Node.js-Tutorium: Hello World: HTTP

Dieser Artikel erfüllt die GlossarWiki-Qualitätsanforderungen **nur teilweise**:

Korrektheit: 4 (großteils überprüft)	**Umfang**: 1 (zu gering)	**Quellenangaben**: 5 (vollständig vorhanden)	**Quellenarten**: 5 (ausgezeichnet)	**Konformität**: 5 (ausgezeichnet)

Node.js-Tutorium Hello World

Übersicht: Teil 1: Konsole | Teil 2: HTTP | Teil 3: TCP

Inhaltsverzeichnis

1 Use Cases

Es soll ein einfacher Node.js-Web-Server erstellt werden, der unter der URL `http://localhost:7777/` eine HTML-Seite mit dem Inhalt `Hallo, Welt!` als Ergebnis liefert.

Auf der Seite soll sich außerdem ein Eingabefeld und ein Butoon befinden. Wenn der Benutzer seinen Namen eingibt und auf den Button klickt, wird er zusätzlich mit seinem Namen begrüßt: `Hallo, <BENUTZERNAME>!`.

2 Ein einfacher HTTP-Server, der `Hallo, Welt!` ausgibt

Erstellen Sie eine Datei `hello-world-http-01.js` und fügen Sie folgenden Code ein:

```
'use strict';

require('http')
  .createServer
    ( function (p_request, p_response)
      { p_response.writeHead(200, {'Content-Type': 'text/html;
charset=utf-8'});
        p_response.write
                   (['<!DOCTYPE html>',
                     '<html>',
                       '<head>',
                         '<title>Hallo-Welt-Server</title>',
                       '</head>',
                       '<body>',
                         '<p>Hallo, Welt!</p>',
                       '</body>',
                     '</html>'
                    ].join('')
                   );
        p_response.end();
      }
    )
  .listen(7777);

console.log("Der Server läuft und lauscht auf Port 7777.");
```

Führen Sie diese Datei in WebStorm oder in der Bash-Konsole auf und öffnen Sie dann in Ihrem Browser die URL `http://localhost:7777/`.

2.1 Analyse von `hello-world-http-01.js`

Die Node-Bibliothek `http` stellt die Methode `createServer` zur verfügung, mit der ein neuer HTTP-Server erstellt werden kann. Ein neu erstellter Server wird gestartet, indem ihm die Nachricht `listen(<POTNUMMER>)` geschickt wird.

Sobald dies geschen ist, horcht der Webserver auf Port `<POTNUMMER>` (in unserem Fall `7777`) auf HHTP-Requests. Sobald ein Browser oder sonst irgend ein Client eine derartige Anfrage schickt, wird die Callback-Funktion aufgerufen, die bei der Definition des Servers angegeben wurde. Diese Funktion muss zwei Parameter haben: `p_request` und `p_response`. Das Objekt `p_request` enthält alle Daten, die der Client an den Server schickt, in das Objekt `p_response` schreibt der Server seine antwort.

Eine Server-Antwort besteht immer aus zwei Teilen: einem HTTP-Header und einem HTTP-Content. Der HTTP-Header enthält Metainformationen:

Den HTTP-Status-Code (z.B. 200 für „OK“ = „Die Anfrage konnte erfolgreich bearbeitet werden und das Anfrage-Ergebnis steh im Contetn-Bereich.“)
Den `Content-Type` (z.B. `text/html; charset=utf-8`)
Die `Content-Length` (die Länge der Antwort im Content-Bereich)

etc.

Der eigentliche Content ist dann eine beliebige Folge von Zeichen, in userem Fall ein HTML-Code. Der Browser liest davon so viele Zeichen, wie im Header unter dem Attribut `Content-Length` angegeben wurden. Diese Attribut wird vom Node-HTTP-Server automatisch ermittelt.

3 Verarbeitung von Benutzerdaten

Um Benutzerdaten verarbeiten zu können, muss im Server auf das Objekt `p_request` zugegriffen werden.

Erstellen Sie eine Datei namens `hello-world-http-02` und fügen Sie folgenden Code ein.

```
'use strict';

var v_http        = require('http'),        // Das HTTP-Server-Paket.
    v_querystring = require('querystring'), // Zur Umwandlung von
Benutzerdaten in JavaScript-Objekte.
    v_documents;                            // Alle HTML-Templates, die
der Server als Antwort schicken kann.

/* Ein HTML-Template ist ein Array, gefüllt mit HTML-Template-Strings.
 * Ein HTML-Template-String ist ein normaler String bestehend aus
 * HTML-Code, der zusätzlich ein oder mehrere Template-Strings
 * der Bauart "{{key}}" enthalten darf. Diese Template-Strings werden
 * von der Methode "m_template_to_html" durch aktuelle Daten
 * ersetzt, bevor der HTML-TExt an den Client (Browser) ausgeliefert
 * wird.
 */
v_documents =
{ '/':
  [ '<!DOCTYPE html>',
    '<html>',
    '  <head>',
    '    <title>Hallo-Server</title>',
    '  </head>',
    '  <body>',
    '    <p>Hallo, Welt!</p>',
    '    <form action="/hallo" method="post">',
    '      <label>Ihr Name: </label>' + '<input type="text"
name="user"/>',
    '      <input type="submit" value="Begrüße mich!">',
    '    </form>',
    '  </body>',
    '</html>'
  ],

  '/hallo':
  [ '<!DOCTYPE html>',
```

```
    '<html>',
    '  <head>',
    '    <title>Hallo-{{user}}-Server</title>',
    '  </head>',
    '  <body>',
    '    <p>Hallo, {{user}}!</p>',
    '  </body>',
    '</html>'
  ],

  'error':
    [ '<!DOCTYPE html>',
      '<html>',
      '  <head>',
      '    <title>Hallo-Server</title>',
      '  </head>',
      '  <body>',
      '    <p><strong>Die angeforderte Seite existiert
nicht.</strong></p>',
      '  </body>',
      '</html>'
    ]
};

/**
 * Extrahiert die Benutzerdaten aus <code>p_request</code>
 * und ruft die Callback-Funktion <code>p_callback</code>
 * auf, sobald dies erledigt ist.  Beim Aufruf wird der
 * Funktion <code>p_callback</code> das erzeugte und bereinigte
 * Datenobjekt als einziges Argument übergeben.
 *
 * @private
 * @param {Object}   p_request  Das Request-Objekt der aktuellen HTTP-
Anfrage.
 * @param {Function} p_callback Die Callback-Funktion, die aufgerufen
wird, sobald
 *                              alle im Request-Objekt enthaltenen Daten
empfangen
 *                              und extrahiert wurden.
 */
function m_get_user_data(p_request, p_callback)
{ var l_data_string = '';

  // Nur Post-Requests dürfen Benutzerdaten enthalten.
  if (p_request.method === 'POST')
  { p_request
      /* Immer, wenn ein Teil der Benutzerdaten angekommen ist,
       * wird der Event 'data' ausgelöst. Die neu angekommen Daten
       * werden zum Daten-String l_data_string hinzugefügt.
       * Wenn der Benutzer mehr als 1000000 Zeichen an Daten schickt,
```

```
       * wird die Verbindung abgebrochen.
       */
      .on('data',
          function (p_data_string)
          { l_data_string += p_data_string;

            // Zu viele Daten => Verbindungsabbruch!
            if (l_data_string.length > 1000000)
            { p_request.connection.destroy(); }
          }
         )
      /* Wenn alle Daten empfangen wurden, werden diese Daten
       * bereinigt (Kleiner- und Größerzeichen werden ersetzt), um
       * Cross-Site-Scripting zu verhindern.
       * Anschließend werden Sie mittels der Callback-Funktion p_callback
       * an den Aufrufer weitergeleitet.
       */
      .on('end',
          function()
          { var l_data = v_querystring.parse(l_data_string);
            for (var k in l_data)
            { if (l_data.hasOwnProperty(k))
              { l_data[k] = l_data[k].replace(new RegExp('<', 'g'),
'&lt;')
                                     .replace(new RegExp('>', 'g'),
'&gt;');
              }
            }
            p_callback(l_data);
          }
         );
  }
  // Bei allen übrigen Requests, insbesondere Get-Requests, werden
Benutzerdaten ignoriert.
  else
  { p_callback(); }
}

/**
 * Ersetzt der Reihe nach in jedem HTML-String des HTML-Dokuments
 * <code>p_document</code> die Template-Parameter <code>{{KEY}}</code>
 * durch <code>p_data['KEY']</code> und reicht danach den modifizierten
 * HTML-String via <code>p_response</code> an den Clientweiter.
 *
 * @private
 * @param {Array}    p_document Ein Array mit HTML-Template-Strings
 * @param {Object}   p_data     Ein Hasharray (Objekt) mit aktuellen
Daten
 * @param {Function} p_response Das Response-Objekt des HTTP-Servers,
 *                              über welches der HTML-Code an den Client
```

```
 *                              geschickt wird.
 */
function m_template_to_html(p_document, p_data, p_response)
{ for (var i = 0, n = p_document.length; i < n; i++)
  { var l_line = p_document[i];
    if (p_data)
    { for (var k in p_data)
      { if (p_data.hasOwnProperty(k))
        { l_line = l_line.replace(new RegExp('{{' + k + '}}', 'g'),
p_data[k]); }
      }
    }
    p_response.write(l_line+'\n');
  }
}

v_http
  /* Der eigentliche HTTP-Server verarbeitet jede Anfrage auf dieselbe
Art und Weise:
   *
   * 1. Ein geeigneter HTTP-Header wird an den Client geschickt.
   * 2. Die Benutzerdaten werden mittels m_get_user_data aus dem Request-
Objekt p_request extrahiert.
   * 3. Sobald das zugehörige Datenobjekt via Callback-Funktion zur
Verfügung steht,
   *    werden im HTML-Template, das der Request-URL zugeordnet ist, die
Template-Parameter
   *    durch die aktuellen (bereinigten) Benutzerdaten ersetzt und das
so entstandene
   *    HTML-Dokument an den Client geschickt.
   */
  .createServer
    ( function(p_request, p_response)
      { var l_document = v_documents[p_request.url] || v_documents.error;
        p_response.writeHead(l_document === v_documents.error ? 404 :
200,
                             {'Content-Type': 'text/html; charset=utf-8'}
                            );
        m_get_user_data
          (p_request,
           function(p_data)
           { m_template_to_html(l_document, p_data, p_response);
             p_response.end();
           }
          );
      }
    )
  /* Der Server lauscht auf Port 7778 auf Benutzeranfragen. */
  .listen(7778);
```

```
console.log("Der Server läuft unter http://localhost:7778/.");
```

Starten Sie das Programm und rufen Sie die URL `http://localhost:7778/` in Ihrem Browser auf. Spielen Sie mit dem Programm, indem Sie Ihrem Namen eingeben, HTML-Code als Namen eingeben (Versuch von Cross-Site-Scripting), bestehende und nicht bestehende URLs von Hand eingeben etc.

3.1 Analyse von `hello-world-http-02.js`

Dies dürfte das längste Hello-World-Programm aller Zeiten sein. Das hat allerdings einen triftigen Grund: Wenn man eine Web-Anwendung erstellt, können potenziell mehr als 7.000.000.000 Benutzer zugreifen. Diese machen Fehler, absichtlich oder unabsichtlich. Dagegen **muss** die Anwendung abgesichert werden. Ein zweiter Grund ist, dass die Kommunikation zwischen Client und Server immer asynchron abläuft.

Das Programm ist prinzipiell wie folgt aufgebaut:

In der Hashmap `v_documents` sind die HTML-Seiten (genauer HTML-Templates) gespeichert, die dem Benutzer portenziell angezeigt werden:

Startseite/Homepage (`/`), die die Welt begrüßt und den Benutzer nach seinem Namen fragt
Begrüßungsseite (`/hallo`), die angezeigt wird, nachdem der Benutzer seinen Namen eingegeben hat
Fehlerseite (`error`), die angezeigt wird, falls der Benutzer eine nicht-existente URL aufruft

Beseer wäre es natürlich, diese Templates nicht direkt im Programm, sondern in separaten HTML-Dateien zu speichern.

Der eigentliche Web-Server (`v_http.createServer(...)`) behandelt jede URL, die ihm vom Client übergeben wird, auf dieselbe Weise:

1. Der Server holt sich das zur vom Client übergebenen URL passende HTML-Template. Fall dies nicht vorhanden ist, holt es das Error-HTML-Template.
2. Er schickt an den Client den HTTP-Header, bestehend aus einem HTTP-Statuscode (`200`: OK, `404`: Datei nicht gefunden) und den Dokumenttyp des Antwortdokuments (`text/html`, UTF8-codiert),
3. Er ruft mit Hilfe der Methode `m_get_user_data` (der Präfix `m_` symbolisiert, dass es sich um eine private MEthode handelt) die vom Client übergebenen Daten ab und bereinigt diese, um Cross-Site-Scripting zu verhindern. Dieser Vorgang läuft asynchron ab, da die Daten portionsweise vom Client zum Server übertragen werden.
4. Sobald alle Benutzerdaten übertragen wurden, ruft `m_get_user_data` eine vom Server bereitgestellte Callbackfunktion auf. Diese Callbackfunktion fügt die Benutzerdaten mittels `m_template_to:html` in das aktuell HTML-Template ein und schickt dieses HTML-Dokument an den Client.
5. Zu guter Letzt schließt der Server die Kommunikation mittels `p_resonse.end()` ab. Das ist für das Serverobjekt der Hinweis, dass er die Gesamtlänge des HTML-Dokuments ermitteln kann, um das HTTP-Header-Feld `Content-Length` zu berechnen und die Daten an den Client auszuliefern.

   Wie man gesehen hat, sind an mehreren Stellen Sicherheitsüberprüfungen eingebaut worden:

   Für nicht-erxistente Seiten wird ein Fehlercode und ein Fehlerdokument als Antwort ausgegeben.
   Die Benutzerdaten werden bereinigt, d.h. HTML-Tags werden neutralisiert, indem die Zeichen < und > durch `<` und `>` ersetzt werden. (Sehen Sie sich mal die HTML-Source von dieser Zeile an. :-)).
   Damit ist es dem Benutzer z.B. nicht mehr nöglich, HTML-Skripte mittels `<script>...</script>` zu schicken, die dann in die Begrüßungsseite eingebaut und beim Client ausgeführt werden.

Es gibt allerdings noch eine weitere Sicherheitsüberprüfung. Die Methode `m_get_user_data` bricht die Verbindung zum Client ab, wenn mehr als 1.000.000 Zeichen übertragen werden sollen. Damit ist es einem Angreifer nicht so leicht möglich, den Server lahmzulegen, indem er einfach ein paar DVD-Inhalte an Daten schickt.

4 Fortsetzung des Tutoriums

Sie sollten nun Teil 3 des Tutoriums bearbeiten.

5 Quellen

Kowarschick (MMProg): Wolfgang Kowarschick; Vorlesung „Multimedia-Programmierung“; Hochschule: Hochschule Augsburg; Adresse: Augsburg; Web-Link; 2018; Quellengüte: 3 (Vorlesung)

6 Siehe auch

Beispiele und Musterlösungen (auf GitHub)

Kategorien:
Node.js-Tutorium: Hello World
Node.js-Beispiele