Verteilte Systeme 1

Technologien des World Wide Web

christian.zirpins@hs-karlsruhe.de

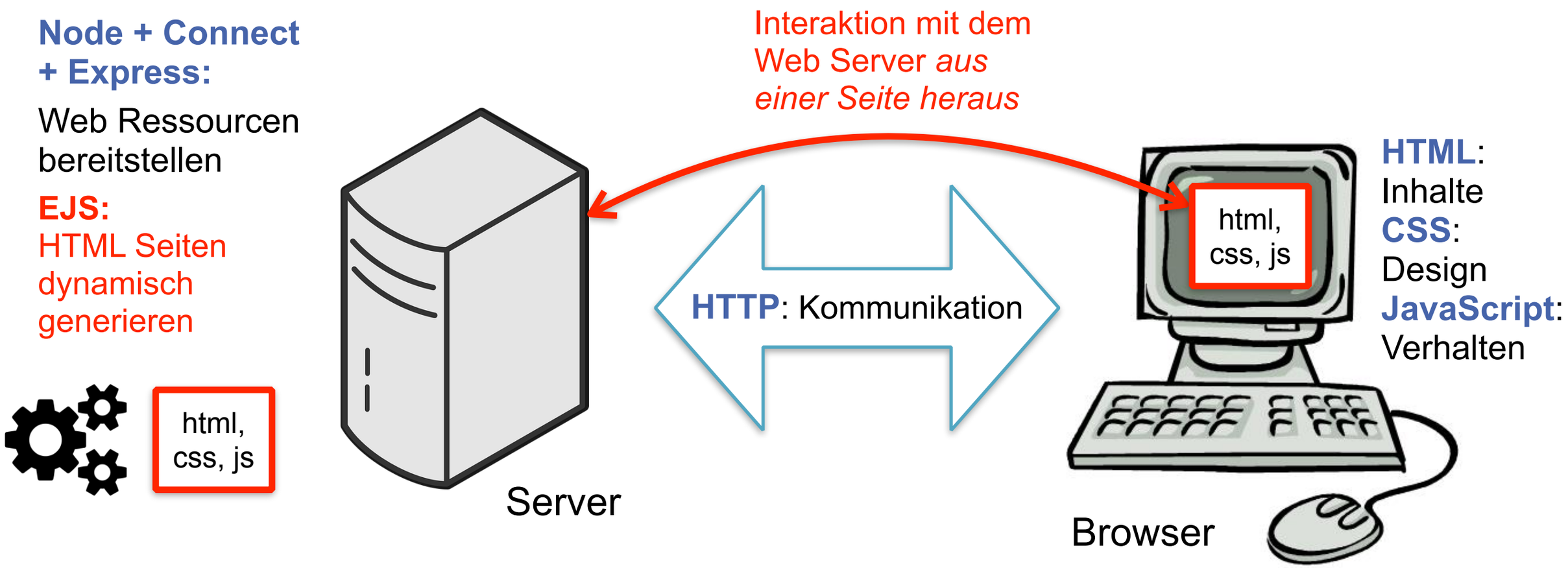
Web Entwicklung mit Ajax & Co

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Heutige Lernziele

Nach dieser Vorlesung können Sie…

- …**“reines” Ajax** (ohne Hilfe von jQuery) implementieren
- …den Zusammenhang von **Express und Connect** erklären
- …die **Template Sprache ejs** verwenden



Ajax

Dynamische Updates auf dem Client

Ajax

Nur im Namen

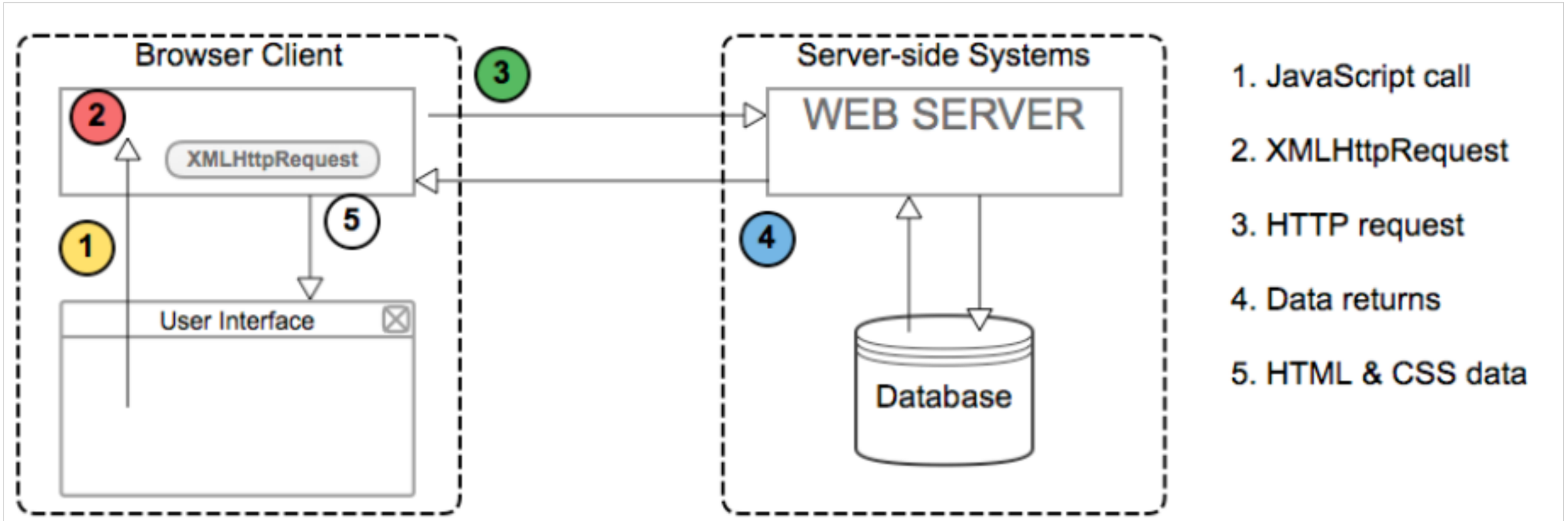
Asynchronous JavaScript and XML

- Ajax ist ein **JavaScript-Mechanismus**, der das dynamische Laden von Inhalten ermöglicht, ohne die Seite manuell neu zu laden oder abrufen zu müssen.
- Ajax ist eine Technologie (keine Sprache oder Produkt)
- Sie sehen diese Technologie jeden Tag: *Chats*, *endloses Scrollen*
- Ajax dreht sich um `XMLHttpRequest`, ein JavaScript Objekt
- **JQuery verbirgt alle Komplexität, macht Ajax-Aufrufe einfach**

Ajax - wie funktioniert das?

1. Webbrowser erstellt ein `XMLHttpRequest`-Objekt
2. `XMLHttpRequest` fordert Daten von einem Webserver an
3. Daten werden vom Server zurückgesendet
4. Auf dem Client fügt JavaScript-Code die Daten in die Seite ein

Ajax - wie funktioniert das?



Ohne Ajax…

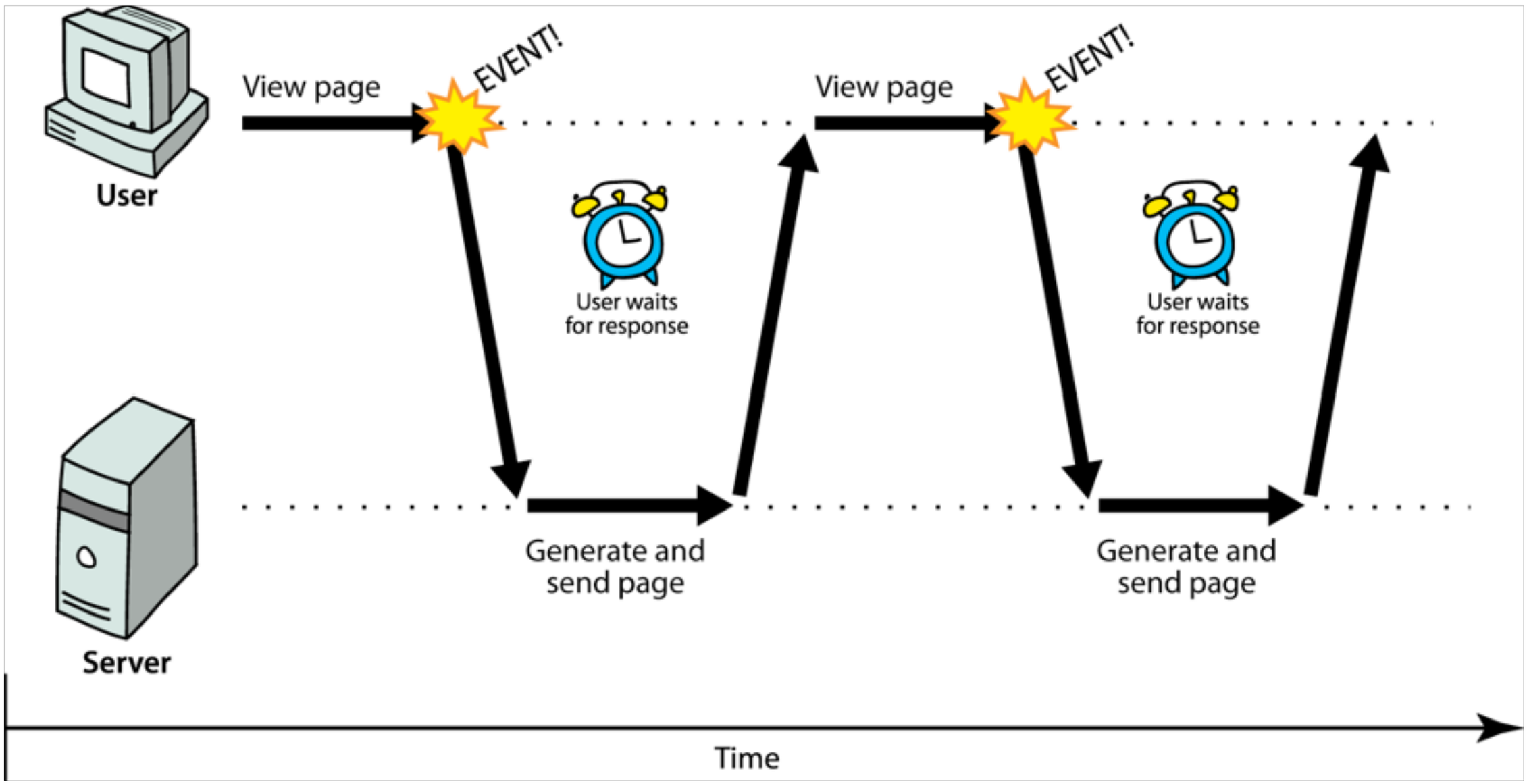


Abb. aus http://www.webstepbook.com/supplements-2ed/slides/chapter12-ajax-xml-json.shtml#slide2

Ajax funktioniert anders

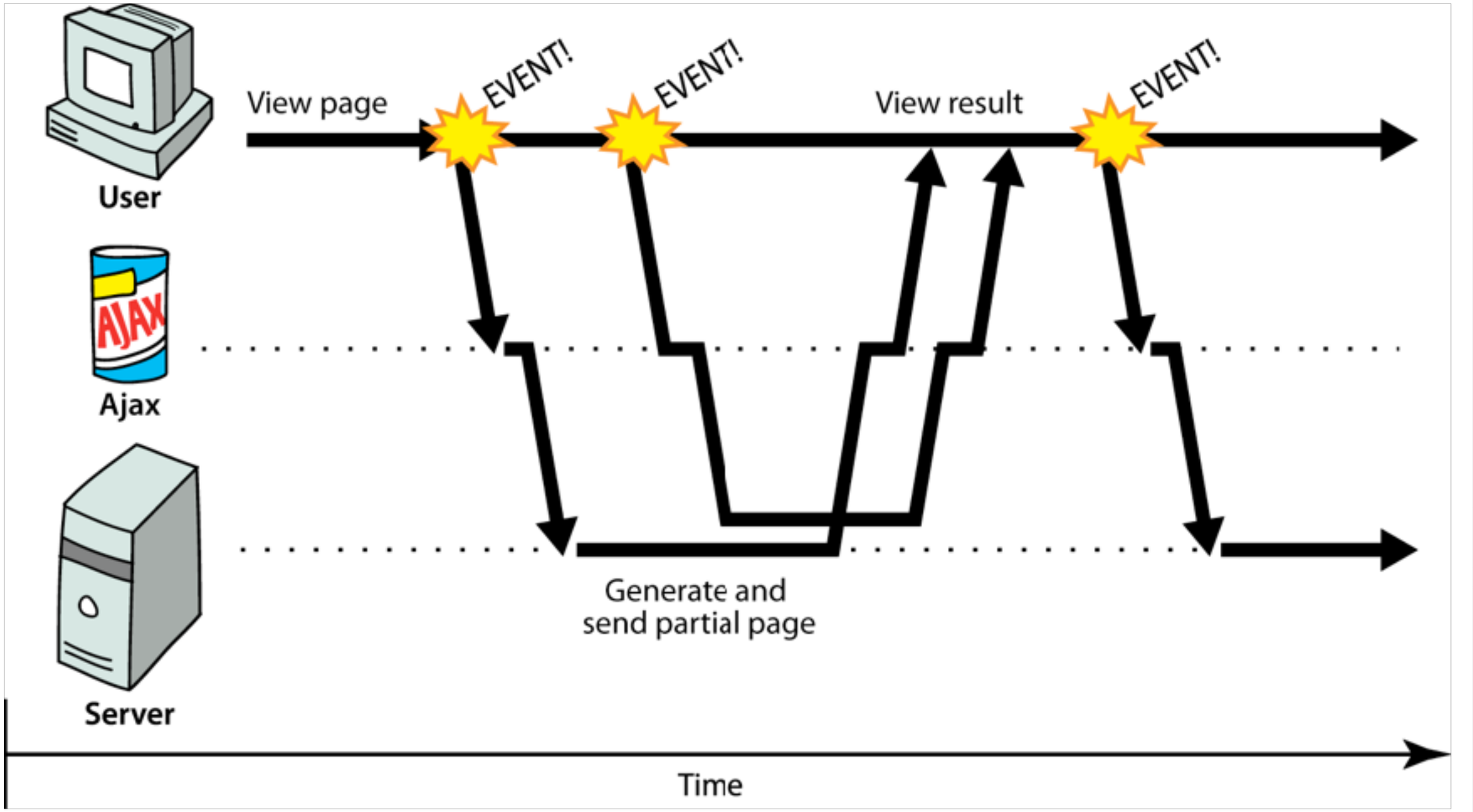


Abb. aus http://www.webstepbook.com/supplements-2ed/slides/chapter12-ajax-xml-json.shtml#slide2

Ajax: synchroner Request

```
// IE6 und frühere IE Versionen nutzen
// stattdessen Microsoft XMLHTTP
var ajax = new XMLHttpRequest();

// Lade Daten von URL (Datei)
// "false" Parameter: synchroner Request
ajax.open('GET', 'example.txt', false);
ajax.send(null);

// Antwortdaten in ajax.responseText
document.getElementById('ttExampleText').value =
  ajax.responseText;
```

Ausführung erst nach Ende von **`ajax.send`**

reines JS

Ajax: XMLHttpRequest

Eigenschaft	*Beschreibung*
`onreadystatechange`	Funktion, die aufgerufen wird, wenn sich die Eigenschaft `readyState` ändert
`readyState`	Hält den Status des `XMLHttpRequest`: 0: Request nicht initialisiert 1: Serververbindung hergestellt 2: Request eingegangen 3: Request wird verarbeitet 4: Request beendet, Antwort liegt vor
`responseText`	Antwortdaten als String
`responseXML`	Antwortdaten als XML
`status`	HTTP Status Code (z.B. "404" oder "200")
`statusText`	HTTP Status Text (z.B. "Not Found" oder "OK")

reines JS

Ajax: XMLHttpRequest

Eigenschaft	*Beschreibung*
`onreadystatechange`	Funktion, die aufgerufen wird, wenn sich die Eigenschaft **`readyState`** ändert
`readyState`	Hält den Status des **`XMLHttpRequest`**: 0: Request nicht initialisiert
`respon`	
`respon`	
`status`	
`status`	

Methode	*Beschreibung*
`abort()`	Abbruch des aktuellen Requests
`getAllResponseHeaders()`	Gibt Header Informationen zurück
`getResponseHeader()`	Gibt einzelne Header Informationen zurück
`open(method,url,async, uname,pswd)`	Bestimmt Request Typ, URL, ob der Request asynchron ausgeführt werden soll u.a. - *method*: GET, POST etc. - *url*: Ort der Datei auf dem Server - *async*: true (asynchron), false (synchron)
`send(string)`	Sendet den Request an den Server - *string*: Nur für POST Requests
`setRequestHeader()`	Fügt Name/Wert-Paar zum Header hinzu

reines JS

Ajax: asynchroner Request

```
var ajax = new XMLHttpRequest();

// Funktion, die bei Statusänderung
// aufgerufen wird
ajax.onreadystatechange = function() {
    // Zustand von Interesse
    if (ajax.readyState == 4) {
        // Verarbeite eingehende Daten
    }
}; //Ende der Funktion

ajax.open("GET", "url", true);
//true bedeutet asynchroner Request

ajax.send(null);
```

reines JS

Ajax: asynchroner Request

```
var ajax = new XMLHttpRequest();

// Funktion, die bei Statusänderu
// aufgerufen wird
ajax.onreadystatechange = function() {
    // Zustand von Interesse
    if (ajax.readyState == 4) {
        // Verarbeite eingehende Daten
    }
}; //Ende der Funktion

ajax.open("GET", "url", true);
//true bedeutet asynchroner Request

ajax.send(null);
```

onreadystatechange feuert, bei jeder (!) Änderung des Status

reines JS

Ajax Sicherheit

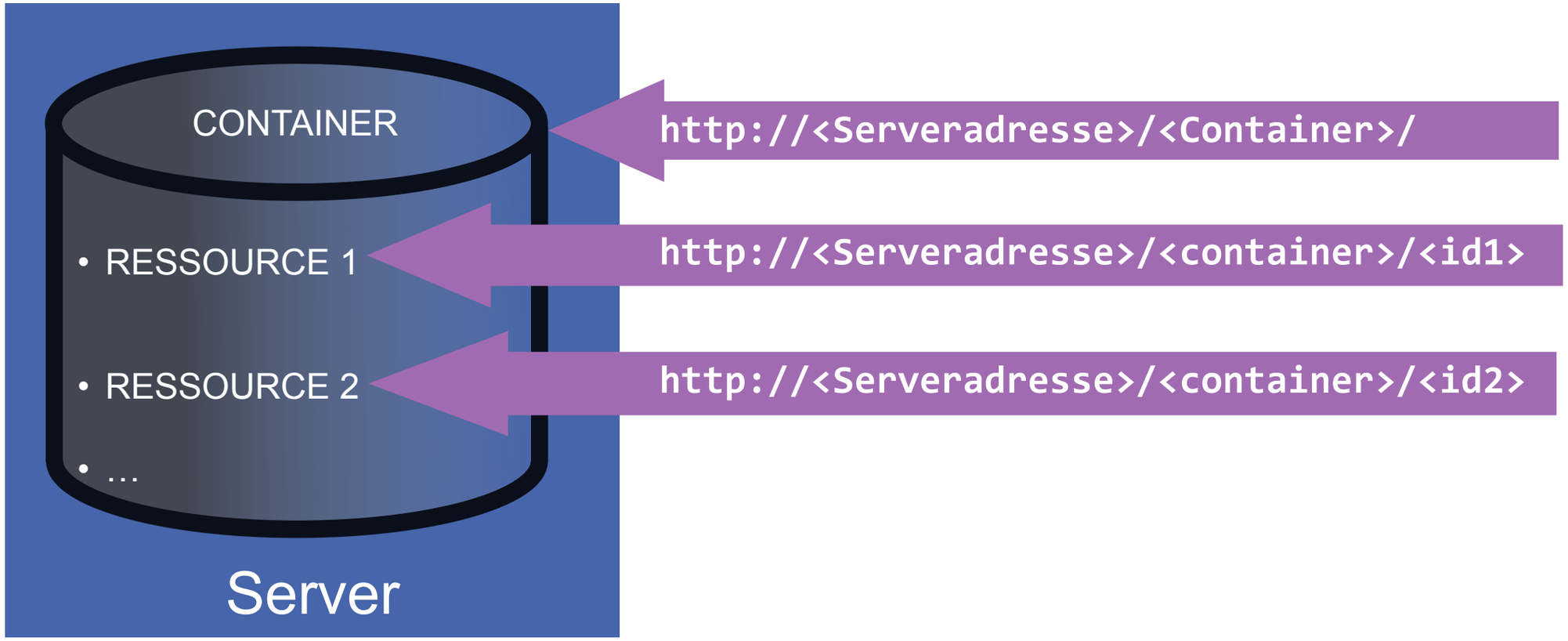
- Praktischerweise kamen alle Daten von "*unserem* Webserver"
- **Sicherheitsbeschränkung von Ajax**: kann nur Dateien vom selben Webserver abrufen wie die aufrufende Seite (**Same-Origin Policy**)
  - Gleicher Ursprung, wenn *Protokoll*, *Port* und *Host* für zwei Seiten gleich sind
- Ajax **kann nicht** von einer lokal auf der Festplatte gespeicherten Webseite ausgeführt werden

LWAD Buch erklärt, wie man dies umgeht (sollte man sein lassen)

Representational State Transfer (REST)

Was ist REST?

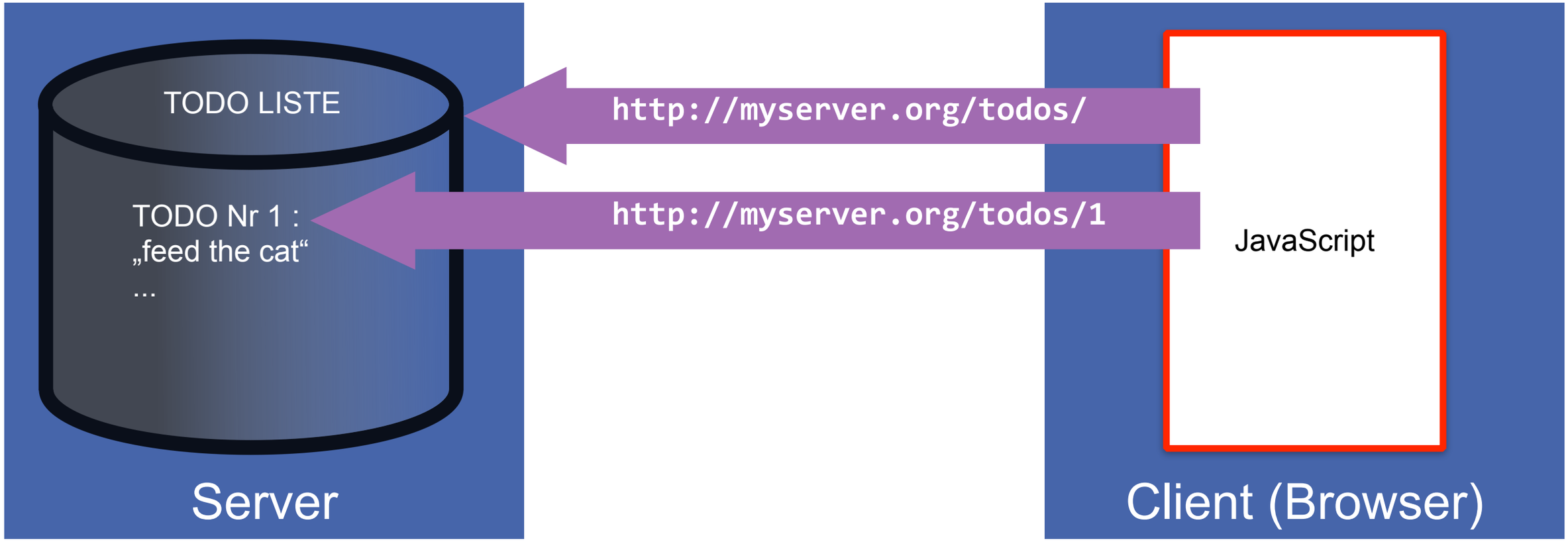
- **REST** ist ein Architektur-Stil für lose gekoppelte (Software) Systeme meist umgesetzt mit Web Techniken wie HTTP, URLs, XML/JSON
- Alle **Ressourcen** (Informationen) haben eine **URL** und können per **HTTP** erzeugt, gelesen, geändert oder gelöscht werden
- Darstellung wahlweise per XML, **JSON**, HTML und ggf. mit **Links**



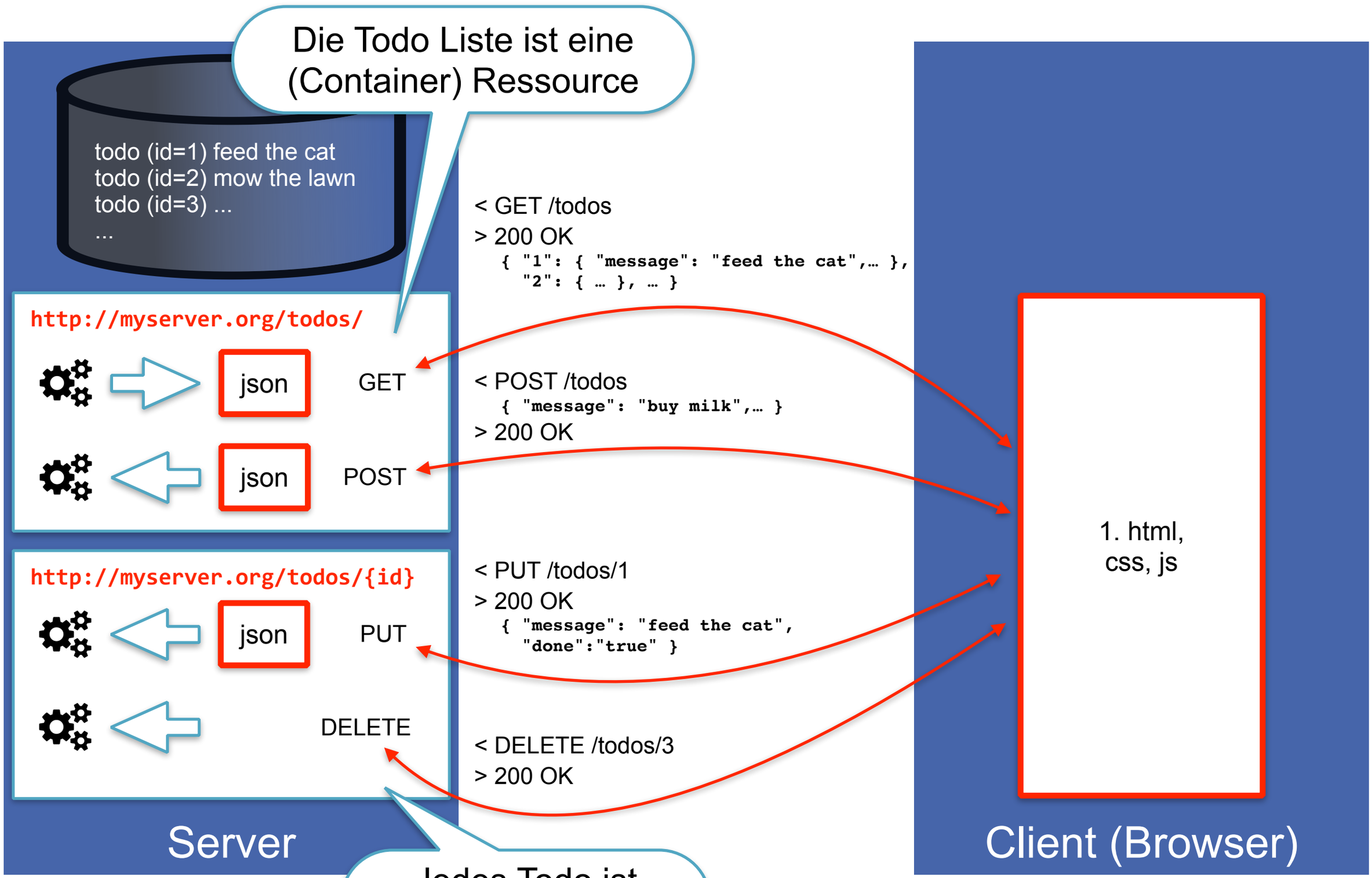
Representational State Transfer (REST)

REST und AJAX

- Mit Ajax können alle REST-Ressourcen direkt aus dem (Web) Client heraus genutzt werden
- Die Todo-Liste im vorherigen Beispiel ist eine REST-Ressource
  - Neben Lesen (GET) und Speichern (POST) von Todos könnten leicht weitere Routen für Änderung und Löschen ergänzt werden



Representational State Transfer (REST)



Node.js

Organisation und Wiederverwendung von Code

Bislang

Serverseitiger Code in einer einzigen Datei

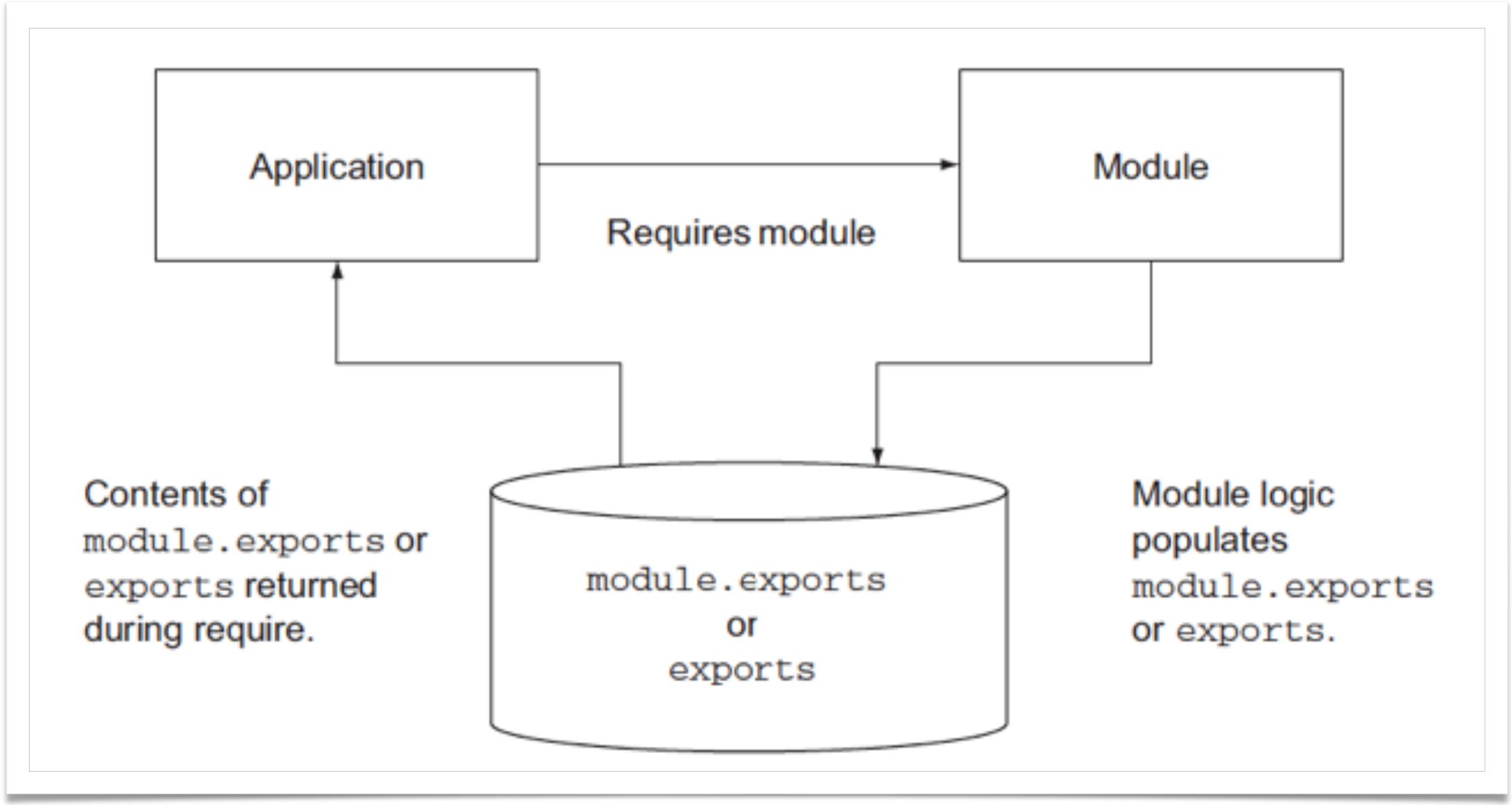
- Für kleine Projekte
- Größere Projekte leiden aber darunter, denn
  - das Debugging ist schwerfällig,
  - Teamarbeit ist umständlich,
  - die Programmierung ist umständlich.

node.js Module

- Code kann in Modulen organisiert werden
- Nicht alle Funktionen und Variablen in einem Modul werden der Anwendung offengelegt
  - Veröffentlichte Elemente werden über `exports` deklariert
- Module können via **npm veröffentlicht** werden
  - Die Verbreitung von Modulen für andere Entwickler ist einfach

node.js Module

- Code kann in Modulen organisiert werden
- Nicht alle Funktionen und Variablen in einem Modul werden der Anwendung offengelegt



Quelle: "Node.js in Action"

Ein Modul erstellen

Ein Modul besteht aus …

- einer einzelnen **Datei** oder
- einem **Verzeichnis** von Dateien (das eine Datei `index.js` enthält)

```
function roundGrade(grade) {
    return Math.round(grade);
}

function roundGradeUp(grade) {
    return Math.round(0.5 + parseFloat(grade));
}

exports.maxGrade = 10;
exports.roundGradeUp = roundGradeUp;
exports.roundGradeDown = function(grade) {
    return Math.round(grade - 0.5);
}
```

Nicht exportiert! Eine Anwendung kann die Funktion nicht nutzen

Ein Modul verwenden

```
var express = require("express");
var url = require("url");
var http = require("http");
var grading = require("./grades");
var app;

var port = process.argv[2];
app = express();
http.createServer(app).listen(port);

app.get("/round", function(req, res) {
    var query = url.parse(req.url, true).query;
    var grade = (query["grade"] != undefined) ?
        query["grade"] : "0";
    res.send("Rounding up: " +
        grading.roundGradeUp(grade) + ", and down: " +
        grading.roundGradeDown(grade));
});
```

`require` liefert den Inhalt des `exports` Objektes

Das Modul hinzufügen (gleiches Verzeichnis)

Modulfunktionen verwenden

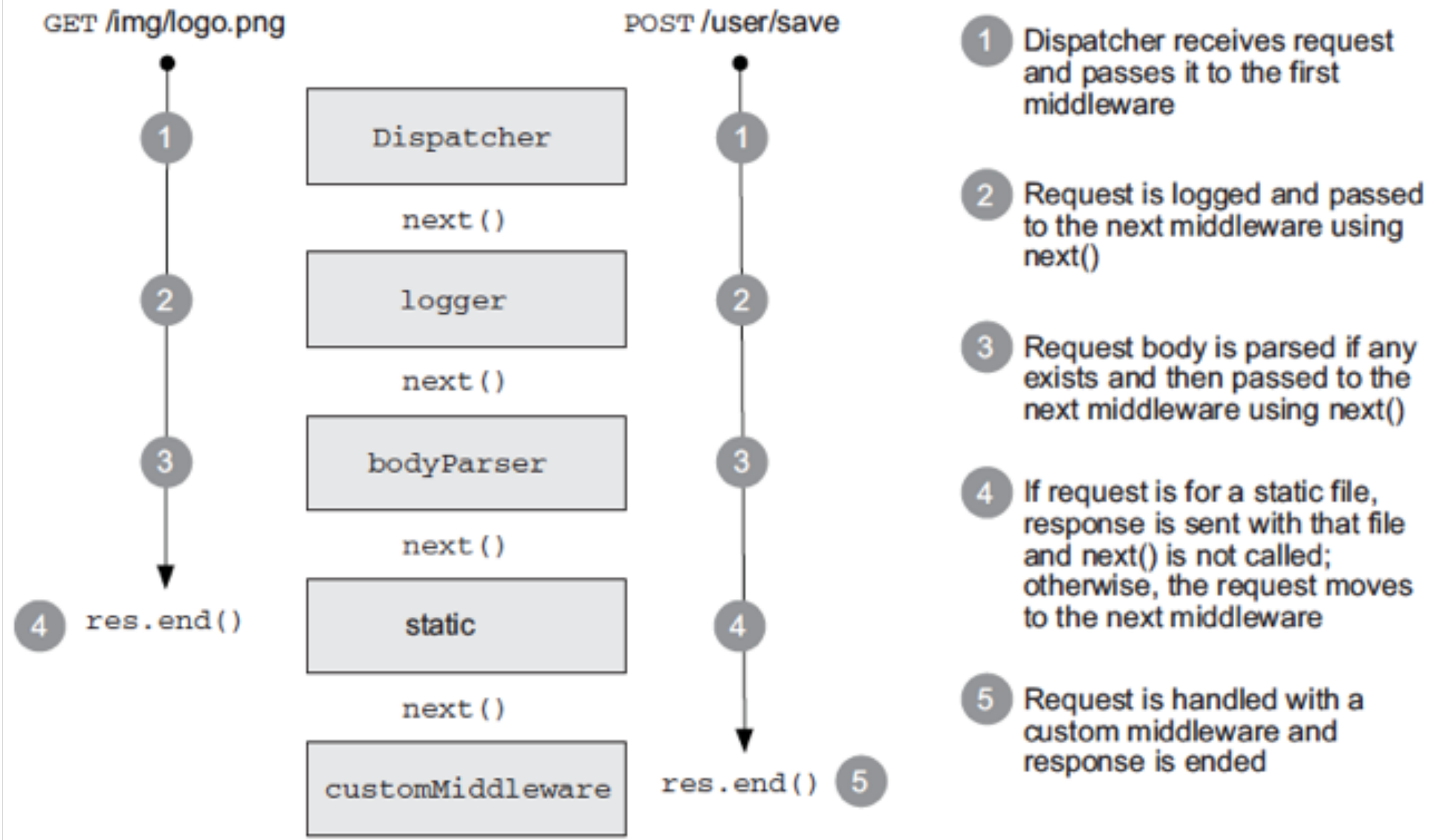
Express und Connect

Was ist Connect?

Express basiert auf **Connect**

- **Connect**: **Framework**, dessen Komponenten ("**Middleware**") verwendet werden, um die Logik einer Web-Anwendung zu erstellen (inspiriert von Ruby's Rack-Framework)
- Middleware: **Funktion**, die die Request- und Reply-Objekte des HTTP-Servers abfängt, die Logik ausführt und die Response beendet oder an die nächste Komponente übergibt
- **Dispatcher** verbindet die Komponenten (daher "Connect")
- Typische Komponenten: Request Logging (Protokollierung), Bereitstellung statischer Dateien, Request-Body-Parsing, Session-Verwaltung usw.

Connect Beispiel



Quelle: "Node.js in Action"

Ein minimales Connect Beispiel

```
var connect = require('connect');
var app = connect();
app.listen(3000)
```

Erinnert an Express Setup…

- Im Beispiel wird keine Middleware deklariert
- Dispatcher ruft jede Middleware-Komponente auf, bis eine antwortet
- Wenn keine antwortet (oder keine existiert), sendet der Dispatcher schließlich eine 404-Antwort an den Client zurück

Middleware Komponenten

- Middleware-Komponenten haben drei Argumente:
  - HTTP-Request Objekt
  - HTTP-Response Objekt
  - Callback-Funktion (`next()`), die anzeigt, dass die Komponente fertig ist und der Dispatcher zur nächsten Komponente wechseln kann
- Middleware-Komponenten sind **klein**, in sich **geschlossen** und **wiederverwendbar** zwischen Anwendungen

Einfache Logger Komponente

- **Ziel**: Erstellen einer Log Datei, die die Request-Methode und die URL von Requests aufzeichnet, die an den Server gehen
- **Erforderlich**: JavaScript-Funktion, die die Request- und Response-Objekte und die `next()` Callback Funktion entgegennimmt

```
var connect = require('connect');

// Middleware Logger Komponente
function logger(request, response, next) {
    console.log('%s\t%s\t%s', new Date(),
        request.method, request.url);
    next();
}
var app = connect();
app.use(logger);
app.listen(3001);
```

Einfache Logger Komponente

- **Ziel**: Erstellen einer Log Datei, die die Request-Methode und die URL von Requests aufzeichnet, die an den Server gehen
- **Erforderlich**: JavaScript-Funktion, die die Request- und Response-Objekte und die `next()` Callback Funktion entgegennimmt

```
var connect = require('connect');

// Middleware Logger Komponente
function logger(request, response, next) {
    console.log('%s\t%s\t%s', new Date(),
        request.method, request.url);
    next();
}
var app = connect();
app.use(logger);
app.listen(3001);
```

Kontrolle geht an den Dispatcher zurück

Middleware registrieren

Nun noch eine Antwort an den Client

- **Ziel**: antworte "Hello World" bei jedem Request
- **Erforderlich**: JavaScript-Funktion, die die Request- und Response-Objekte und die `next()` Callback Funktion entgegennimmt

```
var connect = require('connect');

function logger(request, response, next) { … }

function helloWorld(request, response, next) {
    response.setHeader('Content-Type',
      'text/plain');
    response.end('Hello World!');
}

var app = connect();
app.use(logger);
app.use(helloWorld);
app.listen(3001);
```

Kein Aufruf von next(); Kontrolle geht nicht an Dispatcher zurück!

Es können beliebig viele Komponenten registriert werden

Frage: Was tut dieser Code?

```
var connect = require('connect');
connect()
    .use(logger)
    .use('/admin', restrict)
    .use(serveStaticFiles)
    .use(hello);
```

```
function restrict(req, res, next) {
    var auth = req.headers.authorization;
    if (!auth)
        return next(new Error('Unauthorized'));

    var parts = auth.split(' ');
    var scheme = parts[0];
    var auth = new Buffer.from(parts[1], 'base64')
                        .toString().split(':');
    var user = auth[0]
    var pass = auth[1];
    if (user == "user" && pass == "password") {
        next();
    }
}
```

Frage: Was tut dieser Code?

```
var connect = require('connect');
connect()
    .use(logger)
    .use('/admin', restrict)
    .use(serveStaticFiles)
    .use(hello);

function restrict(req, res, next) {
    var auth = new Buffer.from(parts[1], 'base64')
                    .toString().split(':');
    var user = auth[0]
    var pass = auth[1];
    if (user == "user" && pass == "password") {
        next();
    }
}
```

Jeder Aufruf von **`use()`** gibt das Connect Objekt selbst zurück - ermöglicht Methodenverkettung.

Middleware Komponente wird nur dann aufgerufen, wenn der URL Präfix zutrifft

```
curl --user user:password http://localhost:3001/admin
```

Express Routen als Middleware

```
var express = require("express");
var url = require("url");
var http = require("http");

var port = 3000;
var app = express();
app.use(express.static(__dirname + "/client"));
http.createServer(app).listen(port);

var todos = [];
var t1 = {message: "Mow the cat",type: 1,deadline: "12/12/2016"};
var t2 = {message: "Feed the lawn",type: 3,deadline: "20/12/2016"};
todos.push(t1);
todos.push(t2);

function logger(request, response, next) {
   console.log('%s\t%s\t%s', new Date(), request.method, request.url);
   next();
}

function getTodos(request, response, next) {
   console.log("todos requested!");
   response.json(todos);
}

app.get("/todos", logger, getTodos);
```

Für eine Route können beliebig viele Komponenten registriert werden

Templates mit ejs

Express und HTML...

```
var express = require("express");
var url = require("url");
var http = require("http");
var app;

var port = process.argv[2];
app = express();
http.createServer(app).listen(port);

app.get("/greetme", function(req, res) {
    var query = url.parse(req.url, true).query;
    var name = (query["name"] != undefined) ?
        query["name"] : "Anonymous";
   res.send("<html><head></head><body><h1> Greetings "
          + name + "</h1></body></html>");
});

app.get("/goodbye", function(req, res) {
    res.send("Goodbye you!");
});
```

Express und HTML...

```
var express = require("express");
var url = require("url");
var http = require("http");
var app;

var port = process.argv[2];
app = express();
http.createServer(app).listen(port);

app.get("/greetme", function(req, res) {
    var query = url.parse(req.url, true).query;
    var name = (query["name"] != undefined) ?
        query["name"] : "Anonymous";
   res.send("<html><head></head><body><h1> Greetings "
          + name + "</h1></body></html>");
});

app.get("/goodbye", function(req, res) {
    res.send("Goodbye you!");
});
```

Fehleranfällig, nicht wartbar, scheitert bei jedem ernsthaften Projekt

Stattdessen: Templates

- **Ziel**: schreibe so wenig HTML "von Hand" wie möglich

- Hält den Code sauber - trennt Logik von Markup (HTML-Code)
- Für node.js gibt es viele verschiedene Template-Engines
- Wir konzentrieren uns zunächst auf **EJS** (Embedded JavaScript), eine **Template Engine und Sprache**

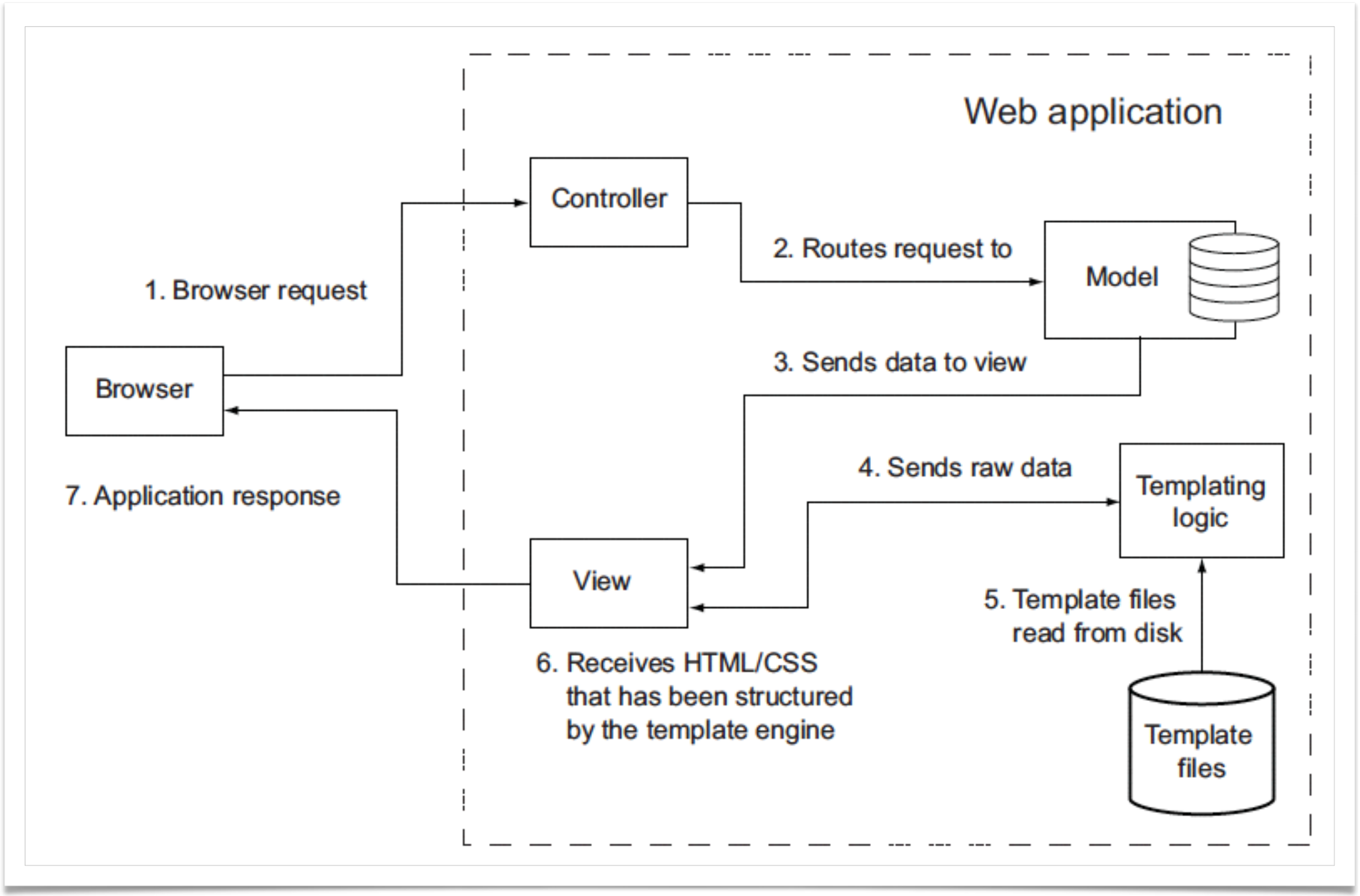
“EJS is a simple templating language that lets you generate HTML markup with plain JavaScript. No religiousness about how to organize things. No reinvention of iteration and control-flow. It's just plain JavaScript.” — http://ejs.co/

Model-View-Controller (MVC)

- Standard Entwurfsmuster um **Logik**, **Daten** und **Präsentation** getrennt zu halten
- User Request für Ressource auf dem Server bewirkt ...

1. **Controller** fordert Anwendungsdaten aus dem **Modell** an
2. **Controller** übergibt die Daten an die **View**
3. **View** formatiert die Daten für den Benutzer (hierzu werden Template-Engines verwendet)

Model-View-Controller (MVC)



Quelle: "Node.js in Action"

Ein erster Blick auf EJS

Start im Terminal: "`npm install ejs`" dann "`node test-ejs.js`"

```
var ejs = require('ejs');
var template = '<%= message %>';
var context = {
    message: 'Hello template!'
};
console.log(ejs.render(template, context));
```

test-ejs.js

Ein erster Blick auf EJS

Start im Terminal: "`npm install ejs`" dann "`node test-ejs.js`"

```
var ejs = require('ejs');
var template = '<%= message %>';
var context = {
    message: 'Hello template!'
};
console.log(ejs.render(template, context));
```

test-ejs.js

`<%-` wenn Escaping nicht gewünscht wird
(Achtung, ermöglicht Cross-Site Scripting Angriffe!)

```
var ejs = require('ejs');
var template = '<%= message %>';
var context = {
    message: "<script>alert('hi!');</script>"
};
console.log(ejs.render(template, context));
```

test-ejs2.js

Standardmäßig "escaped" ejs reservierte Zeichen zu ‚html Entities‘:

```
&lt;script&gt;alert(&#39;hi!&#39;);&lt;/script&gt;
```

EJS Tags

- `<%` Scriptlet-Tag, für Kontrollfluss, ohne Ausgabe
- `<%_` Scriptlet-Tag, das alle Leerzeichen davor entfernt
- `<%=` gibt einen Wert in das Template aus ("escaped")
- `<%-` gibt einen Wert in das Template aus (ohne "Escaping")
- `<%#` Kommentar Tag, keine Ausführung, keine Ausgabe
- `<%%` gibt Literal '`<%`' aus
- `%%>` gibt Literal '`%>`' aus
- `%>` Einfaches Ende-Tag
- `-%>` Trim-Modus Tag, entfernt nachfolgende Zeilenumbruch
- `_%>` Ende-Tag, das alle Leerzeichen danach entfernt

EJS Templates

- Templates werden in Funktionen übersetzt.

```
var ejs = require('ejs');
var fs = require('fs');
var template = fs.readFileSync('views/list.ejs', 'utf8');

var tmp_fun = ejs.compile(template);

var ret = tmp_fun({
  names: ['foo', 'bar', 'baz']
});

console.log(ret);
```

ejs-list.js

```
<% if (names.length) { -%>
  <ul>
  <%_ names.forEach(function (name) { -%>
    <li foo='<%= name _%> '><%= name %></li>
  <%_ }); -%>
  </ul>
<% } %>
```

list.ejs

Ausgabe:

```
<ul>
  <li foo='foo'>foo</li>
  <li foo='bar'>bar</li>
  <li foo='baz'>baz</li>
</ul>
```

EJS Templates

- Templates werden in Funktionen übersetzt.

```
var ejs = require('ejs');
var fs = require('fs');
var template = fs.readFileSync('views/list.ejs', 'utf8');

var tmp_fun = ejs.compile(template);

var ret = tmp_fun({
  names: ['foo', 'bar', 'baz']
});

console.log(ret);
```

ejs-list.js

Template aus Datei laden

Erzeuge Template-Funktion (kann mehrfach angewendet werden).

```
<% if (names.length) { -%>
  <ul>
  <%_ names.forEach(function (name) { -%>
    <li foo='<%= name _%> '><%= name %></li>
  <%_ }); -%>
  </ul>
<% } %>
```

list.ejs

Kontrollfluss mit reinem Javascript (werden nicht gerendert).

Umbrüche (und Leerzeichen) ignorieren

Ausgabe:

```
<ul>
  <li foo='foo'>foo</li>
  <li foo='bar'>bar</li>
  <li foo='baz'>baz</li>
</ul>
```

Konfigurieren von Views mit Express

- Einstellen des "Views" Verzeichnisses (enthält die Templates)

```
app.set('views', __dirname + '/views');
```

Globale Variable in Node

- Einstellen der Template-Engine

```
app.set('view engine', 'ejs');
```

- Hinweis: eine Anwendung kann mehrere Template-Engines gleichzeitig nutzen
- Für node.js gibt es zahlreiche Template-Engines

Daten an einen View übertragen

```
var express = require("express");
var url = require("url");
var http = require("http");
var app;

var port = process.argv[2];
app = express();
http.createServer(app).listen(port);

var todos = [];
todos.push({message: 'Midterm exam tomorrow',dueDate: '12/11/2014'});
todos.push({message: 'Prepare for assignment 5',dueDate: '05/01/2015'});
todos.push({message: 'Sign up for final exam',dueDate: '06/01/2015'});

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');

app.get("/todos", function(req, res) {
    res.render('todos', {
        title: 'My list of TODOs',
        todo_array: todos
    });
});
```

Daten an einen View übertragen

```
var express = require("express");
var url = require("url");
var http = require("http");
var app;

var port = process.argv[2];
app = express();
http.createServer(app).listen(port);

var todos = [];
todos.push({message: 'Midterm exam tomorrow',dueDate: '12/11/2014'});
todos.push({message: 'Prepare for assignment 5',dueDate: '05/01/2015'});
todos.push({message: 'Sign up for final exam',dueDate: '06/01/2015'});

ap                                    'views');
ap

app.get("/todos", function(req, res) {
    res.render('todos', {
        title: 'My list of TODOs',
        todo_array: todos
    });
});
```

ToDo Liste die dem Client bereitgestellt werden soll

render zeigt die Nutzung eines Templates an

Name des Templates (Datei ohne Endung)

Variablen des Templates

EJS Template Datei

```
<!DOCTYPE html>
<html>

<head>
    <title>
        <%= title %>
    </title>
</head>

<body>
    <h1>TODOs</h1>
    <div>
        <% todo_array.forEach(function(todo) { %>
            <div>
                <h3><%= todo.dueDate %></h3>
                <p>
                    <%= todo.message %>
                </p>
            </div>
        <% }) %>
    </div>
</body>

</html>
```

JavaScript zwischen `<%  %>` wird ausgeführt.

JavaScript zwischen `<%=  %>` fügt Inhalte zur Ergebnisdatei hinzu.

Ein letztes Wort zu Templates

- Ejs behält die ursprünglichen HTML-Tags bei. Andere Template-Sprachen tun das ggf. nicht.
- **Pug** (ehem. **Jade**) ist ein beliebtes Beispiel.

```
doctype html
html(lang="en")
  head
    title= pageTitle
    script(type='text/javascript').
      if (foo) {
         bar(1 + 5)
      }
  body
    h1 Jade - node template engine
    #container.col
      if youAreUsingJade
        p You are amazing
      else
        p Get on it!
      p.
        Jade is a terse and simple
        templating language with a
        strong focus on performance
        and powerful features.
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Jade</title>
    <script type="text/javascript">
      if (foo) {
         bar(1 + 5)
      }
    </script>
  </head>
  <body>
    <h1>Jade - node template engine</h1>
    <div id="container" class="col">
      <p>You are amazing</p>
      <p>
        Jade is a terse and simple
        templating language with a
        strong focus on performance
        and powerful features.
      </p>
    </div>
  </body>
</html>
```

https://pugjs.org/

Literatur

- Learning Web App Development, Kapitel 4
- Ethan Brown, *"Web development with Node and Express",* O'Reilly, 2014
- Robert Prediger ; Ralph Winzinger, *"Node.js"*, Hanser, 2015 (Online verfügbar im Hochschulnetz)
- Mike Cantelon, *"Node.js in action"*, Manning, 2014
- Connect, https://github.com/senchalabs/connect