Web-basierte Systeme

10: Serverseitige Implementierung

Wintersemester 2025

Rüdiger Kapitza

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Vorläufiger Vorlesungsplan

16. Oktober	Einführung und Darstellung von Webseiten
23. Oktober	HTML und CSS
30. Oktober	Hypertext Transfer Protocol
6. November	Browser Schnittstellen
13. November	Kommunikationsschnittstellen im Browser
20. November	WebAssembly
27. November	Architektur moderner Browser
4. Dezember	Clientseitige Architekturmuster und serverseitige
11. Dezember	Implementierung von Web-basierten Systemen Vorbereitung Papieranalyse
8. Januar	Papieranalyse
15. Januar	**Lastverteilung durch Zwischenspeicher**
22. Januar	Aspekte von Web Sicherheit
29. Januar	Web3
5. Februar	Zusammenfassung und Ausblick

Serverseitige Implementierung von Web-basierten Systemen

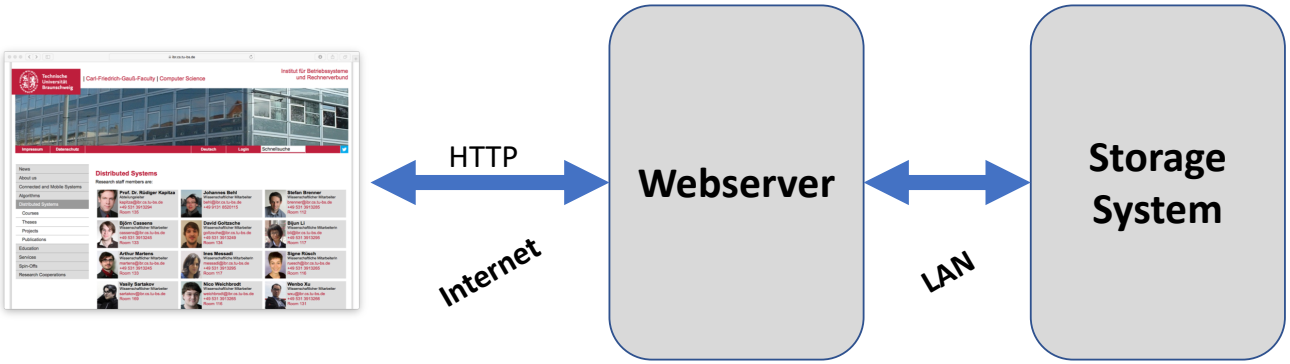
Serverseitige Implementierung

Zielsetzung der Lerneinheit

- Kennenlernen der Basisfunktion & Architektur eines Webservers
- Überblick zu Datenhaltung von Webanwendungen

Serverseitige Implementierung

Basisarchitektur



- Webserver - können grundsätzlich in statisch, statisch aber konfigurierbar und dynamisch unterteilt werden
  - Oft werden statisch & dynamische Webserver kombiniert
- Datenverwaltung - sehr verschiedene Ausprägungen möglich
  - klassische SQL-Datenbanken, NoSQL-DB, Key/Value Speicher aber auch Blob-Speicher und Dateisysteme spielen eine Rolle

Webserver

Webserver

Grundfunktionalität

- Implementiert HTTP und kommuniziert mit dem Browser
- Klassischer Server-Dienst
  - Unterschiedliche Architekturen: Multiprozess, mutithreaded oder Zustandsmaschine sowie Mischformen
- Hauptschleife mit den Aufgaben, eine Verbindung anzunehmen, die Anfrage zu lesen, sie zu verarbeiten, die Antwort zu schreiben und schließlich die Verbindung zu schließen
- Im Falle eines statischen Webservers könnte dies wie folgt aussehen:

```
1  // Answer a single request ...
2  int fd = open("index.html");
3  int len = read(fd, fileContents, sizeOfFile(fd));
4  write(tcpConnection, httpRespHeader, headerSize);
5  write(tcpConnection, fileContents, len);
```

  - Hier ist ein Zugriff auf die Festplatte erforderlich (...aber auch das Dateisystem hat einen Zwischenspeicher)

Webserver

Verwendung des Common Gateway Interface (CGI)

- Anfrage für eine Seite, die dynamisch erstellt werden soll
  - Führt zum Aufruf von `fork()` mit dem entsprechenden Hilfsprogramm und zur Übergabe der Verbindung
  - Das Programm vervollständigt eine Vorlage unter Verwendung von Informationen aus einer Datenbank

Webserver

Frameworks der zweiten Generation

- Der Webserver führt für jede Anfrage ein Programm - den Controller - aus:
  - URL und der HTTP-Request werden geparst, um Parameter für die weitere Verarbeitung zu erhalten
  - Parameter werden verwendet, um das Model der Anwendung beim Datenbanksystem anzufordern
  - Die View-Komponente wird ausgeführt, die unter Verwendung eines Templates und des Models eine HTML-Seite erzeugt

Webserver

Beispiel: Für Rails wird ein Programm pro URL ausgeführt

- URL `/rails_intro/hello`
- Ausführen des `hello.rb`-Controllers, der das Model aus der Datenbank anfordert
- Das Model wird nun mit einem Template zusammengeführt: `hello.html.erb` (Mischung aus HTML und Ruby)
- Welche Rolle spielt hier JavaScript?
  - Nur eine Ressource die eingebunden wird

Webserver

Webserver für JavaScript-Frameworks

- Für die meisten Webanwendungen können im Prinzip einfache Webserver verwendet werden
  - Templates (HTML und CSS)
  - JavaScript
- Weitere Kommunikation zwischen Browser und Server im Kontext von Model-Daten
  - CRUD (Create, Read, Update, Delete) der Model-Daten
  - Zugriff und Modifikation von Sitzungsdaten
- Im Prinzip geringe Anforderungen an die Infrastruktur
  - HTTP GET von statischen Dateien ist dominant
  - Operationen auf den Daten - hier hauptsächlich Interaktion mit der Datenbank

Node.js Motivation und Basiskonzepte

- Man nehme die JavaScript-Engine eines Browsers – v8
  - Eine Sprache auf Client & Server-Seite
- Ergänze Ereignisse und entsprechende Verwaltungsstrukturen
  - Alle Aufrufe werden durch die Hauptereignisschleife behandelt (wie im Browser)
- Ergänzt durch eine Schnittstelle zum Betriebssystem
  - Alle blockierenden Aufrufe müssen gesondert behandelt werden
    - Bspw. Zugriff auf Sockets und Dateien
  - Unterstützung für Ereignisbehandlungsroutinen
- Bereitstellung eines Modulsystems
  - Bspw. die Verwendung von getrennten Sichtbarkeitsbereichen

Webserver

Node.js

Threads

```
1  request = readRequest(socket);
2
3  reply = processRequest(request);
4
5  sendReply(socket, reply);
```

Implementierung

- Operationen sind blockierend
- Es wird ein Scheduler benötigt

Events

```
1  readRequest(socket, function(request) {
2     processRequest(request,
3         function (reply) {
4           sendReply(socket, reply);
5  }); });
```

Implementierung

- Nicht-blockierende und feingranulare Ereignisverarbeitung

Webserver

Node.js

- Interne Severschleife

```
1 while (true) {
2         if (!eventQueue.notEmpty()) {
3            eventQueue.pop().call();
4         }
5 }
```

- Maxime: Niemals warten/blockieren in einer Ereignisbehandlungsroutine
  - Beispiel: `readRequest(socket);`
    1. `launchReadRequest(socket);` (Kehrt sofort zurück!)
    2. Bei Abschluss der Leseoperation: `eventQueue.push(readDoneEventHandler);`

Beispiel: Lesen einer Datei

```
1  var fs = require("fs");
2
3  // fs object wraps OS sync file system calls
4  // OS read() is synchronous but Node's fs.readFile is asynchronous
5  fs.readFile("smallFile", readDoneCallback); // Start read
6
7  function readDoneCallback(error, dataBuffer) {
8  // Node callback convention: First argument is JavaScript Error object
9  // dataBuffer is a special Node Buffer object
10    if (!error)  {
11       console.log("smallFile contents", dataBuffer.toString());
12  } }
```

Thread-basierte versus Ereignis-basierte Programmierung

- Threads: blockieren und warten ist transparent
- Ereignisse: blockieren und warten erfordert Rückrufe
- Daumenregel
  - Falls Code nicht blockiert in beiden Fällen gleich
  - Falls Code blockiert: Einsatz einer Rückrufroutine
    - `return` wird zu einem Funktionsaufruf

Webserver

Express.js - als *minimales* Framework für Node.js

- Bildet eine Schicht über der Node.js-Schnittstelle
  - Unterstützung für HTTP
    - Grundsätzliche Verarbeitung von HTTP-Anfragen – wird schon von Node.js bereitgestellt
  - Routing
    - Abbildung von URLs auf Funktionen
    - Im Kern wird eine Routingtabelle bereitgestellt
  - Middleware-Funktionen
    - Ermöglicht es die Behandlung von Sitzungen, Cookies, Sicherheit, usw. zu ergänzen

Webserver

Minimalbeispiel für Express.js

```
1  var expressApp = express();
2  expressApp.get('/', function(httpRequest,httpResponse){
3    httpResponse.send('hello world');
4  });
5  expressApp.listen(3000);
```

- Express-Anwendung (`expressApp`) stellt Methoden bereit für
  - Routing von HTTP-Anfragen
  - Erzeugen von HTML mittels Templates
  - Schnittstelle zur Konfiguration der Middleware des Preprozessors

Webserver

Routing

- Reihe von Basisroutinen (kleiner Ausschnitt):

```
1 expressApp.get(urlPath, requestProcessFunction);
2 expressApp.post(urlPath, requestProcessFunction);
3 expressApp.put(urlPath, requestProcessFunction);
4 expressApp.delete(urlPath, requestProcessFunction);
5 expressApp.all(urlPath, requestProcessFunction);
```

- `urlPath` kann Parameter enthalten

Webserver

Verarbeiten einer HTTP-Anfrage mit dem httpRequest-Objekt

```
1  expressApp.get('/user/:user_id', function (httpRequest,
       httpResponse))
```

- Objekt mit zahlreichen Eigenschaften
  - Die auch im Kontext der Verarbeitung ergänzt werden (siehe Middleware)
- `request.params` – Parameter der Route (z.B. `user_id`)
- `request.query` – Anfrageparameter (e.g. `&foo=9` `{foo: '9'}`)
- `request.body` – Objekt welches den geparsten Body enthält
- `request.get(field)` – HTTP Header Parameter

Verarbeiten einer HTTP-Antwort mit dem httpResponse object

```
1 expressApp.get('/user/:user_id', function (httpRequest,
      httpResponse) ...
```

- Objekt mit einer Reihe von Methoden zum Setzen von HTTP-Antwortfeldern
  - `response.write(content)` – Erzeugen der Antwort
  - `response.status(code)` – HTTP Code setzten
  - `response.set(prop, value)` – Header-Werte setzten
  - `response.end()` – Beenden und absenden der Antwort
  - `response.send(content)` – Äquivalent zu `write()` & `end()`
- Zusammensetzten einer Antwort

```
1 response.status(code).write(out1).write(out2).end();
```

Middleware

- Wichtig, um sich flexible in die Anfrageverarbeitung einzublenden

```
1  expressApp.all(urlPath, function (request, response, next) {
2      // Do whatever processing on request (or setting response)
3      next(); // pass control to the next handler
4  });
```

- oder Anfragen zu bearbeiten, die den Routing-Mechanismus durchlaufen:

```
1  expressApp.use(function (request, response, next) {...});
```

- Beispiele
  - Überprüfen ob ein Benutzer authentifiziert ist
  - Parsen und konvertieren der Anfrage (bspw. JSON) und hinzufügen des Ergebnisses an `request.body`
  - Sitzungsverwaltung, Cookie-Mgt., Kompression, Verschlüsselung etc.

Datenhaltung

Datenhaltung

Storage Systeme für Webanwendungen

- Hohe Verfügbarkeit - Daten sind zugreifbar und aktualisierbar
  - Muss skalierbar sein und einen nebenläufigen Zugriff ermöglichen
  - Möglichst fehlertolerant – Datenverlust ist fatal
- Ermöglicht es Anwendungsdaten wohlstrukturiert abzulegen
  - Schneller Zugriff auf die Model-Daten einer View
  - Flexible Weiterentwicklung der Anwendung bzw. des Datenmodels wird unterstützt
- Einfache Verwendbarkeit von zentraler Bedeutung

Relationale Datenbanken

- Relationale Datenbanken haben sich gegenüber einer Reihe früherer Ansätze durchgesetzt
- Minimale Wiederholung
  - Daten werden in Tabellen (den Relationen) abgelegt
  - Jede Zeile (Tupel) in einer Tabelle ist ein Datensatz (record)
  - Jedes Tupel besteht aus einer Reihe von Attributwerten (Attribute = Eigenschaften), den Spalten der Tabelle
- Das Datenbankschema beschreibt die Struktur der Datenbank
- Zugriff erfolgt über die *Structured Query Language (SQL)*
  - Sprache mit vielen Optionen, um die richtigen Daten zu erheben
  - Grundidee: Benutzer/in sagt, was er/sie sucht, und das Datenbanksystem bestimmt, wie dies effizient zu tun ist

Objektrelationale Abbildung

- Es hat sich gezeigt, dass ein relationales Datenmodell und der Entwicklungsprozess von Web-Anwendungen nicht immer kompatibel sind
  - Wie passen Objekte und Tabellen zusammen?
  - Evolution von Webanwendungen lässt sich nur schwer auf eine relationale Datenbank abbilden
  - Als Konsequenz wurde eine automatisierte Abbildung von Objekten auf SQL Datenbanken durchgeführt
  - Beispiel für Rails: Active Records
    - Objekte werden auf Datensätze abgebildet
    - Pro Klasse eine Tabelle (als Models in Rails)
    - Objekte sind damit als Zeile in einer Tabelle repräsentiert
    - Attribute sind als Spalten abgebildet
  - Unterstützung zur Schemaerzeugung und Evolution sowie die notwendigen SQL-Operationen für Objekte werden bereitgestellt

NoSQL Datenbanken

- Als weiterer Evolutionsschritt haben sich NoSQL-Datenbanken etabliert, die dem objekt- und dokumentenbasierten Modell heutiger Webanwendungen besser entsprechen
- Prominentes Beispiel: MongoDB
  - Datenmodel besteht aus Sammlungen von Dokumenten (JSON Objekten)
  - Verfügt über eine ausdrucksstarke Anfragesprache
  - Bietet Indizes für einen schnellen Zugriff
  - Zielt auch auf Skalierbarkeit und Zuverlässigkeit ab

Schemafrei oder nicht?

- Das Speichern und Verwalten von JSON-Objekten ist sehr flexibel, aber das ist nicht immer wünschenswert
  - Beispiel: `<h1>Hello {{person.informalName}}</h1>`
  - Dies funktioniert gut, wenn `person.informalName` eine Zeichenkette ist, die eine bestimmte Länge nicht überschreitet
- Wenn man nun ein Schema etablieren will, kann dies mit Hilfe von Validatoren für modifizierende Operationen erfolgen.
- Als Alternative kann man bspw. Mongoose mit der zugehörigen Object Definition Language (ODL) verwenden
  - Stellt Konzepte aus dem Kontext des objekt-relationalen Mappings als einfachere API dar

Serverseitige Implementierung

Zusammenfassung

- Webserver zentrales Element: statisch versus dynamisch
- Datenhaltung: Dialog zwischen Flexibilität und Schema
- Nur an der Oberfläche gekratzt
  - Aspekte wie Aufteilung zwischen statischen und dynamischen Webservern nicht betrachtet
  - Abwägung zwischen verschiedenen Systemen zur Datenhaltung wurde nicht vertieft