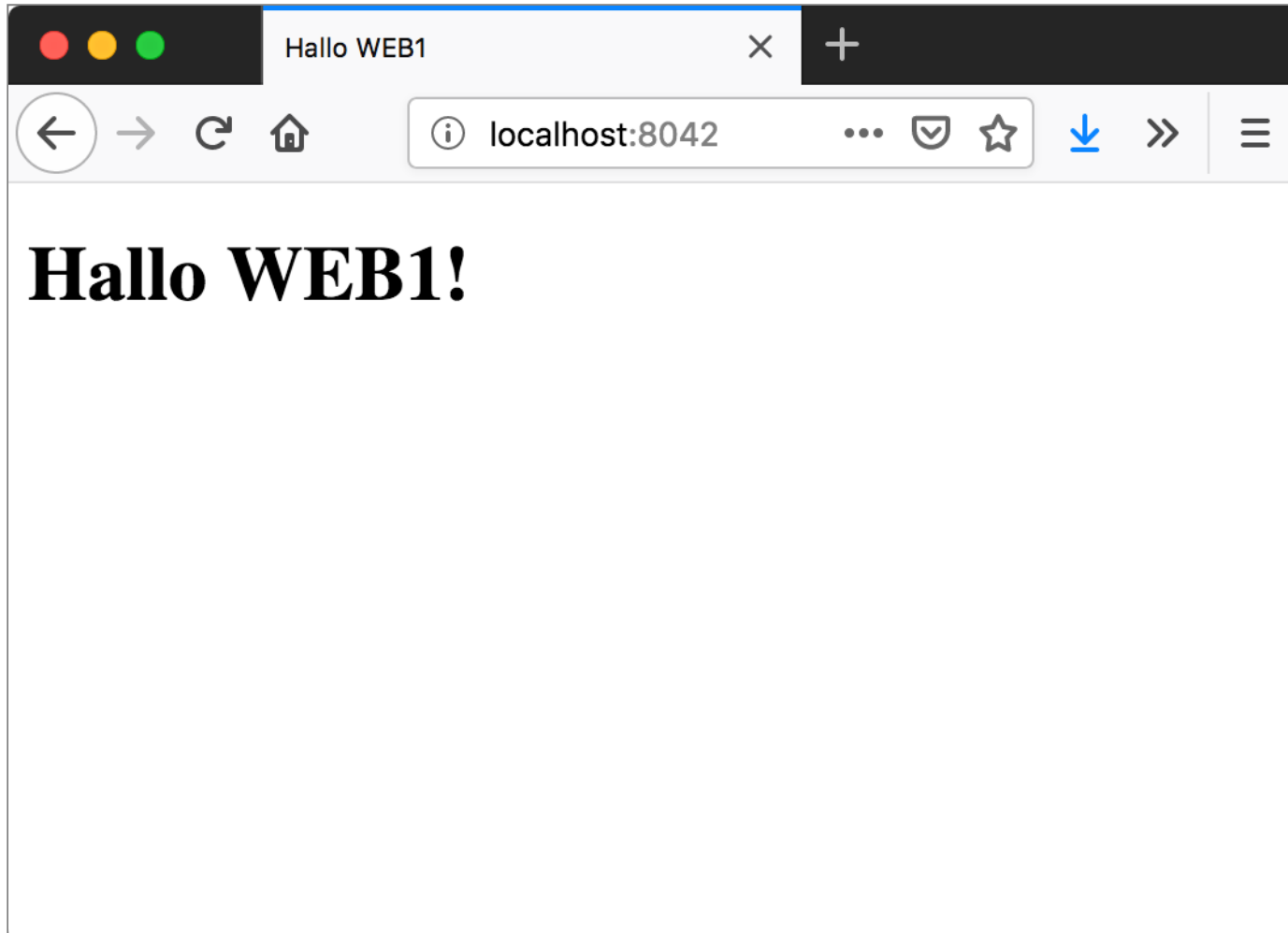


# BEISPIEL: EINFACHE HTML-ANTWORT

Ausgabe im Browser:



```
sven@tsu:~ $ node webServer.js  
Ich lausche nun auf http://localhost:8042  
█
```

### ! Beobachtung:

Mit Benutzung der `listen`-Funktion beendet sich unser Server nicht mehr von selbst (wir müssen den Prozess bewusst beenden, z.B. mit `Ctrl+C`).

### ! Grund:

Der Server "läuft" und wartet auf bestimmte **Ereignisse**



## Zentrale Eigenschaften von Node.js:

- Asynchrone (nicht-blockierende) Ein- und Ausgabe
- Ereignisgetrieben
- Modularer Aufbau

# EREIGNISGETRIEBENE PROGRAMMIERUNG

- Statt eines rein sequentiellen Ablaufes beschreibt der Programmcode, wie auf das Eintreffen bestimmter *Ereignisse* reagiert werden soll
- Beispiele für Ereignisse:
  - Eintreffen einer Anfrage
  - Einlesen einer Datei abgeschlossen
  - Senden von Daten über das Netzwerk abgeschlossen
- Auch das Arbeiten mit DOM-Events im Browser ist ein Beispiel für ereignisgetriebene Programmierung

# EREIGNISSE IN NODE.JS

- Das **Kernmodul "events"** [↗](#) liefert die Basis für ereignisgetriebene Programmierung in Node.js
- Das Modul definiert dazu das `EventEmitter`-Objekt, welches u.A. folgende zentrale Funktionen bietet:

Funktion	Zweck
<a href="#">on</a> <a href="#">↗</a>	Registriert einen <i>Event-Listener</i> für ein bestimmtes Ereignis. Parameter: <ol style="list-style-type: none"><li>1. <code>eventName</code>: Der Name des Ereignisses als String</li><li>2. <code>listener</code>: Eine Callback-Funktion, die ausgeführt wird, sobald das Ereignis eintritt</li></ol>
<a href="#">emit</a> <a href="#">↗</a>	"Emittiert" ein Ereignis, d.h. alle für das entsprechende Ereignis registrierten Callback-Funktionen werden aufgerufen. Parameter: <ol style="list-style-type: none"><li>1. <code>eventName</code>: Der Name des Ereignisses als String</li><li>2. <code>args</code>: Eine beliebige Anzahl von Argumenten, die den Callback-Funktionen beim Aufruf übergeben werden</li></ol>

# BEISPIEL: EventEmitter

```
// Modul "events" einbinden
const events = require('events');
// EventEmitter erzeugen
const emitter = new events.EventEmitter();

// Callback-Funktion (=Event-Listener) für das Ereignis
// "hello" registrieren
emitter.on('hello', function(name) {
    console.log(`Hallo ${name}!`);
});

// Ereignis "hello" erzeugen ("emittieren"), Argument "WEB1"
// für registrierte Callback-Funktionen mitgeben
emitter.emit('hello', "WEB1");
```

Ausgabe:  
Hallo WEB1!

# EREIGNISSE IM "HTTP"-MODUL

- Das "events"-Modul wird von den meisten anderen Node.js-Modulen verwendet
- Beispiel: Das `Server`-Objekt aus dem "http"-Modul ist ein Event-Emitter und bietet Ereignisse, für die Listener registriert werden können, z.B.:

Ereignis	Bedeutung
<code>request</code>	Eine Anfrage ist eingetroffen
<code>connection</code>	Eine (TCP-)Verbindung mit einem Client wurde aufgebaut
<code>listening</code>	Der Server wurde an einen Port und eine IP-Adresse gebunden und wartet auf eingehende Verbindungen
<code>close</code>	Der Server wird beendet

# BEISPIEL: EREIGNISSE IM "HTTP"-MODUL

Ursprüngliches Beispiel - Event-Listener werden implizit registriert:

```
const http = require("http");

const server =
  http.createServer(function(request, response) {
    response.writeHead(200,
      { "content-type": "text/html; charset=utf-8" });

    const html = `<!DOCTYPE html>
      <html>
        <head>
          <title>Hallo WEB1</title>
          <meta charset="utf-8">
        </head>
        <body>
          <h1>Hallo WEB1!</h1>
        </body>
      </html>`;

    response.end(html);
  });

server.listen(8042, function() {
  console.log("Ich lausche auf http://localhost:8042");
});
```

Explizitere Variante (zur Veranschaulichung der involvierten Ereignisse):

```
const http = require("http");
const server = http.createServer();

// Request-Listener explizit für das "request"-Ereignis
// registrieren
server.on("request", function(request, response) {
  response.writeHead(200,
    { "content-type": "text/html; charset=utf-8" });

  const html = `<!DOCTYPE html>
    <html>
      <head>
        <title>Hallo WEB1</title>
        <meta charset="utf-8">
      </head>
      <body>
        <h1>Hallo WEB1!</h1>
      </body>
    </html>`;

  response.end(html);
});

// Listener explizit für das "listening"-Ereignis
// registrieren
server.on("listening", function() {
  console.log("Ich lausche auf http://localhost:8042");
});

server.listen(8042);
```



```
sven@tsu:~ $ node webServer.js  
Ich lausche nun auf http://localhost:8042  
█
```

Der Server "läuft" und wartet auf ~~bestimmte Ereignisse~~ eintreffende Anfragen.

! Web-Server müssen häufig mit einer großen Zahl gleichzeitiger Anfragen von vielen Clients umgehen.

Wie können wir dies sicherstellen?



## Zentrale Eigenschaften von Node.js:

- Asynchrone (nicht-blockierende) Ein- und Ausgabe
- Ereignisgetrieben
- Modularer Aufbau

# BEHANDLUNG GLEICHZEITIGER ANFRAGEN

## Ansatz 1: Nebenläufige Abarbeitung - Ein Thread pro Anfrage

Unterschied zwischen Prozess und Thread?

### Prozess:

- Repräsentation eines ausgeführten Programmes im Betriebssystem
- Bekommt eigene Ressourcen zugewiesen (Bereich im Arbeitsspeicher, Rechenzeit des Prozessors)

### Thread:

- Ausführungsstrang innerhalb eines Prozesses
- Nutzt die Ressourcen des Prozesses
- Ein Prozess kann aus mehreren Threads bestehen

# ANSATZ 1: EIN THREAD PRO ANFRAGE

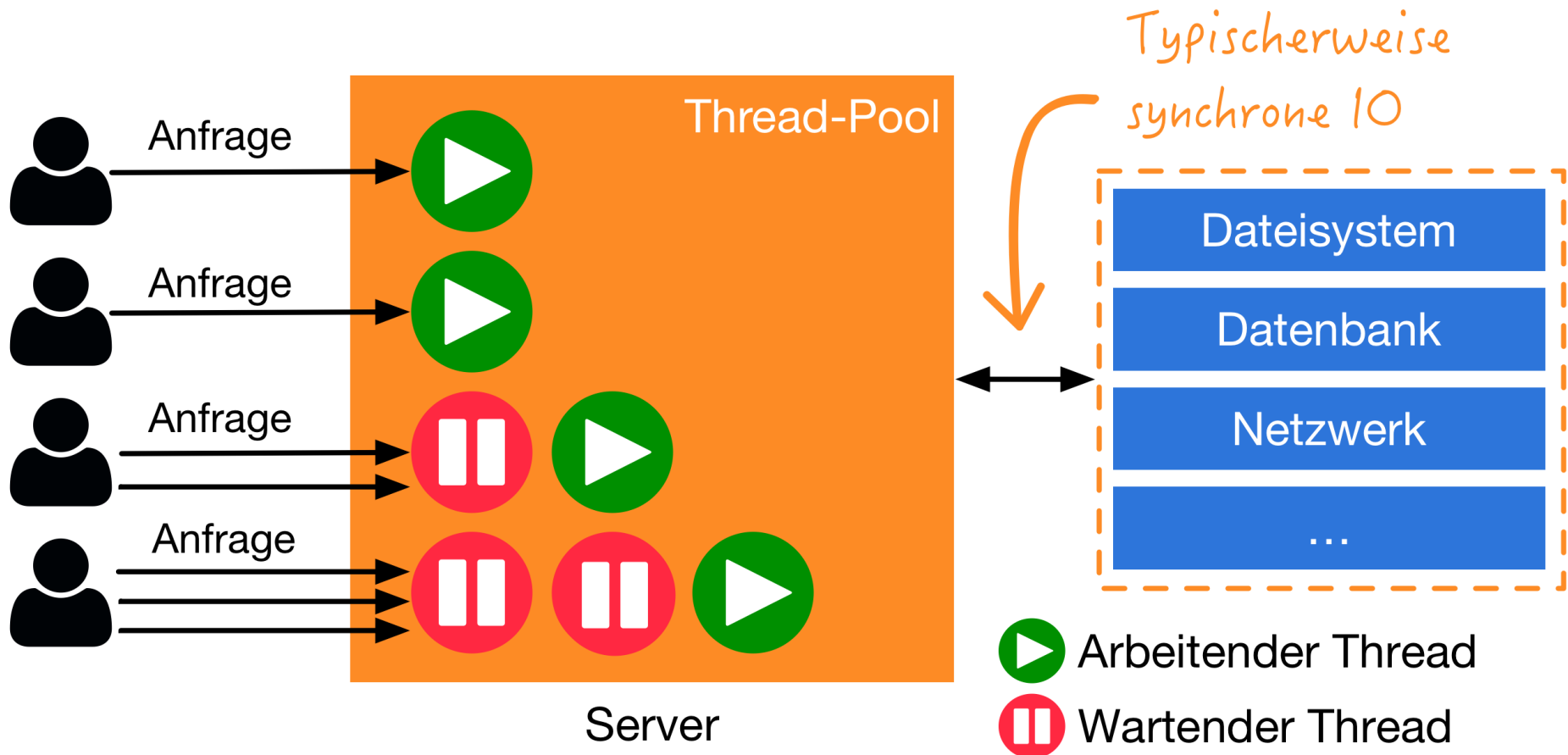


Bild basierend auf: <https://springframework.guru/reactive-streams-in-java/>

# BEHANDLUNG GLEICHZEITIGER ANFRAGEN (2)

## 💡 Ansatz 1: Nebenläufige Abarbeitung - Ein Thread pro Anfrage

- Server verwaltet verfügbare Threads in einem Thread-Pool
- Der Pool ist typischerweise in seiner Größe auf eine maximale Anzahl von Threads beschränkt
- Trifft eine Anfrage ein, so wird diesem ein Thread aus dem Pool zugewiesen
- Der Thread arbeitet die Anfrage ab und steht *nach Abschluss der Abarbeitung* wieder im Pool zur Verfügung

# BEHANDLUNG GLEICHZEITIGER ANFRAGEN (3)

## 💡 Ansatz 1: Nebenläufige Abarbeitung - Ein Thread pro Anfrage

- Der Zugriff auf Ressourcen (z.B. Datenbank) erfolgt in der Regel blockierend - der Thread wartet, bis der Zugriff vollständig erfolgt ist
- Steht kein Thread im Pool mehr zur Verfügung, so müssen ankommende Anfragen warten, bis wieder Threads frei werden

# ANSATZ 1: EIN THREAD PRO ANFRAGE

## Potentielle Nachteile:

- ➖ Synchroner/blockierender IO kann zu schlechter Auslastung des Servers führen (wartende Threads)
- ➖ Bei einer großen Menge von Anfragen können sich diese "aufstauen"

## Beispiele für Server, die diesen Ansatz nutzen:

Apache, Wildfly, Tomcat

# ANSATZ 2: EVENT-LOOP (NODE.JS)

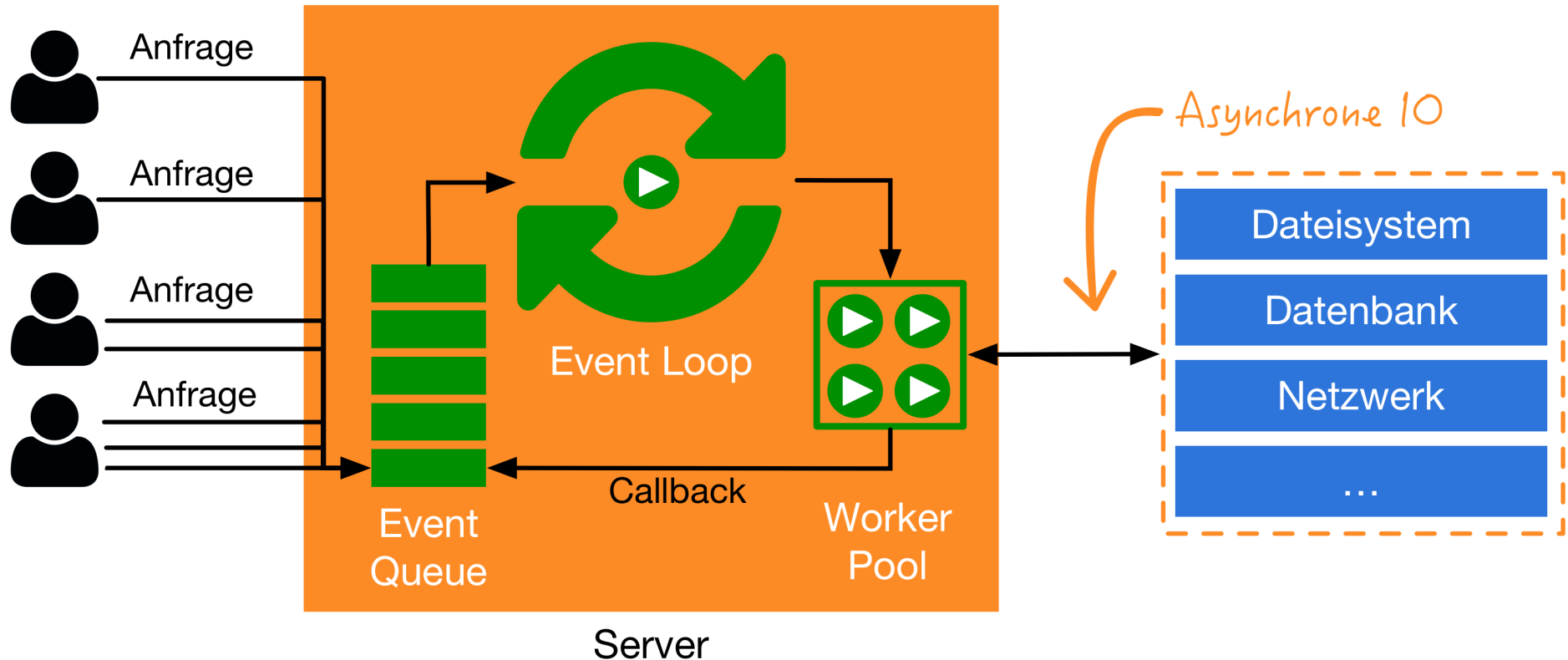


Bild basierend auf: <https://springframework.guru/reactive-streams-in-java/>



# BEHANDLUNG GLEICHZEITIGER ANFRAGEN (4)

## 💡 Ansatz 2: Event-Loop (Node.js)

- Eintreffende Anfragen (Ereignisse) werden in einer Warteschlange eingereiht
- Der Server arbeitet mit einem zentralen *Event-Loop*, der innerhalb *eines* Threads läuft

# BEHANDLUNG GLEICHZEITIGER ANFRAGEN (5)

## 💡 Ansatz 2: Event-Loop (Node.js)

- Der Event-Loop arbeitet kontinuierlich Anfragen aus der Warteschlange ab (er blockiert nicht!)
- Die Abarbeitung der Ereignisse (insbesondere teure Anfragen wie Berechnungen oder Datenbankzugriffe) gibt der Event-Loop an Worker-Threads ab, die den Event-Loop über Callback-Funktionen informieren, wenn sie fertig sind

# ANSATZ 2: EVENT-LOOP (NODE.JS)

## Vorteil:

- ➕ Durch die Nutzung eines Event-Loops kann ein hoher Durchsatz bei der Abarbeitung von Anfragen erzielt werden

## Nachteil:

- ➖ EntwicklerInnen müssen dafür sorgen, dass ihr Code den Event-Loop niemals blockiert

Weitere Beispiele für Server, die diesen Ansatz (bzw. Varianten) nutzen:

Nginx, Netty, Undertow

```
sven@tsu:~ $ node webServer.js  
Ich lausche nun auf http://localhost:8042  
█
```

Der Server "läuft" und wartet auf eintreffende Anfragen.

Wie wir eine Antwort erzeugen, haben wir bereits gesehen  
(→ Response-Objekt).

Wie können wir eintreffende Anfragen verarbeiten?




# ERINNERUNG: REQUEST-LISTENER

- Ein *Request-Listener* ist eine Funktion, die aufgerufen wird, sobald eine Anfrage beim Server eintrifft
- Die Funktion hat zwei Parameter:

Parameter	Zweck
<i>request</i>	Ein Request-Objekt, das die eingetroffene Anfrage repräsentiert
<i>response</i>	Ein Response-Objekt, das die Antwort des Servers repräsentiert

# Request-OBJEKT

- Über die Anfrage teilt der Client dem Server mit, was dieser tun soll
- Das Request-Objekt kapselt alle Daten einer eingegangenen Anfrage
- Für den Zugriff auf die Daten bietet das Objekt u.A. folgende Eigenschaften:

Eigenschaft	Zweck
<a href="#">method</a> 	Die HTTP-Methode aus der Anfragezeile (z.B. GET, POST, DELETE)
<a href="#">url</a> 	Die URL aus der Anfragezeile
<a href="#">headers</a> 	Die HTTP-Header der Anfrage als Objekt (Aufbau analog zum Response-Objekt)

# BEISPIEL: EINFACHES ROUTING VON ANFRAGEN

```
const http = require("http");

const server =
  http.createServer(function(request, response) {
    // URL der Anfrage lesen
    const url = request.url;
    // HTTP-Methode der Anfrage lesen
    const method = request.method;

    // URL und Methode auswerten und Anfrage entsprechend verarbeiten
    // ("Routing", oder auch: "Dispatching")
    if (url === "/") {
      // Antwort für Zugriff auf URL "/" erzeugen
      [...]
    } else if (url.startsWith("/new") && method === "GET") {
      // Antwort für Zugriff auf URL "/new" mit Methode GET erzeugen
      [...]
    } else if (url.startsWith("/new") && method === "POST") {
      // Antwort für Zugriff auf URL "/new" mit Methode POST erzeugen
      [...]
    }
  }).listen(8042, function() {
    console.log("Ich lausche auf http://localhost:8042");
  });
```

# DATEN AUS DEM ANFRAGE-BODY LESEN

- Werden z.B. Daten über ein HTML-Formular per POST-Methode geschickt, so befinden sich diese im Body der HTTP-Anfrage
- Für den Zugriff auf den Body einer Anfrage steht im Request-Objekt jedoch keine Eigenschaft zur Verfügung
- Stattdessen müssen die Daten über einen Datenstrom (*Stream*) gelesen werden



# STREAMS (KURZÜBERBLICK)

- *Streams* dienen zur Verarbeitung von Ein- und Ausgaben
- Ein Stream ist eine kontinuierliche Folge von "Häppchen" (*Chunks*)

## Schematische Darstellung:

Quelle

Konsument

Bild basierend auf: Casciaro M., Mammino L.; Node.js Design Patterns; Packt Publishing; 2019 (S. 141)

# STREAMS (KURZÜBERBLICK)

- *Streams* dienen zur Verarbeitung von Ein- und Ausgaben
- Ein Stream ist eine kontinuierliche Folge von "Häppchen" (*Chunks*)

## Schematische Darstellung:



Bild basierend auf: Casciaro M., Mammino L.; Node.js Design Patterns; Packt Publishing; 2019 (S. 141)

# STREAMS (KURZÜBERBLICK)

- *Streams* dienen zur Verarbeitung von Ein- und Ausgaben
- Ein Stream ist eine kontinuierliche Folge von "Häppchen" (*Chunks*)

## Schematische Darstellung:



Bild basierend auf: Casciaro M., Mammino L.; Node.js Design Patterns; Packt Publishing; 2019 (S. 141)

# STREAMS (KURZÜBERBLICK)

- *Streams* dienen zur Verarbeitung von Ein- und Ausgaben
- Ein Stream ist eine kontinuierliche Folge von "Häppchen" (*Chunks*)

## Schematische Darstellung:

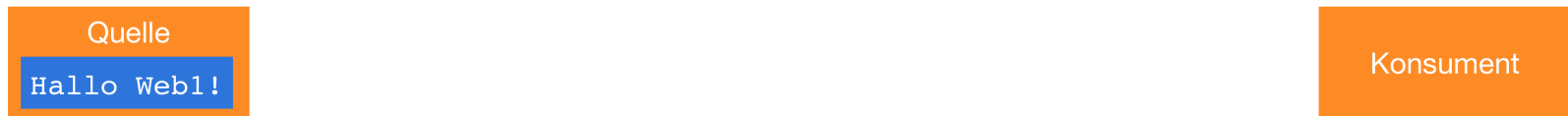


Bild basierend auf: Casciaro M., Mammino L.; Node.js Design Patterns; Packt Publishing; 2019 (S. 141)

# STREAMS (KURZÜBERBLICK)

- *Streams* dienen zur Verarbeitung von Ein- und Ausgaben
- Ein Stream ist eine kontinuierliche Folge von "Häppchen" (*Chunks*)

## Schematische Darstellung:

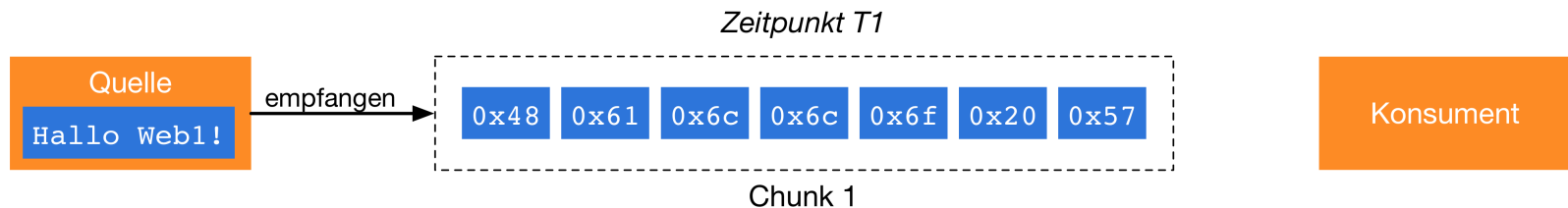


Bild basierend auf: Casciaro M., Mammino L.; Node.js Design Patterns; Packt Publishing; 2019 (S. 141)

# STREAMS (KURZÜBERBLICK)

- *Streams* dienen zur Verarbeitung von Ein- und Ausgaben
- Ein Stream ist eine kontinuierliche Folge von "Häppchen" (*Chunks*)

## Schematische Darstellung:

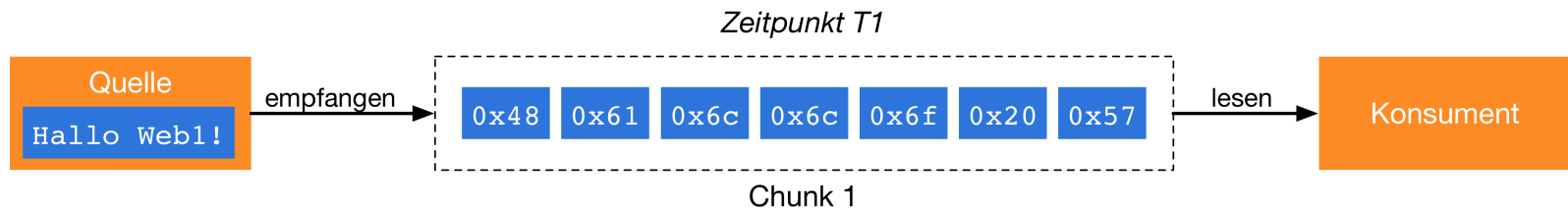


Bild basierend auf: Casciaro M., Mammino L.; Node.js Design Patterns; Packt Publishing; 2019 (S. 141)

# STREAMS (KURZÜBERBLICK)

- *Streams* dienen zur Verarbeitung von Ein- und Ausgaben
- Ein Stream ist eine kontinuierliche Folge von "Häppchen" (*Chunks*)

## Schematische Darstellung:

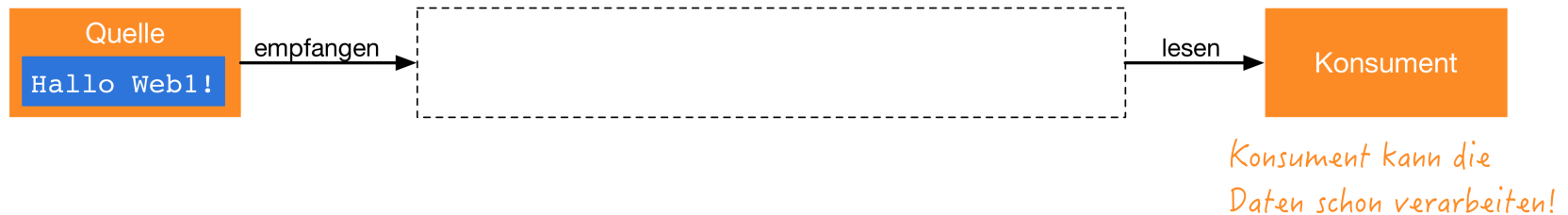


Bild basierend auf: Casciaro M., Mammino L.; Node.js Design Patterns; Packt Publishing; 2019 (S. 141)

# STREAMS (KURZÜBERBLICK)

- *Streams* dienen zur Verarbeitung von Ein- und Ausgaben
- Ein Stream ist eine kontinuierliche Folge von "Häppchen" (*Chunks*)

## Schematische Darstellung:

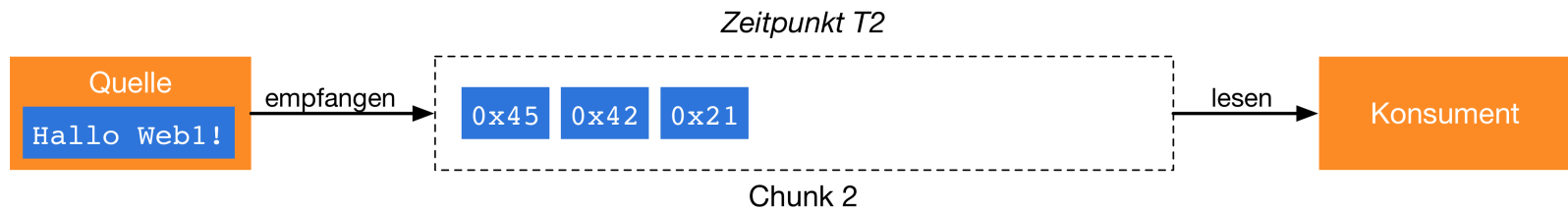


Bild basierend auf: Casciaro M., Mammino L.; Node.js Design Patterns; Packt Publishing; 2019 (S. 141)



# STREAMS (KURZÜBERBLICK)

## Vorteile:

- ➕ Bei größeren Datenmengen kann mit der Verarbeitung begonnen werden, bevor die Daten vollständig vorliegen (höhere Geschwindigkeit, geringere Speicherlast)
- ➕ Streams sind kombinierbar (*pipining*), um z.B. Datentransformationen zu realisieren (z.B. Kompression, Verschlüsselung)

# DATEN AUS DEM ANFRAGE-BODY LESEN

- Das `Request`-Objekt ist ein spezieller Stream: Ein [ReadableStream](#) (d.h. eine Datenquelle, aus der gelesen werden kann)
- Streams sind Event-Emitter, d.h. die Kommunikation mit einem Stream erfolgt über Ereignisse
- Ein `ReadableStream` bietet u.A. folgende Ereignisse:

Ereignis	Bedeutung
<code>readable</code>	Es liegen Daten im Stream vor, die gelesen werden können. Zum Lesen bietet der Stream die Methode <a href="#">read</a>
<code>end</code>	Es liegen keine weiteren Daten mehr im Stream vor
<code>error</code>	Es ist ein Fehler aufgetreten

# BEISPIEL: ANFRAGE-BODY LESEN

```
const http = require("http");

const server = http.createServer(function(request, response) {
  const url = request.url;
  const method = request.method;

  if (url === "/") {
    [...]
  } else if (url.startsWith("/new") && method === "GET") {
    [...]
  } else if (url.startsWith("/new") && method === "POST") {
    let body = "";
    // Listener für das "readable"-Ereignis registrieren
    request.on("readable", function() {
      // Immer wenn Daten verfügbar sind:
      // 1. Diese einlesen
      let data = request.read();
      // 2. Daten in der Variable "body" merken
      body += data !== null ? data : "";
    });
    // Listener für das "end"-Ereignis registrieren
    request.on("end", function() {
      // Sobald keine weiteren Daten verfügbar sind: Verarbeitung der Daten
      // (gespeichert in der "body"-Variable), dann Antwort erzeugen
      [...]
    });
  }
}).listen(8042);
```

## Aufgabe:

Betrachten Sie die bisherigen Code-Beispiele. Welche Nachteile bzw. Probleme sehen Sie, wenn auf diese Weise komplexere Web-Anwendungen mit dem "http"-Modul entwickelt werden sollen?

- ➖ Gesamter Code in lediglich einer Datei
- ➖ Keine Trennung von Zuständigkeiten (Vermischung von Fachlogik, Routing, HTML-Templates, etc)
- ➖ Programmierung auf einem geringen Abstraktionsniveau (sehr "low-level", viel muss manuell programmiert werden)
- ➔ *Bei steigender Komplexität: Schlechte Lesbarkeit, Wartbarkeit und Skalierbarkeit*