

Web-basierte Systeme

09: Serverseitige Implementierung

Wintersemester 2024

Rüdiger Kapitza



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



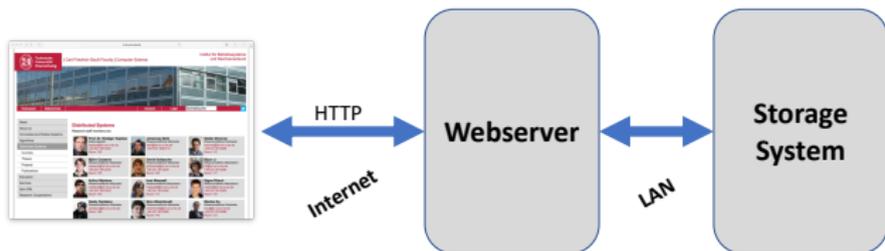
Friedrich-Alexander-Universität
Technische Fakultät

Serverseitige Implementierung von Web-basierten Systemen

Zielsetzung der Lerneinheit

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

Basisarchitektur



- Webserver - können in Prinzip in statisch, statisch aber konfigurierbar und dynamisch unterteilt werden.
 - Oft werden statisch und dynamische Webserver miteinander kombiniert
- Storage Systeme - sehr verschiedene Ausprägungen möglich
 - klassische **SQL-Datenbanken**, **NoSQL-Datenbanken**, Key/Value Stores aber auch Blob-Storage und Dateisysteme spielen eine Rolle

Webserver

Grundfunktionalität

- Implementiert HTTP und kommuniziert mit dem Browser
- Klassischer Server-Dienst
 - Unterschiedliche Architekturen: Prozess-, Thread-basiert oder Zustandsmaschine sowie Mischformen
- Hauptschleife mit der Aufgaben: eine Verbindung anzunehmen, die Anfrage zu lesen, sie zu verarbeiten, das Schreiben der Rückantwort und abschließend beenden der Verbindung
- Im Falle eines statischen Webservers sieht das ungefähr wie folgt aus:

```
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 nötig (...aber auch das Dateisystem hat einen Zwischenspeicher)

Verwendung des Common Gateway Interface (CGI)

- Anfrage nach einer dynamisch zu erstellenden Seite (bspw. index.php)
 - Erwirkt intern ein `fork()` mit dem entsprechenden Hilfsprogramm und es erfolgt eine Übergabe der Verbindung zum Client
 - Programm ergänzt ein Template und greift dazu auf Informationen aus der Datenbank zurück

Frameworks der zweiten Generation

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

Beispiel: Für Rails wird ein Programm pro URL ausgeführt

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

Webserver für JavaScript-Frameworks

- Für die meisten Webanwendungen können im Prinzip einfache Webserver verwendet werden
 - Templates (HTML und CSS)
 - JavaScript Dateien
- 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-Laufzeitumgebung eines Browsers – v8
 - Eine Sprache auf Client & Server-Seite
- Ergänze Ereignisse und entsprechende Verwaltungsstrukturen
 - Alle Aufrufe werden durch die Hauptereignisschleife behandelt (wie im Browser)
- Ergänze eine Schnittstelle zum Betriebssystem
 - Alle blockierenden Aufrufe müssen behandelt werden (Zugriff auf Sockets und Dateien)
 - Unterstützung für Ereignisbehandlungsroutinen
- Bereitstellung eines vernünftigen Modulsystems
 - Bspw. die Verwendung von getrennten Sichtbarkeitsbereichen

Node.js

Threads

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

Implementierung

- Operationen sind blockierend
- Es wird Scheduler benötigt

Events

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

Implementierung

- Nicht-blockierend und feingranulare Verarbeitung von Ereignissen

Node.js

■ Interne Severschleife

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

■ Maxime: Niemals warten oder 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 Rückrufoutine
 - ...return wird zu einem Funktionsaufruf

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

- Stellt eine dünne Schicht oberhalb der Node.js-Schnittstelle bereit
 - 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

Express.js Minimalbeispiel

```
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 zum konfigurieren der Middleware des Preprozessors

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

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 geparsen 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) ...
```

- Object with a number of methods for setting HTTP response fields
 - `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()` und `end()`
- Zusammensetzen einer Antwort

```
1 response.status(code).write(content1).write(content2).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 aber nur Anfragen die durch den Routingmechanismus laufen:

```
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

Storage Systeme für Webanwendungen

- Hohe Verfügbarkeit - Daten sind zugreifbar und können aktualisiert werden
 - 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
 - Eine Weiterentwicklung der Anwendung und es Datenmodels wird unterstützt
- Wie immer ist eine einfache Verwendbarkeit von zentraler Bedeutung

Relationale Datenbanken

- Relationale Datenbanken haben sich gegen eine ganze Reihe von früheren Ansätzen durchgesetzt
- Minimale Wiederholung (siehe RDB 1)
 - 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 Nutzer sagt was er sucht und das Datenbanksystem bestimmt wie das effizient zu tun ist

Objektrelationale Abbildung

- Mit der Zeit hat sich herausgestellt das ein relationales Datenmodell und Webanwendungen nicht immer zusammenpassen
 - Wie passen Objekte und Tabellen zusammen?
 - Evolution von Webanwendungen kann nur schwierig auf eine relationale Datenbank abgebildet werden
 - Als Konsequenz wurde eine automatisierte Abbildung von Objekten auf SQL Datenbanken durchgeführt
 - Beispiel: 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 nötigen SQL Operation für Objekte werden bereitgestellt

NoSQL Datenbanken

- Als weiterer Evolutionsschritt haben sich die NoSQL-Datenbanken etabliert die besser zum Objekt/Dokument-basierten Model aktueller Webanwendungen passen
- Prominentes Beispiel: MongoDB
 - Datenmodel besteht aus Sammlungen von Dokumenten (JSON Objekten)
 - Verfügt über eine ausdrucksstarke Anfragesprache
 - Stellt Indices für schnellen Zugriff bereit
 - Zielt auch auf Skalierbarkeit und Zuverlässigkeit ab

Schemafrei oder nicht?

- 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` einer Zeichenkette entspricht und eine gewisse Größe nicht übersteigt
- Wenn man nun ein Schema etablieren will geht dies im Prinzip mittels Validatoren für modifizierende Operationen
- Als Alternative kann man bspw. Mongoose mit der zugehörigen Object Definition Language (ODL) verwenden
 - Bildet Konzepte aus dem Kontext der objektrelationalen Abbildung auf eine einfachere API ab.

Zusammenfassung

- Webserver zentrales Element: statisch versus dynamisch
 - Unklar wohin die Reise geht ...
- Datenhaltung auch im Dialog zwischen Flexibilität und Bedarf nach 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
 - Usw.