  
    ...  
    var text = service.responseText;  
    document.getElementById(„res“).innerHTML = text;  
}
```

```
<body>  
<div id=„res“></div>  
</body>
```



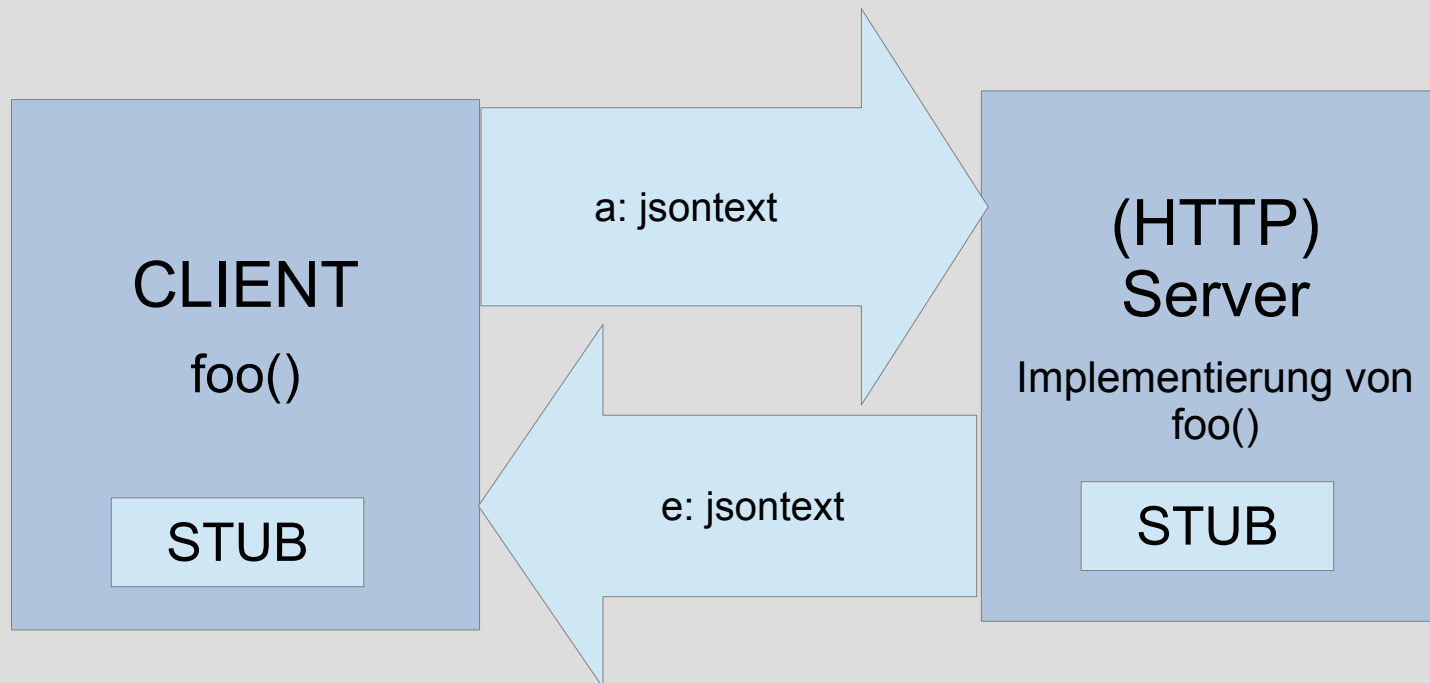
# Webdienste und AJAX

- Schon viel erreicht: RPC Calls im Browser inkl. Refresh
- allerdings haben wir hier immer noch einfache Datentypen (Strings).
- Oft will man aber komplexe Daten als Argumente übergeben und als Ergebnis bekommen („Objekte“)



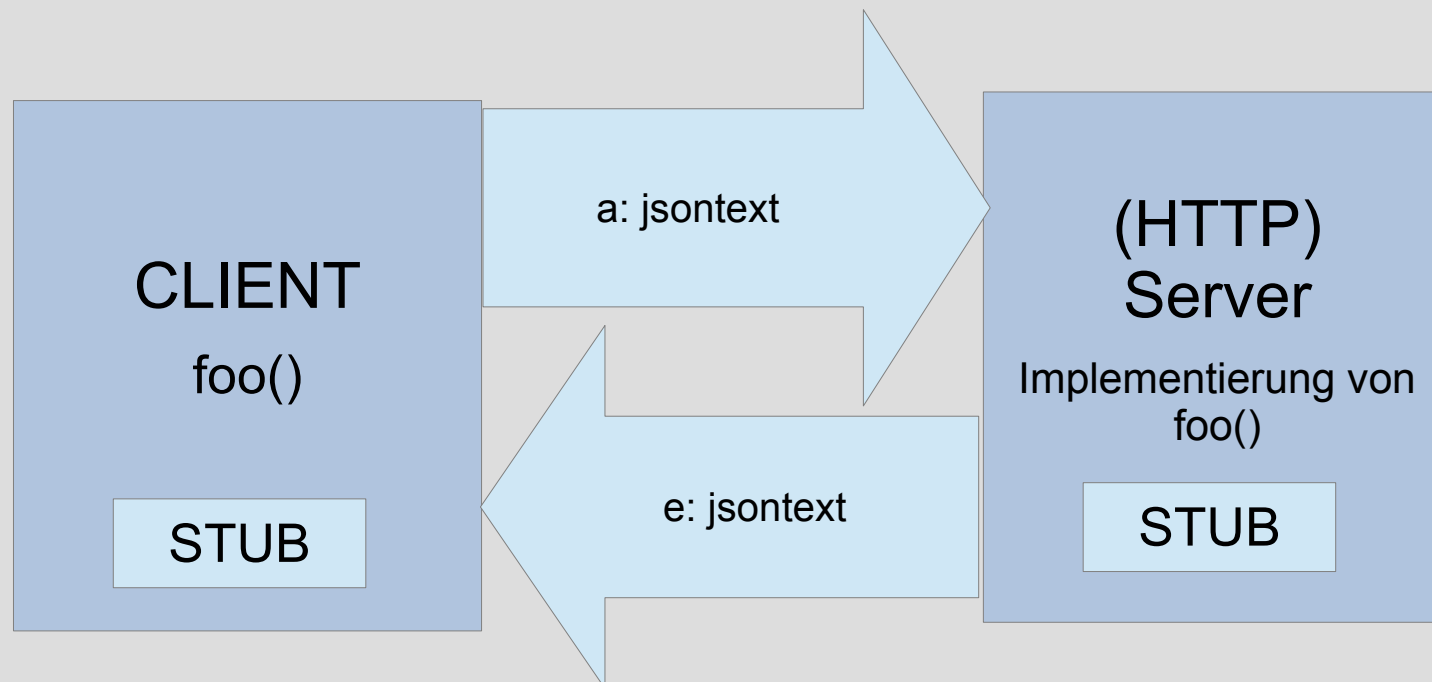
# Webdienste und AJAX

→ Um Objekte zu übertragen, muss man sie serialisieren.  
Das geht z.B. mit JSON oder auch XML



# Webdienste und AJAX

- Mittels JSON / XML kann man auch viel komfortabler Funktionen aufrufen
- XML-RPC, Soap usw: Erste komfortable Frameworks



# Zusammenfassung

- (1) HTTP als Transportprotokoll inkl TLS als Sicherheitsschicht
- (2) CGI als Grundlage für Web-RPC
- (3) Mechanismen für Parameterübergabe, Sicherheit
- (4) Objektrealisierung mit JSON / XML
- (5) Ausblick: High-Level-Frameworks: XML-RPC