

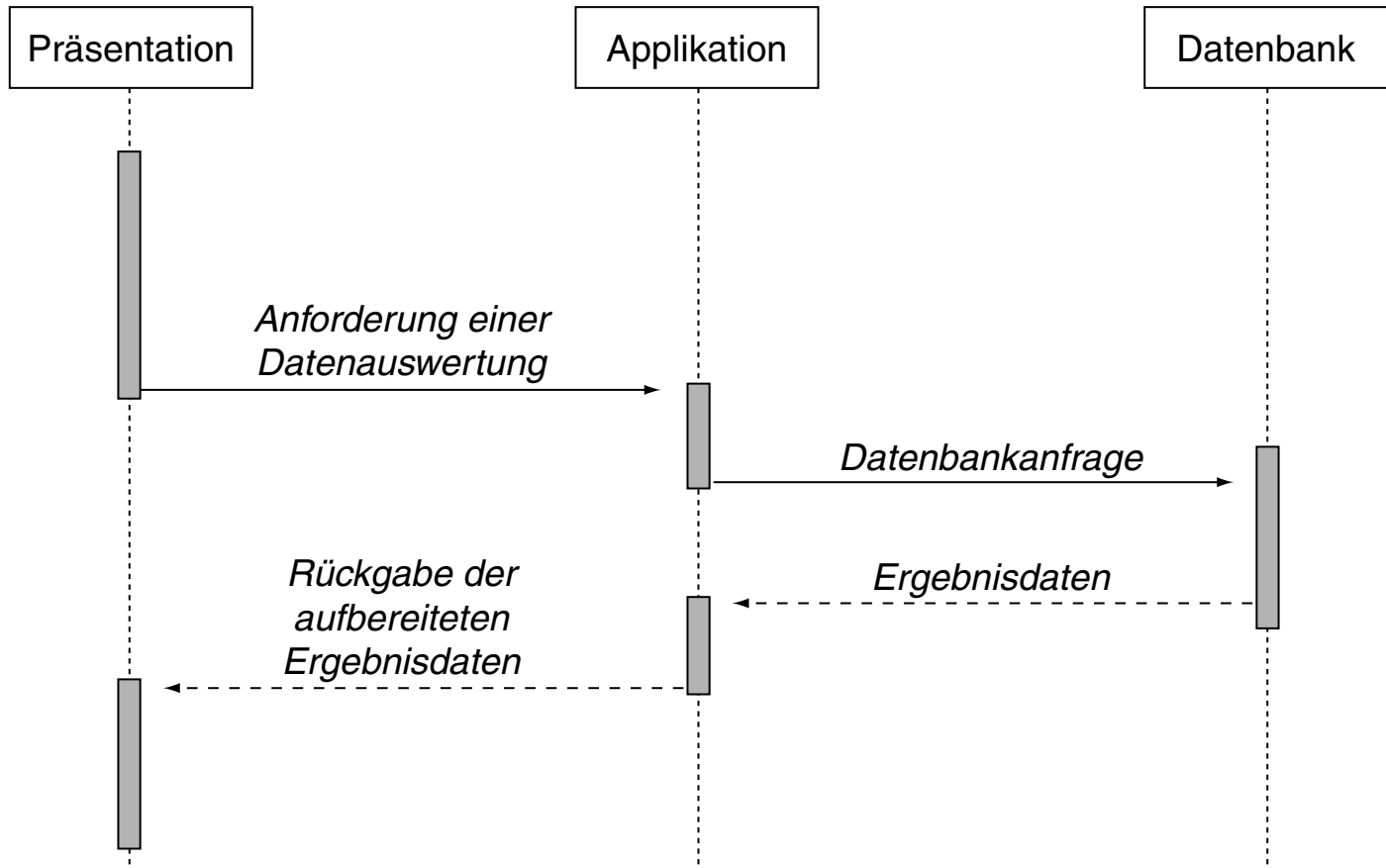
Kapitel WT:VI

VI. Architekturen und Middleware

- ❑ Client-Server-Architekturen
- ❑ Ajax
- ❑ REST
- ❑ WebSockets
- ❑ Remote Procedure Call RPC
- ❑ Message-oriented Middleware

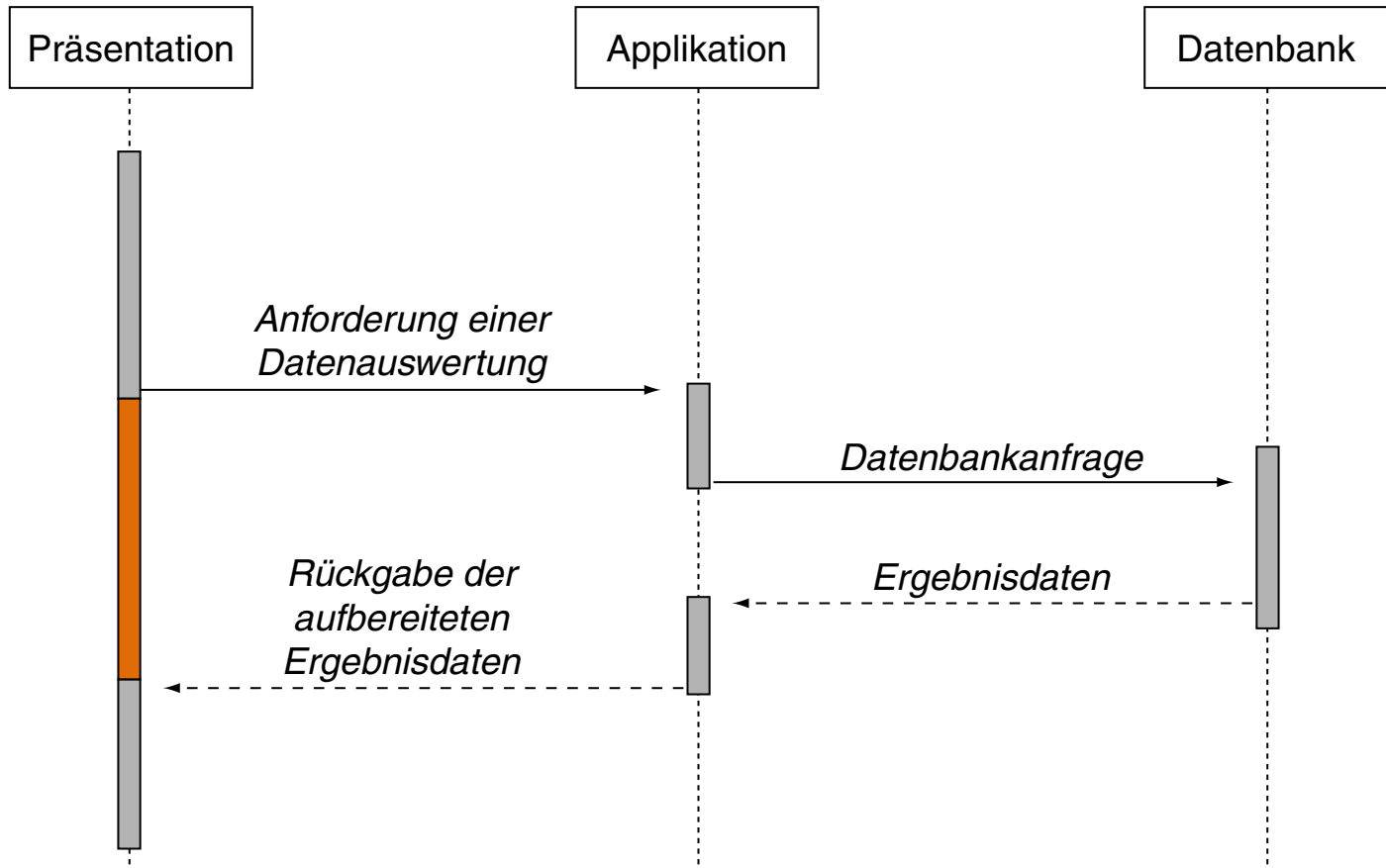
Client-Server-Architekturen

3-Tier Architektur: Sequenzdiagramm



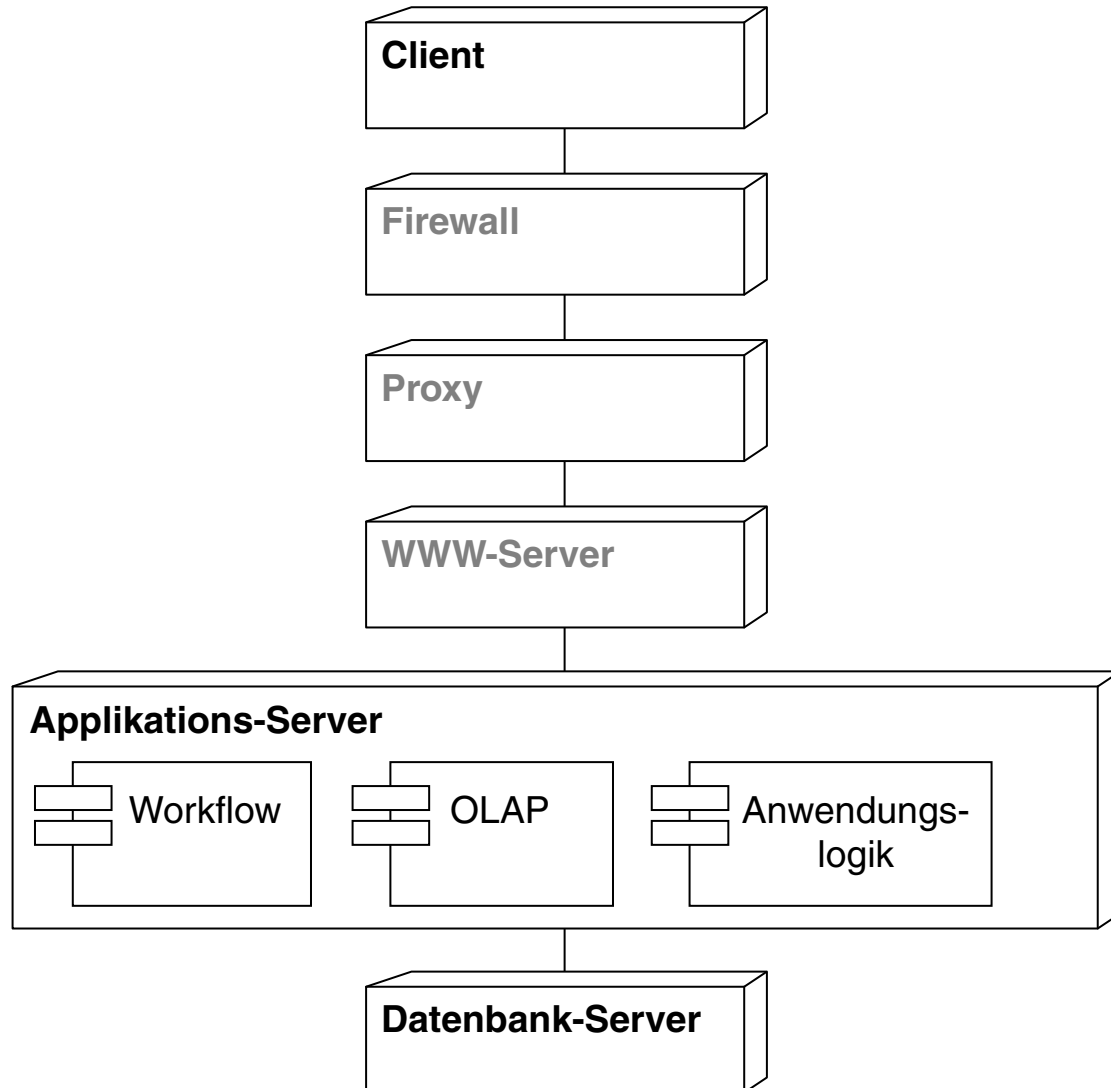
Client-Server-Architekturen

3-Tier Architektur: Sequenzdiagramm



Client-Server-Architekturen

3-Tier Architektur: Deployment-Diagramm



Client-Server-Architekturen

Architekturmuster für Web-Anwendungen

Architekturmuster

Applikationslogik
(Model + Controller)

Präsentation
(View)

Zero Client

herstellerspezifischer
Applikationsserver

herstellerspezifischer Viewer /
eingeschränkter Browser

Client-Server-Architekturen

Architekturmuster für Web-Anwendungen (Fortsetzung)

Architekturmuster

Applikationslogik
(Model + Controller)

Präsentation
(View)

Zero Client

herstellerspezifischer
Applikationsserver

herstellerspezifischer Viewer /
eingeschränkter Browser

Thin Client

im Applikationsserver,
z.B. Jakarta EE-Anwendung

HTML + JavaScript im Browser

Client-Server-Architekturen

Architekturmuster für Web-Anwendungen (Fortsetzung)

Architekturmuster

Applikationslogik
(Model + Controller)

Präsentation
(View)

Zero Client

herstellerspezifischer
Applikationsserver

herstellerspezifischer Viewer /
eingeschränkter Browser

Thin Client

im Applikationsserver,
z.B. Jakarta EE-Anwendung

HTML + JavaScript im Browser

Rich Client

sowohl im Client als auch im
Applikationsserver

HTML + JavaScript im Browser,
WebAssembly

Client-Server-Architekturen

Architekturmuster für Web-Anwendungen (Fortsetzung)

Architekturmuster

Applikationslogik
(Model + Controller)

Präsentation
(View)

Zero Client

herstellerspezifischer
Applikationsserver

herstellerspezifischer Viewer /
eingeschränkter Browser

Thin Client

im Applikationsserver,
z.B. Jakarta EE-Anwendung

HTML + JavaScript im Browser

Rich Client

sowohl im Client als auch im
Applikationsserver

HTML + JavaScript im Browser,
WebAssembly

Fat Client
(managed)

Middleware mit Client-
Applikationsserver für
SW-Verteilung, etc.

clientseitiges Framework

Client-Server-Architekturen

Architekturmuster für Web-Anwendungen (Fortsetzung)

| Architekturmuster | Applikationslogik (Model + Controller) | Präsentation (View) |
|-------------------------------------|---|--|
| <u>Zero Client</u> | herstellerspezifischer Applikationsserver | herstellerspezifischer Viewer / eingeschränkter Browser |
| <u>Thin Client</u> | im Applikationsserver, z.B. Jakarta EE-Anwendung | HTML + JavaScript im Browser |
| <u>Rich Client</u> | sowohl im Client als auch im Applikationsserver | HTML + JavaScript im Browser, WebAssembly |
| <u>Fat Client</u> (managed) | Middleware mit Client- Applikationsserver für SW-Verteilung, etc. | clientseitiges Framework |
| Fat Client (OS-dependent) | überwiegend im Client: Desktop-Applikation | Windows- oder Linux-GUI |

Client-Server-Architekturen

Architekturmuster für Web-Anwendungen (Fortsetzung)

Architekturmuster

Applikationslogik
(Model + Controller)

Präsentation
(View)

Zero Client

Thin Client

Rich Client

Fat Client
(managed)

Fat Client
(OS-dependent)

“Data Shipping”



“Operation Shipping”

Bemerkungen:

- ❑ “Data Shipping” bzw. “Operation Shipping” ist aus der Sicht des Servers zu verstehen, der entweder (1) die (auf dem Server verarbeiteten) Daten oder (2) den Code (zur Verarbeitung der Daten) zum Client schickt.
- ❑ Ein Thin Client ist ein einfacher Computer, der für die Interaktion einer serverbasierten Computerumgebung optimiert wurde. Der Server übernimmt den größten Teil der Arbeit, z. B. das Starten von Softwareprogrammen, die Durchführung von Berechnungen und die Speicherung von Daten.
- ❑ Ein Fat Client ist ein voll ausgestatteter Rechner, der ein vollwertiges Betriebssystem, lokale Software und eigene Ressourcen wie Rechenleistung, Speicher und Netzwerkanbindung besitzt. Im Gegensatz zu einem Thin Client ist er in der Lage, viele Aufgaben im Standalone-Betrieb zu erledigen. [\[IP Insider\]](#)

Eine alternative Bezeichnung für Fat Client ist Thick Client.

Client-Server-Architekturen

Implementierung von Architekturmustern

- ❑ Ajax.
 - dynamisches (genauer: asynchrones) Web für Thin Clients
- ❑ REST.
 - Repräsentation und Modifikation von Ressourcen im Internet
- ❑ WebSockets.
- ❑ Remote Procedure Call RPC.
- ❑ Distributed Object Systems.
 - DCOM, CORBA, SOAP
- ❑ Message-oriented Middleware.
 - Broker-basierte Technologie zur asynchronen Kopplung von Server-Anwendungen

Bemerkungen (Middleware) :

- ❑ Technologien mit denen sich ein bestimmtes Architekturmuster implementieren lässt, werden mit dem Begriff “Middleware” in Verbindung gebracht.

Middleware – auch als “Architectural Glue” bezeichnet – realisiert die Infrastruktur für und zwischen Komponenten. Es gibt verschiedene Kategorien von Middleware, je nach Granularität und Art der Komponenten.

- ❑ Middleware ist Software, welche die Erstellung verteilter Anwendungen dadurch vereinfacht, dass sie standardisierte Mechanismen zur Kommunikation von verteilten Komponenten zur Verfügung stellt.

Kapitel WT:VI

VI. Architekturen und Middleware

- ☐ Client-Server-Architekturen
- ☐ Ajax
- ☐ REST
- ☐ WebSockets
- ☐ Remote Procedure Call RPC
- ☐ Message-oriented Middleware

Ajax

Einführung [Sequenzdiagramm (3-Tier)]

Ajax = *Asynchronous* JavaScript and XML

Ajax

Einführung [\[Sequenzdiagramm \(3-Tier\)\]](#)

Ajax = *Asynchronous* JavaScript and XML

Charakteristika:

- ❑ das synchrone Request-Response-Paradigma wird aufgebrochen
- ❑ Web-Seiten müssen nicht als Ganzes ersetzt, sondern können teilweise überladen werden. Schnittstelle: DOM-API
- ❑ auf klar definierten, offenen Standards basierend
- ❑ Browser- und plattformunabhängig
- ❑ es wird keine Art von „Ajax-Server“ benötigt, sondern auf bekannten Web-Server-Technologien aufgesetzt

Anwendung:

- ❑ Realisierung interaktiver [Thin Clients](#)
- ❑ (graphische) Bedienelemente mit Feedback

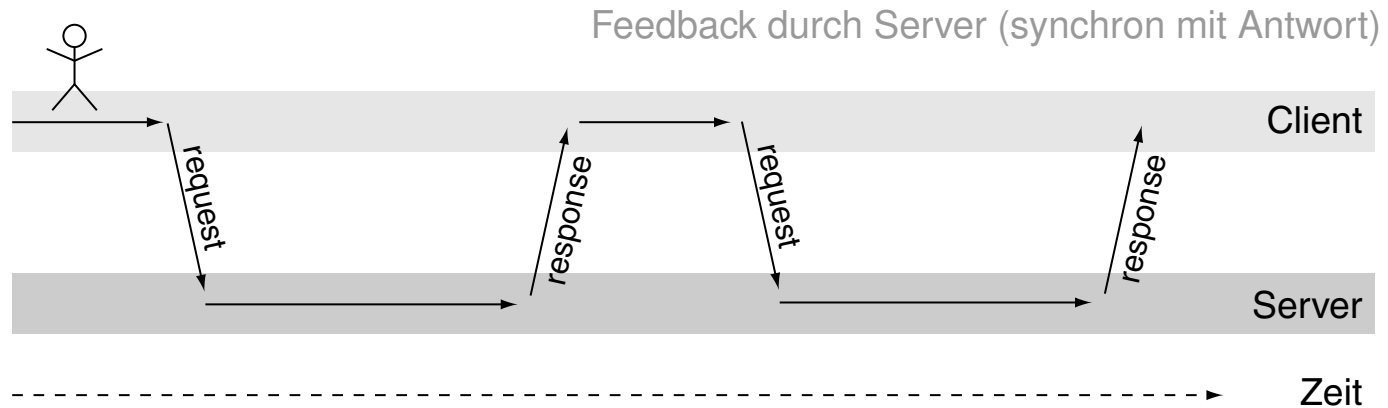
Bemerkungen:

- ❑ Die Kernidee von Ajax besteht darin, einen HTTP-Request *nebenläufig* (= non-blocking) auszuführen und das Ergebnis des Requests in den DOM-Seitenbaum des Browsers einzufügen.
- ❑ Teilweise wird Ajax als Client-Side-Technologie bezeichnet. [[apache.org](https://www.apache.org/)]
- ❑ Tatsächlich steht bei Ajax die Art und die Abwicklung der Kommunikation zwischen Client und Server im Vordergrund. Somit kann man Ajax als eine Technologie zur Umsetzung eines Architekturmusters verstehen: “Ajax isn’t a technology, it’s more of a pattern – a way to identify and describe a useful design technique.” [McCarthy 2005, [IBM](https://www.ibm.com/)]

Ajax

Einführung

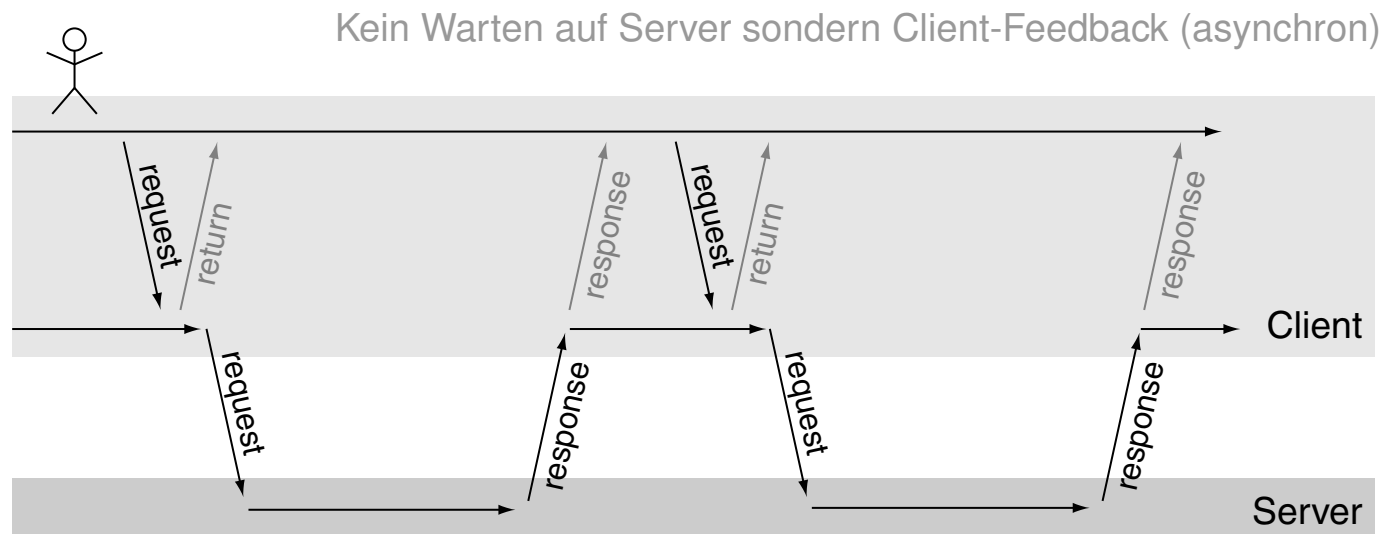
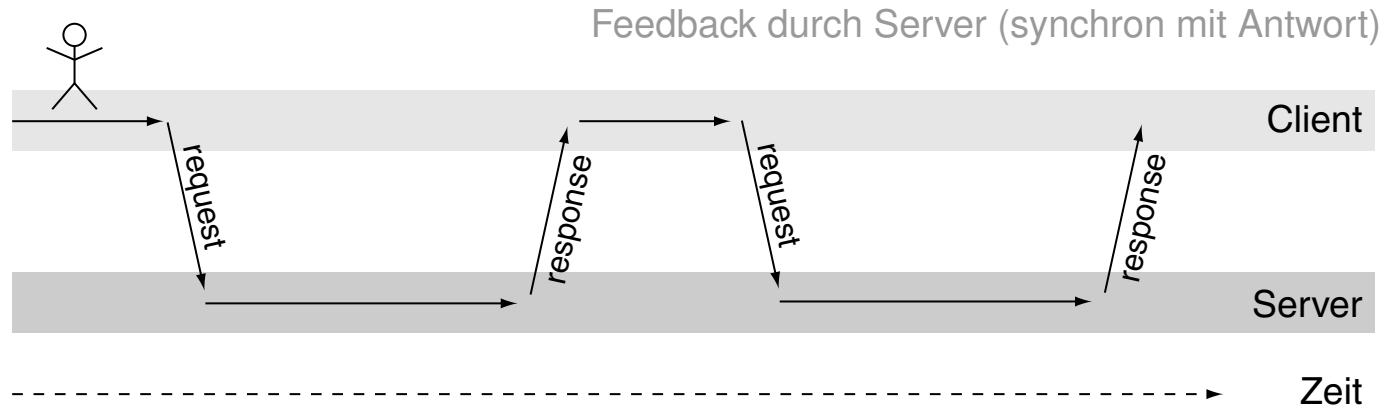
[Sequenzdiagramm (3-Tier)]



Ajax

Einführung

[Sequenzdiagramm (3-Tier)]



Ajax

Bestandteile einer Ajax-Anwendung

1. Event-Handler.
2. Server-Funktion.
3. Callback-Funktion.

Ajax

Bestandteile einer Ajax-Anwendung (Fortsetzung)

1. Event-Handler.

- ❑ realisiert als JavaScript-Funktion im Client, beispielsweise im HTML-Dokument
- ❑ wird bei entsprechender Anwenderaktion aufgerufen
- ❑ instanziiert – bei jedem Aufruf – ein XMLHttpRequest-Objekt
- ❑ meldet eine Callback-Funktion im XMLHttpRequest-Objekt an
- ❑ ruft die zur Anwenderaktion gehörende Server-Funktion auf

2. Server-Funktion.

3. Callback-Funktion.

Ajax

Bestandteile einer Ajax-Anwendung (Fortsetzung)

1. Event-Handler.

- ❑ realisiert als JavaScript-Funktion im Client, beispielsweise im HTML-Dokument
- ❑ wird bei entsprechender Anwenderaktion aufgerufen
- ❑ instanziiert – bei jedem Aufruf – ein XMLHttpRequest-Objekt
- ❑ meldet eine Callback-Funktion im XMLHttpRequest-Objekt an
- ❑ ruft die zur Anwenderaktion gehörende Server-Funktion auf

2. Server-Funktion.

- ❑ wird mittels Standardtechnologie (CGI, PHP-Script, Servlet, etc.) auf einem Web-Server zur Verfügung gestellt
- ❑ generiert eine XML-Datei mit Wurzel `<response>` als Rückgabewert

3. Callback-Funktion.

Ajax

Bestandteile einer Ajax-Anwendung (Fortsetzung)

1. Event-Handler.

- ❑ realisiert als JavaScript-Funktion im Client, beispielsweise im HTML-Dokument
- ❑ wird bei entsprechender Anwenderaktion aufgerufen
- ❑ instanziiert – bei jedem Aufruf – ein XMLHttpRequest-Objekt
- ❑ meldet eine Callback-Funktion im XMLHttpRequest-Objekt an
- ❑ ruft die zur Anwenderaktion gehörende Server-Funktion auf

2. Server-Funktion.

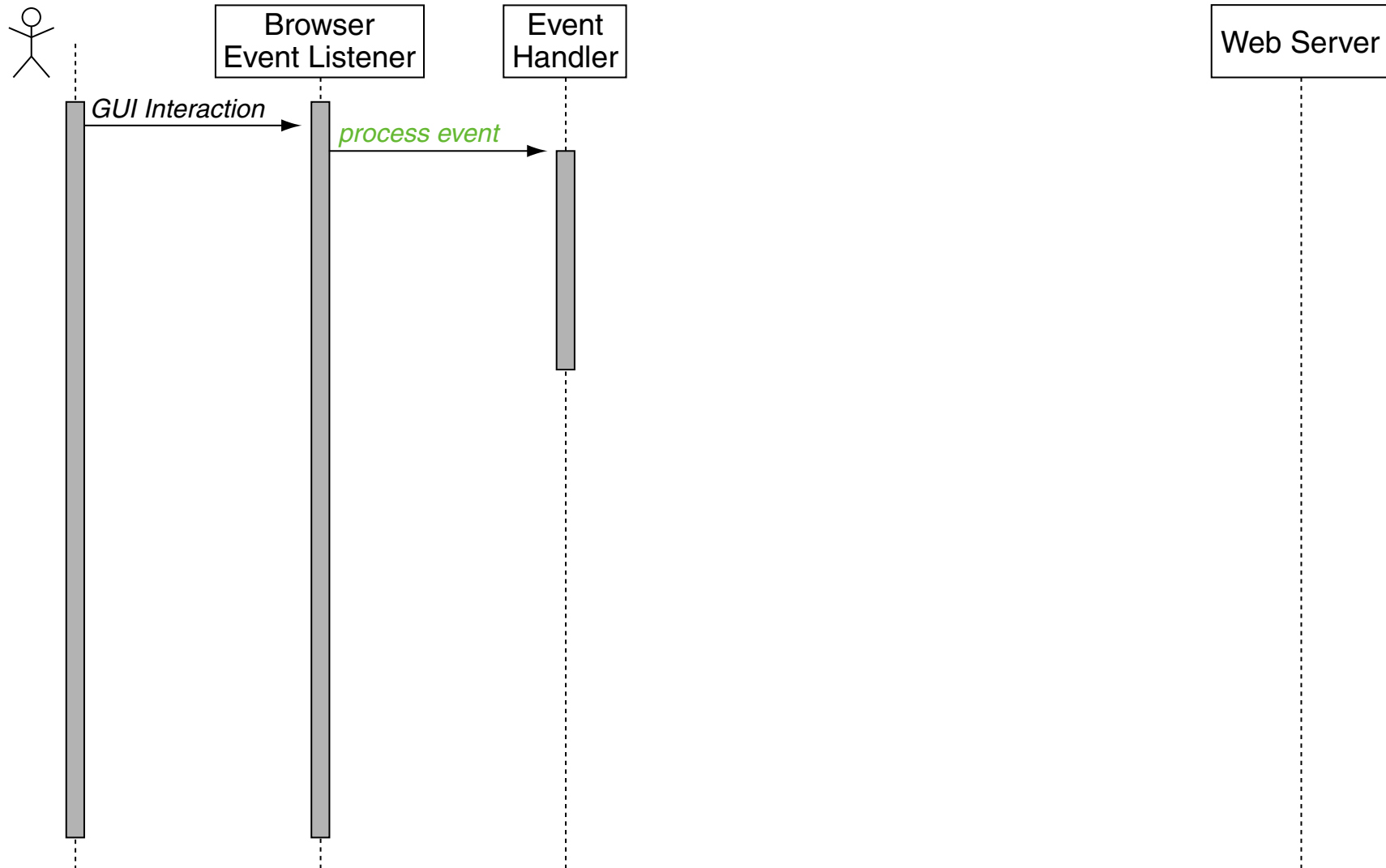
- ❑ wird mittels Standardtechnologie (CGI, PHP-Script, Servlet, etc.) auf einem Web-Server zur Verfügung gestellt
- ❑ generiert eine XML-Datei mit Wurzel `<response>` als Rückgabewert

3. Callback-Funktion.

- ❑ realisiert als JavaScript-Funktion im Client, typischerweise im HTML-Dokument
- ❑ wird mittels `readyState-Events` vom XMLHttpRequest-Objekt aufgerufen (“call back”)
- ❑ filtert bezüglich `readyState 4 (DONE)` und HTTP Status-Code 200 (OK)
- ❑ verarbeitet die zurückgegebene XML-Datei der Server-Funktion oder ruft für die Verarbeitung eine weitere JavaScript-Funktion auf und modifiziert den DOM

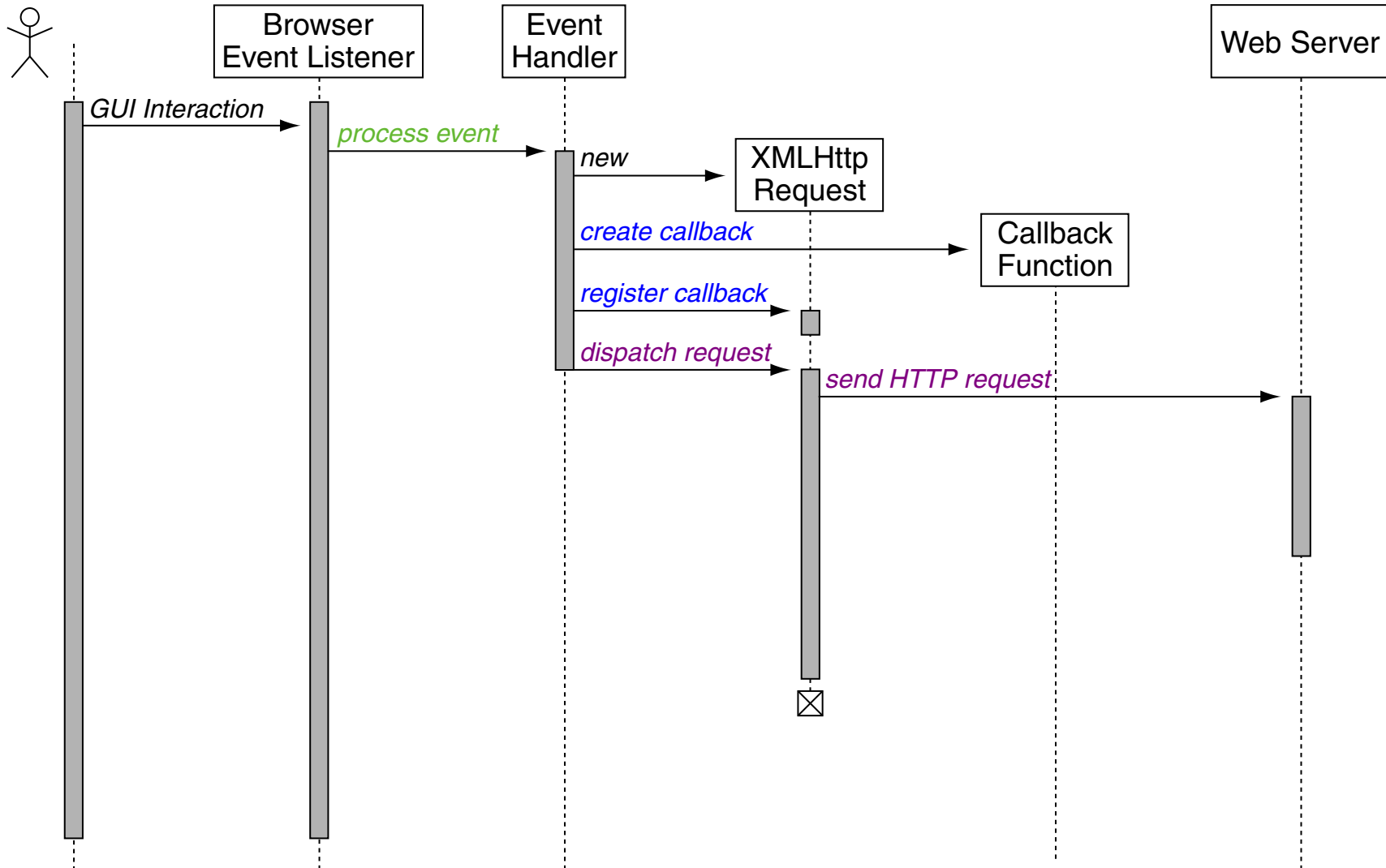
Ajax

Ablauf einer Ajax-Interaktion



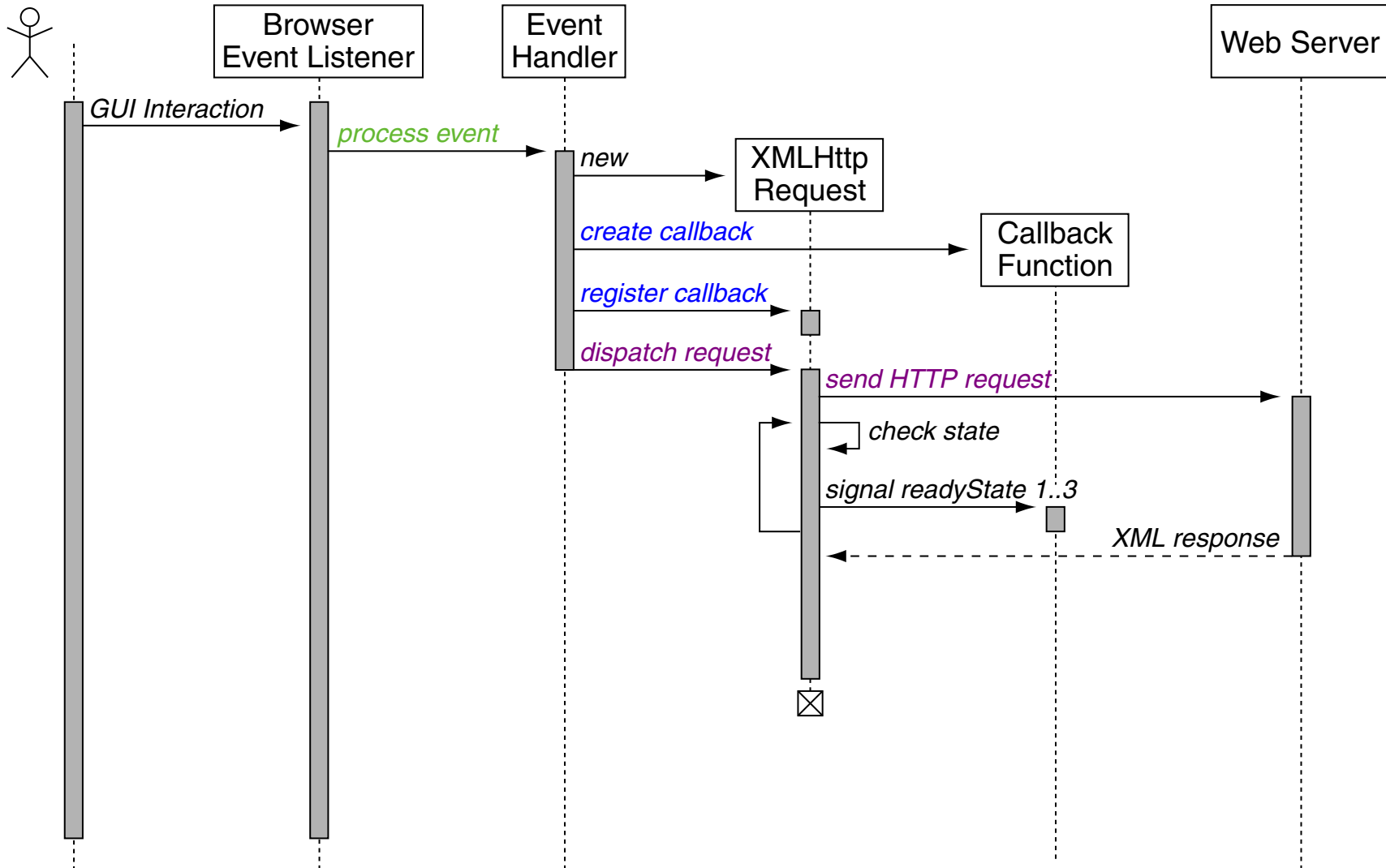
Ajax

Ablauf einer Ajax-Interaktion (Fortsetzung)



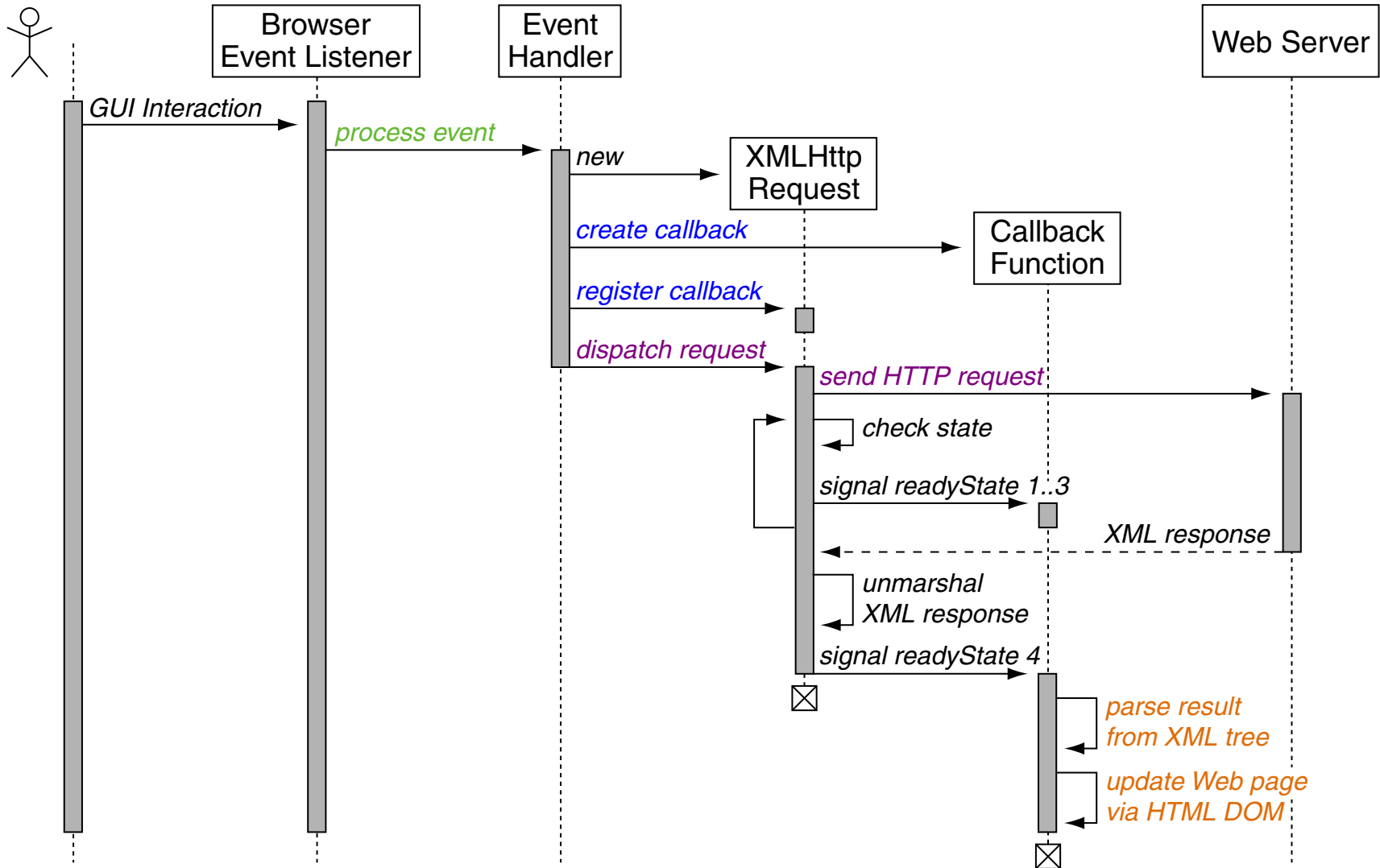
Ajax

Ablauf einer Ajax-Interaktion (Fortsetzung)



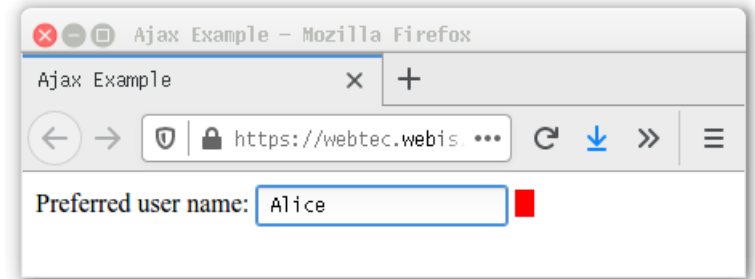
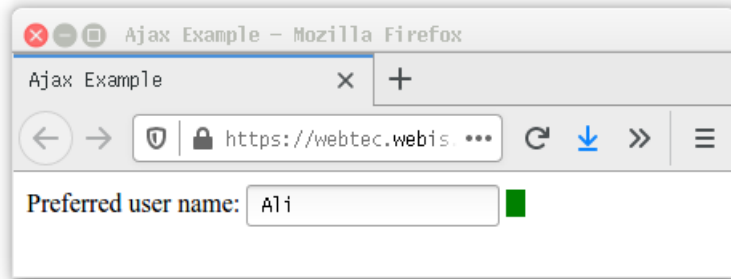
Ajax

Ablauf einer Ajax-Interaktion (Fortsetzung)



Ajax

Beispiel: Überwachung von Eingabefeld



[AJAX: [Aufruf](#)]

Beispiel: Überwachung von Eingabefeld (Fortsetzung)



```
<style>  
    #nameCheck.available{background-color: green;}  
    #nameCheck.unavailable{background-color: red;}  
</style>  
  
...  
<form action="">  
    <label for="username">Preferred user name:</label>  
    <input id="username" name="username" type="text" />  
    <span id="nameCheck" class="available">   </span>  
</form>
```


Beispiel: Überwachung von Eingabefeld (Fortsetzung) [Sequenzdiagramm (1+2)]

HTML-Datei:

```
<style>  
    #nameCheck.available{background-color: green;}  
    #nameCheck.unavailable{background-color: red;}  
</style>  
  
...  
<form action="">  
    <label for="username">Preferred user name:</label>  
    <input id="username" name="username" type="text" />  
    <span id="nameCheck" class="available">   </span>  
</form>
```

1a. JavaScript-Code zur Registrierung des generischen Event-Handlers:

```
let query = document.getElementById("username")
query.addEventListener("keyup",
    function() {
        genericEventHandler(
            "./check-username.php?q=" + query.value, // URL of server function.
            processUsernameResponse) // Reference to function object.
    }
)
```

Beispiel: Überwachung von Eingabefeld (Fortsetzung) [Sequenzdiagramm (1+2)]

HTML-Datei:

```
<style>  
    #nameCheck.available{background-color: green;}  
    #nameCheck.unavailable{background-color: red;}  
</style>  
  
...  
<form action="">  
    <label for="username">Preferred user name:</label>  
    <input id="username" name="username" type="text" />  
    <span id="nameCheck" class="available">   </span>  
</form>
```

1a. JavaScript-Code zur Registrierung des generischen Event-Handlers:

```
let query = document.getElementById("username")
query.addEventListener("keyup",
    function() {
        genericEventHandler(
            "./check-username.php?q=" + query.value, // URL of server function.
            processUsernameResponse) // Reference to function object.
        }
    )
```


Ajax

Beispiel: Überwachung von Eingabefeld (Fortsetzung) [\[Sequenzdiagramm \(1+2\)\]](#)

1b. JavaScript-Code des generischen Event-Handlers:

```
function genericEventHandler(url, processResponseXML) {  
    let req = new XMLHttpRequest();  
  
    if(req) {  
        req.addEventListener(                                // Register callback function.  
            "readystatechange",  
  
            function() {                                    // Create anonymous callback function.  
                if(req.readyState === 4) { // Check if readyState is DONE.  
                    if(req.status === 200) { // Check if server response is OK.  
                        processResponseXML(req.responseXML);  
                    } else { alert("HTTP error: " + req.status); }  
                }  
            }  
        );  
  
        req.open("GET", url, true); // Dispatch request.  
        req.send(null);             // Send HTTP request.  
    }  
}
```

Ajax

Beispiel: Überwachung von Eingabefeld (Fortsetzung) [\[Sequenzdiagramm \(1+2\)\]](#)

1b. JavaScript-Code des generischen Event-Handlers:

```
function genericEventHandler(url, processResponseXML) {  
    let req = new XMLHttpRequest();  
  
    if(req) {  
        req.addEventListener(                                // Register callback function.  
            "readystatechange",  
  
            function() {                                    // Create anonymous callback function.  
                if(req.readyState === 4) { // Check if readyState is DONE.  
                    if(req.status === 200) { // Check if server response is OK.  
                        processResponseXML(req.responseXML);  
                    } else { alert("HTTP error: " + req.status); }  
                }  
            }  
        );  
  
        req.open("GET", url, true); // Dispatch request.  
        req.send(null);             // Send HTTP request.  
    }  
}
```

Ajax

Beispiel: Überwachung von Eingabefeld (Fortsetzung) [[Sequenzdiagramm \(3\)](#)]

2. Server-Funktion `check-username.php`; generiert XML-Antwortdatei [PHP: [Aufruf](#)]:

```
<?php
header('Content-Type: text/xml');
echo '<?xml version="1.0" encoding="UTF-8" standalone="yes"?>';

function nameInUse($q) {
    if (isset($q)){
        switch(strtolower($q)) {
            case 'alice' :
            case 'bob' :
            case 'fred' :
            case 'mary' :
            case 'peter' :
                return '1';
            default:
                return '0';
        }
    }else{ return '0'; }
}

?>

<response>
    <result> <?php echo nameInUse($_GET['q']) ?> </result>
</response>
```

Ajax

Beispiel: Überwachung von Eingabefeld (Fortsetzung) [[Sequenzdiagramm \(3\)](#)]

2. Server-Funktion `check-username.php`; generiert XML-Antwortdatei [PHP: [Aufruf](#)]:

```
<?php
header('Content-Type: text/xml');
echo '<?xml version="1.0" encoding="UTF-8" standalone="yes"?>';

function nameInUse($q) {
    if (isset($q)){
        switch(strtolower($q)) {
            case 'alice' :
            case 'bob' :
            case 'fred' :
            case 'mary' :
            case 'peter' :
                return '1';
            default:
                return '0';
        }
    }else{ return '0'; }
}

?>

<response>
    <result> <?php echo nameInUse($_GET['q']) ?> </result>
</response>
```

Ajax

Beispiel: Überwachung von Eingabefeld (Fortsetzung) [\[Sequenzdiagramm \(3\)\]](#)

3. JavaScript-Code (aufgerufen aus [Callback-Funktion](#)) zur Verarbeitung der XML-Antwortdatei:

```
function processUsernameResponse(xmltree) {  
    let result = xmltree.documentElement  
        .getElementsByTagName('result')[0].firstChild.data;  
    let message = document.getElementById('nameCheck');  
  
    if(result === 1) { // Update the DOM regarding the database result.  
        message.className = 'unavailable';  
    } else {  
        message.className = 'available';  
    }  
}
```

Bemerkungen:

- ❑ Das Beispiel beschreibt ein wiederverwendbares Pattern für Ajax-Anwendungen:

Der JavaScript-Code des Event-Handlers (1b) kann unverändert in jeder Ajax-Anwendung zum Einsatz kommen. Die Server-Funktion (2) realisiert die eigentliche Datenverarbeitung und kommt zum Einsatz wie bislang auch. Zur Verarbeitung der XML-Antwort der Server-Funktion (2) ist eine JavaScript-Funktion (3) zu definieren.

Die Funktionen (2) und (3) bilden die Argumente von `genericEventHandler()`, der für beliebige Events im HTML-Dokument durch den Aufruf von `addEventListener()` (1a) registriert werden kann.

- ❑ Mittels `function() { ... }` in `genericEventHandler()` wird eine anonyme Callback-Funktion in Form eines Funktionsliterals definiert.

Programmiersprachentechnisches Konzept: Bei der anonymen Callback-Funktion handelt es sich um eine Closure. [\[Wikipedia\]](#)

Kapitel WT:VI

VI. Architekturen und Middleware

- ☐ Client-Server-Architekturen
- ☐ Ajax
- ☐ **REST**
- ☐ WebSockets
- ☐ Remote Procedure Call RPC
- ☐ Message-oriented Middleware

REST

Einführung

REST = Representational State Transfer

Im Web-Kontext:

- Etablierte Web-Semantik für Web-Service-APIs nutzen
- URLs definieren Ort und Namen der Ressourcen eines Web-Service
- die (Namen der) Request-Methoden des HTTP-Protokolls bezeichnen (passend ihrer Semantik) die Funktionen eines Web-Service

REST

Einführung

REST = Representational State Transfer

Im Web-Kontext:

- Etablierte Web-Semantik für Web-Service-APIs nutzen
- URLs definieren Ort und Namen der Ressourcen eines Web-Service
- die (Namen der) Request-Methoden des HTTP-Protokolls bezeichnen (passend ihrer Semantik) die Funktionen eines Web-Service

Historie

- 1995 Ursprung ist das von [Roy Fielding](#) entworfene HTTP Object Model
- 2000 Dissertation Roy Fielding: REST-Architekturstil bzw. “RESTful Application”
- 2014 steigende Aufmerksamkeit und Akzeptanz in der Web-Community
- 2022 beliebtester Architekturstil („Protokoll“) zur Spezifikation von Web-Service-APIs

REST

Einführung (Fortsetzung)

Geforderte Eigenschaften einer RESTful Application:

1. Client-Server-Architektur (Server stellt Dienst für Clients bereit)
2. **Zustandslosigkeit** (jeder REST-Aufruf enthält alle notwendigen Informationen)
3. Ausnutzung von HTTP Caching
4. **einheitliche Schnittstelle** (siehe Web-Kontext: Namen der HTTP-Request-Methoden)
5. Systeme sind mehrschichtig (Vereinfachung der Architektur)
6. Code on Demand (Client kann Code zur lokalen Ausführung erhalten)

REST

Einführung (Fortsetzung)

Geforderte Eigenschaften einer RESTful Application:

1. Client-Server-Architektur (Server stellt Dienst für Clients bereit)
2. **Zustandslosigkeit** (jeder REST-Aufruf enthält alle notwendigen Informationen)
3. Ausnutzung von HTTP Caching
4. **einheitliche Schnittstelle** (siehe Web-Kontext: Namen der HTTP-Request-Methoden)
5. Systeme sind mehrschichtig (Vereinfachung der Architektur)
6. Code on Demand (Client kann Code zur lokalen Ausführung erhalten)

Anwendung:

- ❑ Maschine-zu-Maschine-Kommunikation
- ❑ langlebige und selbstdokumentierende Web-Services

Bemerkungen:

- ❑ Für die Umsetzung des REST-Paradigmas wird ein zustandsloses Client-Server-Protokoll verwendet. Als Anwendungsschicht-Protokolle werden hauptsächlich HTTP und HTTPS eingesetzt.
- ❑ Wird über HTTP zugegriffen, so gibt die verwendete HTTP-Methode, darunter GET, POST, PUT und DELETE, an, welche Operation des Dienstes gewünscht ist.
- ❑ Die Methoden GET, HEAD, PUT und DELETE müssen laut HTTP-Spezifikation idempotent sein, was in diesem Zusammenhang bedeutet, dass das mehrfache Aufruf einer Funktion sich nicht anders auswirkt als ein einziger Aufruf.
- ❑ REST ist ein Programmierparadigma, das mit verschiedenen Mechanismen implementiert werden kann. Eine Besonderheit ist die Verwendung passender Namen von HTTP-Methoden in Zusammenhang mit der Semantik der auszuführenden Funktion.

Bemerkungen: (Fortsetzung)

- ❑ Für ein tieferes Verständnis der Verwendung von HTTP-Request-Methoden und deren intendierte Semantik sei auf die aktuelle RFC verwiesen [\[RFC 7231\]](#) :
 - [4.3.1](#) GET is the primary mechanism of information retrieval [...].
 - [4.3.3](#) The POST method requests that the target resource process [...] the request according to the resource's own specific semantics. For example, [...] providing a block of data, [...] creating a new resource.
 - [4.3.4](#) The PUT method requests that the state of the target resource be created or replaced with the state defined by the representation enclosed in the request message payload.
 - [4.3.5](#) For example, a resource that was previously created using a PUT request, [...] might allow a corresponding DELETE request to undo those actions.

REST

Konzepte

Einheitliche Abbildung der Funktionen (einer API) eines Web-Services auf „klassische“ Operationen bzw. Methodennamen:

| Web-Service | | CRUD | SQL | Java List | HTTP |
|-------------|---|--------|--------|-----------|--------|
| Funktion 1 | → | create | insert | add | POST |
| Funktion 2 | | read | select | get | GET |
| Funktion 3 | | update | update | set | PUT |
| Funktion 4 | | delete | delete | remove | DELETE |
| ... | | ... | ... | ... | ... |

Vergleiche die intendierte Semantik von HTTP-Request-Methoden. [\[RFC 7231\]](#)

REST

XML-Beispieldokument [\[personen.xml\]](#)

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl" href="personen.xsl" ?>

<personen>
  <person>
    <name>
      <vorname>Alan</vorname>
      <nachname>Turing</nachname>
    </name>
    <geburtstag>23. Juni 1912</geburtstag>
    <beruf>Mathematiker</beruf>
    <beruf>Informatiker</beruf>
  </person>

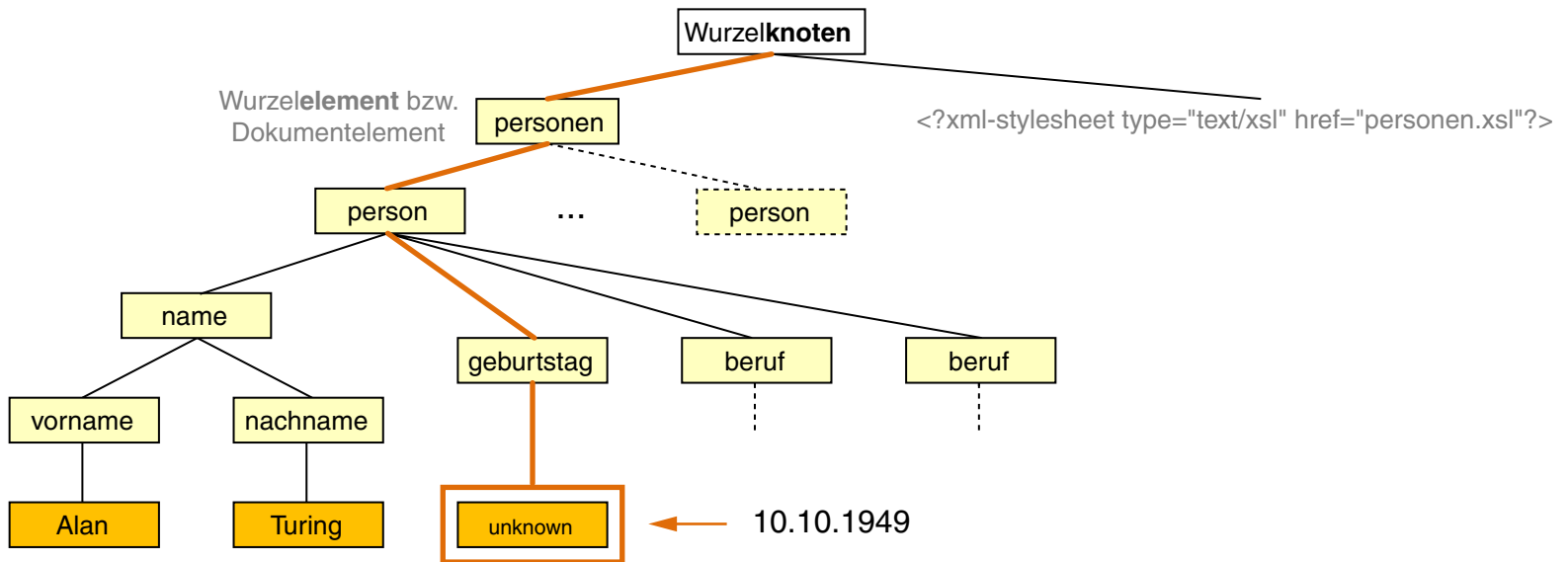
  <person>
    <name>
      <vorname>Judea</vorname>
      <nachname>Pearl</nachname>
    </name>
    <geburtstag>unknown</geburtstag>
    <beruf>Informatiker</beruf>
  </person>
</personen>
```

REST

XML-Beispieldokument (Fortsetzung)

Aufgabe [\[WT:III APIs für XML\]](#) :

1. Die Person „Judea Pearl“ finden.
2. Seinen Geburtstag auf einen bestimmten Wert setzen.



REST

Beispiel: Umsetzung als non-RESTful Web-Service

API-Aufrufe mit HTTP-GET + Parameter für Funktionen und Daten:

- ❑ Zurücksetzen in Ausgangssituation. [\[Aufruf\]](#)

`https://server/parameters/personen?action=reset`

- ❑ Anzeigen aller Personen. [\[Aufruf\]](#)

`https://server/parameters/personen?action=get`

- ❑ Anzeigen der Person an Index 1. [\[Aufruf\]](#)

`https://server/parameters/personen?person=1&action=get`

- ❑ Setzen des Geburtstages der Person an Index 1. [\[Aufruf\]](#)

`https://server/parameters/personen?person=1&action=set&geburtstag=10.10.1949`

REST

Beispiel: Umsetzung als RESTful Web-Service

API-Aufrufe mit **HTTP-GET/PUT** + **Representational State** + **Ressourcen-URL**:

- ❑ Zurücksetzen in Ausgangssituation.

```
curl -X GET "https://server/personen.xml" > personen.xml  
curl -X GET "https://server/pearl.xml" > pearl.xml  
curl -X PUT --data @personen.xml "https://server/rest/personen"
```

- ❑ Anzeigen aller Personen.

```
curl -X GET "https://server/rest/personen"
```

- ❑ Anzeigen der Person an Index 1.

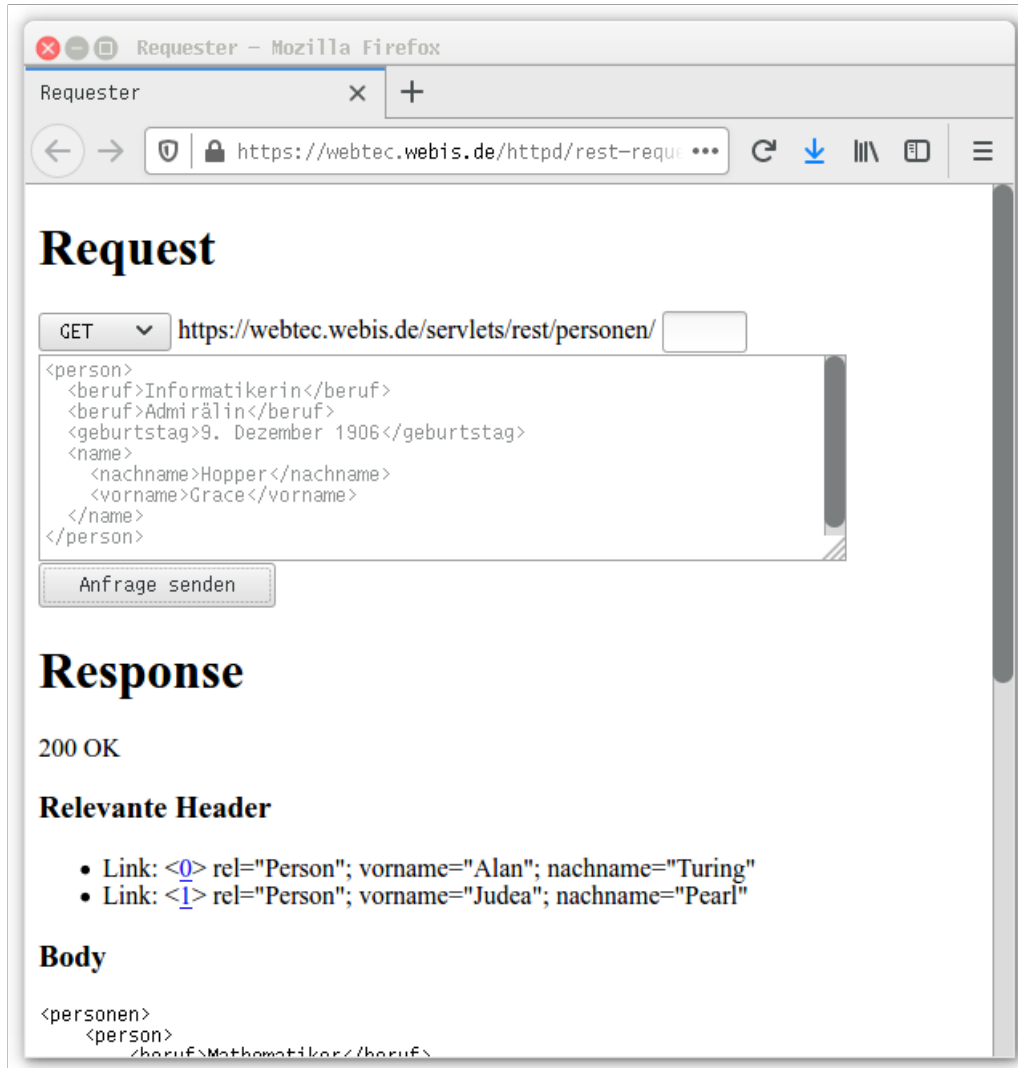
```
curl -X GET "https://server/rest/personen/1"
```

- ❑ Setzen des Geburtstages der Person an Index 1.

```
curl -X PUT --data @pearl.xml "https://server/rest/personen/1"
```

REST

Beispiel: Umsetzung als RESTful Web-Service (Fortsetzung)



Request

GET

```
<person>
  <beruf>Informatikerin</beruf>
  <beruf>Admirälin</beruf>
  <geburtstag>9. Dezember 1906</geburtstag>
  <name>
    <nachname>Hopper</nachname>
    <vorname>Grace</vorname>
  </name>
</person>
```

Anfrage senden

Response

200 OK

Relevante Header

- Link: <0> rel="Person"; vorname="Alan"; nachname="Turing"
- Link: <1> rel="Person"; vorname="Judea"; nachname="Pearl"

Body

```
<personen>
  <person>
    <beruf>Mathematiker</beruf>
```

[Aufruf]

Ajax

Quellen zum Nachlernen und Nachschlagen im Web

- ❑ Google. *Google Web Toolkit GWT*.
code.google.com/p/webtoolkit
- ❑ McCarthy. *Ajax for Java developers: Build dynamic Java applications*.
www.ibm.com/developerworks/java/library/j-ajax1
- ❑ McLellan. *Very Dynamic Web Interfaces*.
www.xml.com/pub/a/2005/02/09/xml-http-request.html
- ❑ W3 Schools. *Ajax Introduction*.
www.w3schools.com/ajax/xml/ajax_intro.asp