Skalierbarer, serverseitiger Code in JavaScript

Einführung in Node.js

Tom Hughes-Croucher
& Mike Wilson
Übersetzung von Thomas Demmig

O'REILLY®

Inhalt

KAPITEL 4

Core-APIs

Node besitzt viele APIs, aber manche sind wichtiger als andere. Diese Core-APIs bilden das Rückgrat jeder Node-Anwendung, und Sie werden feststellen, dass Sie sie immer wieder verwenden werden.

Events

Die erste API, die wir uns anschauen werden, ist die `Events`-API. Der Grund hierfür liegt darin, dass sie zwar abstrakt, aber ein fundamentaler Bestandteil ist, die von jeder anderen API zum Funktionieren benötigt wird. Wenn Sie diese API verstanden haben, werden Sie alle anderen effektiver nutzen können.

Wenn Sie jemals JavaScript für Ihren Browser programmiert haben, dann haben Sie auch schon Events verwendet. Allerdings stammt das Event-Modell im Browser mehr vom DOM denn von JavaScript selbst, und viele der Konzepte rund um das DOM haben außerhalb dieses Kontexts keinen Sinn. Lassen Sie uns das DOM-Modell für Events anschauen und mit der Implementierung in Node vergleichen.

Das DOM besitzt ein benutzergetriebenes Event-Modell, das auf der Benutzerinteraktion basiert, wobei eine Gruppe von Oberflächen-Elementen in einer Baumstruktur angeordnet ist (HTML, XML und so weiter). Das bedeutet, dass es bei einer Interaktion des Benutzers mit der Oberfläche ein Event und einen dazugehörigen Kontext gibt – das HTML/XML-Element, auf dem der Klick oder eine andere Aktivität ausgeführt wurde. Dieser Kontext besitzt ein Elternelement und potenzielle Kindelemente. Da der Kontext eine Baumstruktur ist, gibt es im Modell die Konzepte des Aufsteigens und Abfangens von Events, wodurch Elemente ober- oder unterhalb im Baum das ausgelöste Event empfangen können.

So kann zum Beispiel in einer HTML-Liste ein click-Event auf ein `<li>` von einem Listener am Elternelement `<ul>` abgefangen werden. Umgekehrt kann ein Klick auf das `<ul>` zu einem Listener des `<li>` weitergereicht werden. Da JavaScript-Objekte diese Art von Baumstruktur nicht besitzen, ist das Modell in Node viel einfacher.

EventEmitter

Da das Eventmodell in den Browsern mit dem DOM verbunden ist, wurde in Node die Klasse `EventEmitter` erstellt, um eine grundlegende Event-Funktionalität anbieten zu können. Die gesamte Event-Funktionalität in Node dreht sich um `EventEmitter`, da diese auch als Interface-Klasse entworfen ist, mit der sich andere Klassen erweitern lassen. Es wäre sogar unüblich, eine Instanz von `EventEmitter` direkt aufzurufen.

`EventEmitter` besitzt eine Handvoll Methoden, von denen die beiden wichtigsten `on` und `emit` sind. Die Klasse stellt diese Methoden für die Verwendung durch andere Klassen bereit. Die Methode `on` erzeugt einen Event Listener für ein Event, wie in Übung 4-1 zu sehen ist.

Übung 4-1: Mit der Methode on auf ein Event lauschen

```
server.on('event', function(a, b, c) {
  //tue etwas
});
```

Die Methode `on` erwartet zwei Parameter: den Namen des Events, auf das gelauscht werden soll, und die Funktion, die aufzurufen ist, wenn das Event ausgelöst wurde. Da es sich bei `EventEmitter` um eine Interface-Pseudoklasse handelt, sollte eine Klasse, die von `EventEmitter` erbt, mit dem Schlüsselwort `new` aufgerufen werden. Schauen Sie sich Übung 4-2 an, um zu sehen, wie wir eine neue Klasse als Listener erstellen.

Übung 4-2: Eine neue Klasse erstellen, die Events mit EventEmitter unterstützt

```
var util = require('util'),
    EventEmitter = require('events').EventEmitter;

var Server = function() {
  console.log('Initialisieren');
};

util.inherits(Server, EventEmitter);

var s = new Server();

s.on('abc', function() {
  console.log('abc');
});
```

In diesem Beispiel nehmen wir als erstes das Modul `util` auf, so dass wir die Methode `inherits` verwenden können. Mit `inherits` erhält die Klasse `EventEmitter` eine Möglichkeit, ihre Methoden der von uns erzeugten Klasse `Server` mitzugeben. So können alle neuen Instanzen von `Server` als `EventEmitter` verwendet werden.

Dann nehmen wir das Modul `events` auf. Allerdings benötigen wir nur Zugriff auf die Klasse `EventEmitter` innerhalb dieses Moduls. Beachten Sie die Großschreibung von `EventEmitter`, anhand derer zu erkennen ist, dass es sich um eine Klasse handelt. Wir haben keine Methode `createEventEmitter` verwendet, da wir nicht vorhaben, einen `Event-`

Emitter direkt zu nutzen. Wir wollen nur seine Methoden zur Klasse Server hinzufügen, die wir noch erzeugen werden.

Wenn wir die notwendigen Module eingebunden haben, erstellen wir als nächstes unsere einfache Klasse Server. Diese stellt nur eine simple Funktion bereit, die beim Initialisieren eine Nachricht protokolliert. In einer echten Implementierung würden wir den Prototyp der Server-Klasse mit den Funktionen versehen, die wir einsetzen wollten. Der wichtige Schritt ist das Verwenden von util.inherits, um unsere Server-Klasse mit der Superklasse EventEmitter zu versehen.

Wenn wir die Klasse Server nutzen wollen, instanziieren wir sie per new Server(). Diese Instanz von Server hat Zugriff auf die Methoden der Superklasse (EventEmitter), womit wir unserer Instanz mit Hilfe der Methode on einen Listener hinzufügen können.

Bis jetzt würde der von uns hinzugefügte Event Listener allerdings nie aufgerufen werden, weil das Event abc nie gefeurt wird. Das können wir ändern, indem wir den Code in Übung 4-3 ergänzen, um das Event per emit auszulösen.

Übung 4-3: Ein Event auslösen

```
s.emit('abc');
```

Zum Aufrufen des Event Listeners muss nun nur die Methode emit genutzt werden, die die Server-Instanz von EventEmitter geerbt hat. Beachten Sie, dass diese Events Instanz-basiert sind. Es gibt keine *globalen* Events. Wenn Sie die Methode on aufrufen, verbinden Sie sich mit einem bestimmten Objekt, das auf EventEmitter basiert. Selbst verschiedene Instanzen der Klasse Server nutzen Events nicht gemeinsam. s aus dem Code in Übung 4-3 kennt die Events einer anderen Instanz von Server (zum Beispiel erstellt durch var z = new Server();) nicht.

Callback-Syntax

Ein wichtiger Teil bei der Arbeit mit Events ist der Umgang mit Callbacks. In Kapitel 3 schauen wir uns die beste Vorgehensweise im Detail an, aber hier soll es um die Funktionsweise von Callbacks in Node gehen. Diese nutzen ein paar Standard-Muster, aber zuerst wollen wir zeigen, was überhaupt möglich ist.

Beim Aufruf von emit können Sie neben dem Eventnamen noch eine beliebig lange Liste mit Parametern übergeben. In Übung 4-4 sehen Sie drei solcher Parameter. Diese werden an die Funktion weitergereicht, die auf das Event lauscht. Wenn Sie zum Beispiel ein request-Event vom http-Server empfangen, erhalten Sie zwei Parameter: req und res. Als das request-Event ausgelöst wurde, wurden diese Parameter als zweites und drittes Argument an emit übergeben.

Übung 4-4: Parameter beim Auslösen eines Events übergeben

```
s.emit('abc', a, b, c);
```

Es ist wichtig, zu verstehen, wie Node die Event Listener aufruft, da dies Auswirkungen auf Ihren Programmierstil hat. Wird `emit()` mit Argumenten aufgerufen, dann wird der Code in Übung 4-5 verwendet, um jeden Event Listener aufzurufen.

Übung 4-5: Event Listener per emit aufrufen

```
if (arguments.length <= 3) {
  // schnell
  handler.call(this, arguments[1], arguments[2]);
} else {
  // langsamer
  var args = Array.prototype.slice.call(arguments, 1);
  handler.apply(this, args);
}
```

Dieser Code nutzt beide JavaScript-Methoden zum Aufrufen einer Funktion. Wenn `emit()` drei oder weniger Argumente übergeben werden, nutzt diese Methode die schnellere Variante `call`. Sonst greift sie auf das langsamere `apply` zurück, um alle Argumente als `array` zu übergeben. Merken Sie sich dabei, dass Node bei beiden Aufrufvarianten das Argument `this` direkt verwendet. Das bedeutet, der Kontext, in dem die Event Listener aufgerufen werden, ist der Kontext von `EventEmitter` – *nicht* der ursprüngliche Kontext. Mit Node REPL können Sie sehen, was passiert, wenn etwas per `EventEmitter` aufgerufen wird (Übung 4-6).

Übung 4-6: Die durch EventEmitter ausgelöste Kontextänderung

```
> var EventEmitter = require('events').EventEmitter,
...     util = require('util');
>
> var Server = function() {};
> util.inherits(Server, EventEmitter);
> Server.prototype.outputThis= function(output) {
...   console.log(this);
...   console.log(output);
... };
[Function]
>
> Server.prototype.emitOutput = function(input) {
...   this.emit('output', input);
... };
[Function]
>
> Server.prototype.callEmitOutput = function() {
...   this.emitOutput('innerEmitOutput');
... };
[Function]
>
> var s = new Server();
> s.on('output', s.outputThis);
{ _events: { output: [Function] } }
> s.emitOutput('outerEmitOutput');
{ _events: { output: [Function] } }
outerEmitOutput
```

```
> s.callEmitOutput();
{ _events: { output: [Function] } }
innerEmitOutput
> s.emit('output', 'Direct');
{ _events: { output: [Function] } }
Direct
true
>
```

In diesem Beispiel wird zunächst eine Klasse `Server` eingerichtet. Sie erhält Funktionen, um das Event `output` auszulösen. Die Methode `outputThis` wird dem Event `output` als Event Listener zugeordnet. Wenn wir per `emit` das Event `output` in unterschiedlichen Kontexten aufrufen, bleiben wir im Scope des Objekts `EventEmitter`, daher ist der Wert von `this`, auf den `s.outputThis` Zugriff hat, derjenige, der zum `EventEmitter` gehört. Somit muss die Variable `this` als Parameter übergeben und einer Variablen zugewiesen werden, wenn wir sie in Callback-Funktionen eines Events verwenden wollen.

HTTP

Einer der wichtigsten Einsatzbereiche von Node.js ist die Arbeit als Webserver. Dies ist ein so zentraler Teil des Systems, dass Ryan Dahl beim Start des Projekts den HTTP-Stack von V8 umgeschrieben hat, um ihn in einen nicht-blockierenden zu ändern. Auch wenn sich sowohl die API als auch die Interna der ersten HTTP-Implementierung seitdem sehr geändert haben, sind die zentralen Aufgaben immer noch die gleichen. Die Node-Implementierung von HTTP ist nicht-blockierend und schnell. Ein Großteil des Codes wurde von C nach JavaScript verschoben.

HTTP nutzt ein in Node häufig verwendetes Muster. Pseudoklassen-Fabriken bieten eine einfache Möglichkeit, einen neuen Server zu erstellen.[1] Die Methode `http.createServer()` liefert uns eine neue Instanz der HTTP-`Server`-Klasse. Bei dieser handelt es sich um die Klasse, mit der wir die Aktionen definieren, die beim Empfang eintreffender HTTP-Requests von Node ausgeführt werden sollen. Es gibt noch ein paar weitere wichtige Elemente im HTTP-Modul und in anderen Node-Modulen. Dabei handelt es sich um die Events, die die Klasse `qServer` auslöst, und die Datenstrukturen, die den Callbacks übergeben werden. Wenn Sie diese drei Typen von Klassen kennen, können Sie das HTTP-Modul sinnvoll einsetzen.

HTTP-Server

Die Verwendung als HTTP-Server ist vermutlich das häufigste Einsatzgebiet für Node. In Kapitel 1 richten wir einen HTTP-Server ein und nutzen ihn, um eine sehr einfache Response zu liefern. Aber HTTP hat noch viel mehr Facetten. Die Server-Komponente des HTTP-Moduls bietet die grundlegenden Tools, um komplexe und umfassende Webserver

1 Wenn wir von einer *Pseudoklasse* sprechen, beziehen wir uns auf die Definition aus Douglas Crockfords *JavaScript: The Good Parts* (O'Reilly). Ab jetzt werden wir »Klasse« sagen, wenn wir uns auf eine »Pseudoklasse« beziehen.

zu erstellen. In diesem Kapitel werden wir die Grundlagen für den Umgang mit Requests und Responses beschreiben. Auch wenn Sie letztlich einen Server wie Express nutzen, der auf einer höheren Ebene betrieben wird, sind viele der Konzepte Erweiterungen dessen, was hier definiert wurde.

Wie wir schon gesehen haben, muss zunächst ein neuer Server definiert werden. Dazu nutzt man die Methode `http.createServer()` . Diese liefert eine neue Instanz der Klasse `Server` zurück, die nur ein paar Methoden besitzt, da ein Großteil der Funktionalität über Events bereitgestellt wird. Die Klasse `http` für Server besitzt sechs Events und drei Methoden. Dabei werden die Methoden vor allem zum Initialisieren des Servers genutzt, während die Events während der Laufzeit des Servers Verwendung finden.

Lassen Sie uns mit dem einfachsten und kürzesten HTTP-Server-Code beginnen (siehe Übung 4-7).

Übung 4-7: Ein einfacher und sehr kleiner HTTP-Server

```
require('http').createServer(function(req,res){res.writeHead(200, {});
res.end('Hallo Welt');}).listen(8125);
```

Dieses Beispiel ist *kein* guter Code, aber es zeigt ein paar wichtige Punkte auf. Wir werden den Code später noch überarbeiten. Zuerst wird hier das Modul `http` per `require` eingebunden. Beachten Sie, wie wir Methoden verketten können, um auf das Modul zuzugreifen, ohne es zuvor einer Variablen zuweisen zu müssen. Viele Codeelemente in Node liefern eine Funktion zurück,[2] womit wir diese direkt aufrufen können. Aus dem eingebundenen Modul `http` rufen wir `createServer` auf. Dabei benötigen wir nicht zwingend ein Argument, aber wir übergeben eine Funktion, die an das Event `request` gebunden wird. Schließlich weisen wir den Server, den wir per `createServer` erstellt haben, an, auf den Port 8125 zu lauschen.

Wir hoffen, dass Sie bei produktiv genutzten Programmen niemals solchen Code wie in diesem Beispiel schreiben, aber er zeigt die Flexibilität der Syntax und die potenziell mögliche Kürze der Sprache. Lassen Sie unseren Code nun viel expliziter gestalten. In Übung 4-8 sollten wir ihn viel besser verstehen und warten können.

Übung 4-8: Ein einfacher, aber deutlich verständlicherer HTTP-Server

```
var http = require('http');
var server = http.createServer();
var handleReq = function(req,res){
  res.writeHead(200, {});
  res.end('Hallo Welt');
};
server.on('request', handleReq);
server.listen(8125);
```

Dieses Beispiel implementiert ebenfalls den minimalen Webserver. Aber wir haben die Objekte benannten Variablen zugewiesen. Dadurch lässt sich der Code nicht nur leichter

2 Das funktioniert in JavaScript, weil es First-Class-Funktionen unterstützt.

lesen als in der verketteten Variante, sondern Sie können ihn auch besser wiederverwenden. So ist es zum Beispiel nicht unüblich, `http` in einer Datei mehrfach zu nutzen. Wenn Sie gleichzeitig einen HTTP-Server und einen HTTP-Client benötigen, ist eine Wiederverwendung des Modulobjekts sehr hilfreich. Auch wenn JavaScript Sie nicht gerade dazu zwingt, sich über den Speicherverbrauch Gedanken machen zu müssen, sollten Sie nicht gedankenlos überall Objekte herumliegen lassen. Anstatt einen anonymen Callback zu verwenden, haben wir auch der Funktion, die das Event `request` bearbeitet, einen Namen gegeben. Dabei geht es weniger um den Speicherverbrauch als um eine bessere Lesbarkeit. Wir sagen nicht, dass Sie keine anonymen Funktionen verwenden sollen, aber wenn Sie Ihren Code so gestalten, dass man die einzelnen Elemente besser findet, hilft das bei der Wartung enorm.

In Teil I finden Sie mehr zum Programmierstil, speziell in den Kapiteln 1 und 2.

Da wir den Event Listener für `request` nicht an die Fabrikmethode für das `http.Server`-Objekt übergeben haben, müssen wir ihn explizit hinzufügen. Dazu reicht ein Aufruf der Methode `on` von `EventEmitter`. Schließlich rufen wir noch wie im vorigen Beispiel die Methode `listen` zusammen mit dem Port auf, auf den wir lauschen wollen. Die Klasse `http` bietet noch weitere Funktionen, aber dieses Beispiel enthält schon die allerwichtigsten.

Der `http`-Server unterstützt eine Reihe von Events, die entweder mit der TCP- oder HTTP-Verbindung zum Client zu tun haben. Die Events `connection` und `close` werden beim Aufbau beziehungsweise bei der Trennung einer TCP-Verbindung zum Client ausgelöst. Denken Sie daran, dass manche Clients HTTP 1.1 nutzen, das ein Keepalive unterstützt. Deren TCP-Verbindungen bleiben daher eventuell über mehrere HTTP-Requests hinweg geöffnet.

Die Events `request`, `checkContinue`, `upgrade` und `clientError` sind für HTTP-Requests gedacht. Wir haben schon das Event `request` genutzt, um über einen neuen HTTP-Request informiert zu werden.

Das Event `checkContinue` weist auf ein spezielles Ereignis hin. Es ermöglicht Ihnen, eine direktere Kontrolle über einen HTTP-Requests zu erhalten, in dem der Client Datenhäppchen an den Server streamt. Schickt der Client Daten an den Server, wird geprüft, ob er fortfahren kann – dann wird dieses Event ausgelöst. Wenn dafür ein Event Handler registriert wird, dann wird das Event `request` *nicht* ausgelöst.

Das `upgrade`-Event wird ausgelöst, wenn ein Client ein Protokoll-Upgrade anfordert. Der `http`-Server weist HTTP-Upgrade-Requests ab, sofern es keinen Event Handler für dieses Event gibt.

Schließlich leitet das `clientError`-Event alle Fehler-Ereignisse weiter, die vom Client geschickt wurden.

Der HTTP-Server kann ein paar Events werfen. Das wichtigste ist `request`, aber Sie können auch Events erhalten, die mit der `TCP`-Verbindung zum Request oder mit anderen Elementen des Request-Lebenszyklus verbunden sind.

Wenn für einen Request ein neuer TCP-Stream erzeugt wird, dann wird ein Event `connection` geworfen. Dieses Event übergibt den TCP-Stream für den Request als Parameter. Der Stream ist auch für jeden Request in der Variablen `request.connection` zu finden. Allerdings wird für jeden Stream nur ein `connection`-Event gefeuert. Es können also viele `requests` durchgeführt werden, wobei eventuell trotzdem nur ein `connection`-Event ausgelöst wird.

HTTP-Clients

Node lässt sich auch wunderbar für ausgehende HTTP-Verbindungen einsetzen. Das ist in vielen Situationen nützlich, wie zum Beispiel beim Verwenden von Webservices, beim Verbinden zu Dokumenten-Datenbanken oder einfach zum Auslesen von Websites. Sie können für die HTTP-Requests das gleiche `http`-Modul nutzen, sollten aber die Klasse `http.ClientRequest` verwenden. Es gibt zwei Fabrikmethoden für diese Klasse: eine allgemeiner einsetzbare und eine einfachere, dafür aber bequemere Methode. Lassen Sie uns die allgemein einsetzbare Methode in Übung 4-9 anschauen.

Übung 4-9: Einen HTTP-Request erstellen

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

var opts = {
  host: 'www.google.com'
  port: 80,
  path: '/',
  method: 'GET'
};

var req = http.request(opts, function(res) {
  console.log(res);
  res.on('data', function(data) {
    console.log(data);
  });
});

req.end();
```

Das Erste, was Sie hier an Neuem sehen, ist ein `options`-Objekt, das eine ganze Menge an Funktionalität für den Request definiert. Wir müssen den `host`-Namen definieren (auch wenn eine IP-Adresse ausreicht), den `port` und den `path`. Die `method` ist optional, der Standardwert ist dann `GET`. In dem Beispiel geben wir also an, dass es sich bei dem Request um einen `HTTP GET`-Request an `http://www.google.com/` und den Port `80` handelt.

Als Nächstes nutzen wir das `options`-Objekt, um mit der Fabrikmethode `http.request()`eine Instanz von `http.ClientRequest` zu erzeugen. Diese Methode erwartet ein `options`-Objekt und ein optionales Callback-Argument. Der übergebene Callback lauscht

auf das response-Event . Wenn ein solches Event eintrifft, können wir das Ergebnis des Requests verarbeiten. Im Beispiel geben wir einfach das Response-Objekt an der Konsole aus. Aber Sie sollten sich darüber bewusst sein, dass der Body der HTTP-Response über einen Stream im response-Objekt empfangen wird. Sie können sich also auch auf das data-Event des response-Objekts registrieren, um die Daten direkt nach dem Eintreffen zu erhalten (mehr Informationen dazu finden Sie in Abschnitt »Lesbare Streams« auf Seite 71).

Der letzte wichtige Punkt besteht darin, dass wir den Request per end() abschließen müssen. Da es sich um einen GET-Request handelte, haben wir keine Daten an den Server geschickt, aber bei anderen HTTP-Methoden wie PUT oder POST ist das eventuell notwendig. Bis zum Aufruf der end()-Methode löst request den HTTP-Request nicht aus, da es nicht weiß, ob es immer noch auf Daten warten soll.

HTTP GET-Requests abschicken

Da GET bei HTTP so häufig genutzt wird, gibt es eine spezielle Fabrikmethode, um die Verwendung zu vereinfachen (siehe Übung 4-10).

Übung 4-10: Einfache HTTP GET-Requests

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

var opts = {
  host: 'www.google.com'
  port: 80,
  path: '/',
};

var req = http.get(opts, function(res) {
  console.log(res);
  res.on('data', function(data) {
    console.log(data);
  });
});
```

Dieses Beispiel von http.get() macht genau das gleiche wie das vorige Beispiel, aber es ist kürzer. Wir brauchen das method-Attribut des Konfigurationsobjekts nicht und auch der Aufruf von request.end() ist hier überflüssig, weil er schon enthalten ist.

Wenn Sie die beiden vorigen Beispiele ausführen, erhalten Sie pure Buffer-Objekte zurück. Wie später in diesem Kapitel noch beschrieben werden wir, handelt es sich bei einem Buffer um eine spezielle in Node definierte Klasse, die das Speichern von beliebigen, binären Daten ermöglicht. Es ist natürlich möglich, direkt mit dieser zu arbeiten, aber häufig soll eine bestimmte Kodierung genutzt werden, wie zum Beispiel UTF-8 (eine Kodiervariante für Unicode-Zeichen). Sie können dies mit der Methode response.setEncoding() festlegen (siehe Übung 4-11).

Übung 4-11: Die reine Buffer-Ausgabe mit der bei einer angegebenen Kodierung vergleichen

```
> var http = require('http');
> var req = http.get({host:'www.google.com', port:80, path:'/'}, function(res) {
... console.log(res);
... res.on('data', function(c) { console.log(c); });
... });
> <Buffer 3c 21 64 6f 63 74 79 70

...

65 2e 73 74>
<Buffer 61 72 74 54 69

...

69 70 74 3e>

>
> var req = http.get({host:'www.google.com', port:80, path:'/'}, function(res) {
... res.setEncoding('utf8');
... res.on('data', function(c) { console.log(c); });
... });
> <!doctype html><html><head><meta http-equiv="content-type

...

load.t.prt=(f=(new Date).getTime());
})();
</script>

>
```

Im ersten Fall übergeben wir `ClientResponse.setEncoding()` nicht und erhalten daher Datenpakete in `Buffern`. Auch wenn die Ausgabe hier gekürzt ist, sehen Sie, dass es sich nicht nur um einen einzelnen `Buffer` handelt, sondern dass eine Reihe von `Buffern` mit Daten zurückgegeben wurden. Im zweiten Beispiel werden die Daten als UTF-8 zurückgegeben, weil wir `res.setEncoding('utf8')` definiert haben. Die vom Server zurückgelieferten Datenpakete sind immer noch die gleichen, aber das Programm erhält sie in korrekter Kodierung als `strings` und nicht als reine `Buffer`. Auch wenn das in der Ausgabe nicht deutlich zu sehen ist – es gibt einen `string` für jeden der ursprünglichen `Buffer`.

Daten für HTTP POST und PUT hochladen

Bei HTTP geht es aber nicht nur um `GET`. Sie müssen vielleicht auch andere HTTP-Methoden wie `POST` oder `PUT` aufrufen, die zum Ändern von Daten auf dem Server gedacht sind. Der Request ähnelt dabei stark dem `GET`-Request, nur schreiben Sie hier Daten in den *Upstream*, wie Sie in Übung 4-12 sehen können.

Übung 4-12: Daten auf den Server schreiben

```
var options = {
  host: 'www.example.com',
  port: 80,
  path: '/submit',
  method: 'POST'
};

var req = http.request(options, function(res) {
  res.setEncoding('utf8');
  res.on('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });
});

req.write("meine Daten");
req.write("mehr von meinen Daten");

req.end();
```

Dieses Beispiel ähnelt Übung 4-10, nutzt aber die Methode `http.ClientRequest.write()`. Sie ermöglicht es Ihnen, Daten an den Server zu schicken, wobei allerdings wie schon erwähnt explizit `http.ClientRequest.end()` aufgerufen werden muss, wenn Sie alle Daten verschickt haben. Immer dann, wenn `ClientRequest.write()` aufgerufen wird, werden Daten an die Gegenstelle geschickt (sie werden also nicht gepuffert), aber der Server wird erst reagieren, wenn `ClientRequest.end()` aufgerufen wurde.

Sie können Daten an einen Server mit Hilfe von `ClientRequest.write()` streamen, indem Sie die Schreibvorgänge an das `data`-Event eines `Stream` binden. Das ist ideal, wenn Sie zum Beispiel eine Datei von der Festplatte per HTTP an einen Server schicken müssen.

Das ClientResponse-Objekt

Das `ClientResponse`-Objekt enthält eine Reihe von Informationen über die Response. Der größte Teil ist ziemlich klar. Zu einigen der offensichtlichen Eigenschaften, die häufig von Nutzen sind, gehören `statusCode` (mit dem HTTP-Status) und `header` (mit dem Response-Header-Objekt). Auch sehr hilfreich sind eine Reihe von Streams und Eigenschaften, mit denen Sie direkt arbeiten können (wenn Sie wollen) .

URL

Das `URL`-Modul enthält Tools für den einfachen Umgang mit URL-Strings. Es ist ausgesprochen nützlich, wenn Sie mit URLs arbeiten müssen. Das Modul enthält drei Methoden: `parse`, `format` und `resolve`. Schauen wir uns zuerst Übung 4-13 an, in dem die Methode `parse` in Node REPL vorgestellt wird.

Übung 4-13: Eine URL mit dem Modul URL parsen

```
> var URL = require('url');
> var myUrl = "http://www.nodejs.org/some/url/?with=query&param=that&are=awesome
```

```
#alsoahash";
> myUrl
'http://www.nodejs.org/some/url/?with=query&param=that&are=awesome#alsoahash'
> parsedUrl = URL.parse(myUrl);
{ href: 'http://www.nodejs.org/some/url/?with=query&param=that&are=awesome#alsoahash'
, protocol: 'http:'
, slashes: true
, host: 'www.nodejs.org'
, hostname: 'www.nodejs.org'
, hash: '#alsoahash'
, search: '?with=query&param=that&are=awesome'
, query: 'with=query&param=that&are=awesome'
, pathname: '/some/url/'
}
> parsedUrl = URL.parse(myUrl, true);
{ href: 'http://www.nodejs.org/some/url/?with=query&param=that&are=awesome#alsoahash'
, protocol: 'http:'
, slashes: true
, host: 'www.nodejs.org'
, hostname: 'www.nodejs.org'
, hash: '#alsoahash'
, search: '?with=query&param=that&are=awesome'
, query:
   { with: 'query'
   , param: 'that'
   , are: 'awesome'
   }, pathname: '/some/url/'
}
>
```

Erst einmal müssen wir natürlich das Modul `URL` einbinden. Beachten Sie, dass die Namen von Modulen im Quellcode immer in Kleinbuchstaben angegeben werden müssen. Dann haben wir eine URL als String erstellt, die alle möglichen Komponenten enthält. Das Parsen ist nun sehr einfach: Wir rufen die Methode `parse` des `URL`-Moduls mit dem String als Parameter auf. Uns wird dann eine Datenstruktur mit den Teilen der geparsten URL zurückgeliefert. Die Komponenten dieser Struktur sind folgende:

- `href`
- `protocol`
- `host`
- `auth`
- `hostname`
- `port`
- `pathname`
- `search`
- `query`
- `hash`

`href` enthält die vollständige URL, die an `parse` übergeben wurde. `protocol` ist das in der URL genutzte Protokoll (`http://`, `https://`, `ftp://` und so weiter). `host` ist der vollständig

qualifizierte Hostname der URL. Dabei kann es sich um so etwas einfaches wie den Hostnamen eines lokalen Servers handeln, wie zum Beispiel `printserver`, oder um einen vollständig qualifizierten Domainnamen wie `www.google.com`. Auch eine Portnummer wie `8080` kann enthalten sein oder sogar ein Benutzername und ein Passwort wie in `un:pw@ftpserver.com`. Die verschiedenen Teile des Hostnamens sind noch einmal in `auth` für die Benutzerdaten, `port` für den Port und `hostname` für den eigentlichen Hostnamen aufgesplittet. Beachten Sie, dass sich in `hostname` der vollständige Hostname befindet – einschließlich der Top-Level Domain (TLD, zum Beispiel `.com`, `.net` und so weiter) – und der spezifische Server. Wenn die URL zum Beispiel `http://sport.yahoo.com/nhl` lautet, finden Sie in `hostname` nicht nur die Second Level Domain (`yahoo.com`) oder nur den Host (`sport`), sondern den gesamten Namen (`sport.yahoo.com`). Das `URL`-Modul enthält keine Methode, um den Hostnamen weiter in seine Komponenten aufzuteilen, wie zum Beispiel in Domain und TLD.

Die nächste Gruppe an URL-Komponenten bezieht sich auf alles, was nach dem Host folgt. Der `pathname` ist der gesamte Dateipfad nach dem `host`. In `http://sports.yahoo.com/nhl` ist dies `/nhl`. Die nächste Komponente ist `search`, in der die `HTTP GET`-Parameter der URL abgelegt sind. Wenn die URL zum Beispiel `http://mydomain.com/?foo=bar&baz=qux` lautet, enthält die `search`-Komponente `?foo=bar&baz=qux`. Beachten Sie, dass sie das `?` am Anfang enthält. Der `query`-Parameter ähnelt der `search`-Komponente. Sein Inhalt ist abhängig vom Aufruf von `parse`.

`parse` erwartet zwei Argumente: den `url`-String und einen optionalen Booleschen Wert, der festlegt, ob der Query-String mit Hilfe des Moduls `querystring` geparst werden sollte (siehe nächster Abschnitt). Ist das zweite Argument `false`, enthält `query` nur einen String, der dem aus `search` ähnelt – nur ohne das führende `?`. Wenn Sie keinen Wert für das zweite Argument übergeben, ist der Standardwert `false`.

Die letzte Komponente ist der `fragment`-Teil der URL. Dies ist der Teil der URL, der auf das `#` folgt. Meist wird er verwendet, um auf benannte Anker in HTML-Seiten zu verweisen. So kann sich `http://abook.com/#chapter2` zum Beispiel auf das zweite Kapitel auf einer Webseite beziehen, die ein komplettes Buch enthält. Die `hash`-Komponente würde in diesem Fall den Wert `#chapter2` enthalten. Auch hier achten Sie bitte darauf, dass das `#` im String enthalten ist. Manche Sites, wie zum Beispiel `http://twitter.com`, verwenden komplexere Fragmente für Ajax-Anwendungen, aber auch hier gelten die gleichen Regeln. So enthält die URL für den Twitter-Account *mentions* (`http://twitter.com/#!/mentions`) einen `pathname` mit dem Wert `/`, aber einen Hash mit dem Wert `#!/mentions`.

querystring

Das Modul `querystring` ist ein sehr einfaches Hilfsmodul, das den Umgang mit Query-Strings erleichtert. Wie schon im vorigen Abschnitt besprochen handelt es sich bei Query-Strings um Parameter, die am Ende einer URL kodiert sind. Wenn man sie aber nur als normale JavaScript-Strings erhält, sind sie nicht ganz so einfach zu handhaben. Das Modul `querystring` bietet eine einfache Möglichkeit, Objekte aus den Query-Strings

zu erstellen. Die wichtigsten Methoden, die das Modul bereitstellt, sind `parse` und `decode`, aber es gibt auch ein paar interne Hilfsmethoden, unter anderem `escape`, `unescape`, `unescapeBuffer`, `encode` und `stringify`, die das Modul nach außen zur Verfügung stellt. Wenn Sie einen Query-String haben, können sie ihn mit `parse` in ein Objekt umwandeln (siehe Übung 4-14).

Übung 4-14: Einen Query-String mit dem Modul querystring in Node REPL parsen

```
> var qs = require('querystring');
> qs.parse('a=1&b=2&c=d');
{ a: '1', b: '2', c: 'd' }
>
```

Hier wandelt die Klassenfunktion `parse` den Query-String in ein Objekt um, dessen Eigenschaften und Werte den Schlüsseln und Werten im Query-String entsprechen. Sie sollten allerdings auf ein paar Dinge achten. Zum einen werden Zahlen als Strings zurückgegeben, nicht als echte Zahlenwerte. Da JavaScript nur lose typisiert ist und einen String in einer numerischen Operation selbst in eine Zahl umwandelt, funktioniert das ziemlich gut. Denken Sie trotzdem daran, wenn diese Umwandlung einmal nicht funktioniert.

Zum anderen müssen Sie darauf achten, dass Sie den Query-String ohne das führende `?` übergeben, das ihn vom Rest der URL abgrenzt. Eine typische URL sieht zum Beispiel so aus: `http://www.bobsdiscount.com/?item=304&location=san+francisco`. Der Query-String beginnt mit einem `?`, um ihn vom Dateipfad abzutrennen, aber wenn Sie das `?` mit an `parse` übergeben, wird der erste Schlüssel mit einem `?` beginnen – was ziemlich sicher nicht das ist, was Sie möchten.

Diese Bibliothek ist in vielen Situationen nützlich, da Query-Strings auch außerhalb von URLs verwendet werden. Wenn Sie Inhalte über einen `HTTP POST` erhalten, die per `x-form-encoded` kodiert wurden, liegen diese ebenfalls im Query-String-Format vor. Alle Browserhersteller haben dieses Vorgehen als Standard umgesetzt, und Formulare in HTML verschicken Daten an den Server ebenfalls auf diese Art und Weise.

Das Modul `querystring` wird zudem als Hilfsmodul für das Modul `URL` genutzt. Beim Dekodieren von URLs können Sie `URL` sogar direkt auffordern, den Query-String in ein Objekt umzuwandeln, anstatt nur den String zurückzugeben. Das ist detaillierter im vorigen Abschnitt beschrieben und das Parsen erfolgt dabei mit der `parse`-Methode aus `querystring`.

Ein weiterer wichtiger Teil von `querystring` ist `encode` (Übung 4-15). Diese Funktion erwartet ein Objekt mit Schlüssel/Wert-Paaren und wandelt es in einen String um. Das ist besonders nützlich, wenn Sie mit `HTTP`-Requests arbeiten, vor allem bei `POST`-Daten. So können Sie mit einem JavaScript-Objekt arbeiten, bis Sie die Daten durch die Leitung schicken müssen und es dafür einfach kodieren. Sie können jedes beliebige JavaScript-Objekt nutzen, aber idealerweise sollten Sie ein Objekt verwenden, das nur die Daten enthält, die Sie haben wollen, weil die Methode `encode` alle Eigenschaften des Objekts hinzufügt. Handelt es sich bei einer Eigenschaft allerdings nicht um einen String, einen Booleschen Wert oder eine Zahl, wird sie nicht serialisiert und der Schlüssel nur mit einem leeren Wert aufgenommen.

Übung 4-15: Ein Objekt in einen Query-String umwandeln

```
> var myObj = {'a':1, 'b':5, 'c':'cats', 'func': function(){console.log('dogs')}}
> qs.encode(myObj);
'a=1&b=5&c=cats&func='
>
```

I/O

I/O (also Ein- und Ausgabe) ist einer der Hauptbereiche, der Node von anderen Frameworks unterscheidet. Dieser Abschnitt führt Sie durch die APIs, die nicht blockierenden I/O in Node bereitstellt.

Streams

Viele Komponenten in Node stellen eine kontinuierliche Ausgabe bereit oder können einen kontinuierlich eintreffenden Datenstrom verarbeiten. Damit sich diese Komponenten konsistent verhalten, stellt die `stream`-API eine abstrakte Schnittstelle bereit. Diese API bietet allgemeine Methoden und Eigenschaften, die in spezifischen Implementierungen von Streams zur Verfügung stehen. Streams können lesbar, schreibbar oder beides sein. Alle Streams sind Instanzen von `EventEmitter`, so dass sie Events werfen können.

Lesbare Streams

Bei der API für lesbare Streams handelt es sich um eine Reihe von Methoden und Events, die Zugriff auf Datenblöcke bieten, wenn diese Blöcke von den zugrunde liegenden Datenquellen eintreffen. Bei lesbaren Streams geht es vor allem um das Auslösen von `data`-Events. Diese Events repräsentieren den Datenstrom als einen Strom aus Events. Um damit besser umgehen zu können, besitzen Streams eine Reihe von Features, mit denen Sie einstellen können, wie viel Daten Sie erhalten und wie schnell dies geschehen soll.

Der einfache Stream in Übung 4-16 liest nur Daten in Blöcken aus einer Datei. Jedes Mal, wenn ein neuer Block zur Verfügung steht, wird dieser einem Callback in einer Variablen namens `data` bereitgestellt. In diesem Beispiel protokollieren wir die Daten einfach an der Konsole. In realen Einsatzfällen werden Sie die Daten entweder an eine andere Stelle weiterstreamen oder sie in größeren Einheiten sammeln, bevor Sie sie weiterverarbeiten. Das `data`-Event ermöglicht also nur den Zugriff auf die Daten, und Sie müssen sich selbst überlegen, was Sie dann damit anfangen wollen.

Übung 4-16: Einen lesbaren Datenstream erzeugen

```
var fs = require('fs');
var filehandle = fs.readFile('data.txt', function(err, data) {
  console.log(data)
});
```

Lassen Sie uns eines der häufig genutzen Muster bei der Arbeit mit Streams genauer anschauen. Das *Spooling-Muster* wird verwendet, wenn eine Ressource erst vollständig

zur Verfügung stehen muss, bevor wir mit ihr arbeiten können. Wir wissen, dass es wichtig ist, die Eventschleife von Node nicht zu blockieren, damit die Performance nicht leidet. Wir können also erst weiterarbeiten, wenn alles vorhanden ist. In diesem Szenario (Übung 4-17) nutzen wir daher einen Stream, um die Daten *abzurufen*, aber wir *verarbeiten* sie erst, wenn genug Daten vorhanden sind. Das geschieht meist, wenn der Stream beendet wurde, der Zeitpunkt kann aber auch durch ein anderes Event oder eine andere Bedingung bestimmt werden.

Übung 4-17: Einen kompletten Stream mit dem Spooling-Muster einlesen

```
          //abstrakter Stream
var spool = "";
stream.on('data', function(data) {
  spool += data;
});
stream.on('end', function() {
  console.log(spool);
});
```

Filesystem

Das Filesystem-Modul ist offensichtlich sehr nützlich, da Sie es für den Zugriff auf Dateien auf der Festplatte benötigen. Es versucht, beim Dateizugriff möglichst einen POSIX-Stil beizubehalten. Dabei handelt es sich um ein mehr oder weniger einzigartiges Modul, da alle Methoden in asynchronen und synchronen Versionen vorliegen. Aber wir legen Ihnen sehr ans Herz, die asynchronen Methoden zu verwenden, sofern Sie nicht gerade Befehlszeilenskripten mit Node bauen. Selbst dann ist es häufig besser, die asynchronen Versionen zu verwenden, auch wenn Sie dann ein wenig mehr Code schreiben müssen. Denn dadurch können Sie auf mehrere Dateien gleichzeitig zugreifen und die Laufzeit Ihres Skrips verringern.

Das Hauptproblem, dem sich die Leute bei der Arbeit mit asynchronen Aufrufen gegenübersehen, ist die Reihenfolge, was für die Datei-I/O in besonderem Maße gilt. Häufig möchte man eine Reihe von Datei-Verschiebungen, -Umbenennungen, -Kopieraktionen und -Leseoperationen gleichzeitig ausführen. Aber wenn eine dieser Operationen von einer anderen abhängt, kann das zu Problemen führen, weil die Rückkehr-Reihenfolge der Methoden nicht garantiert ist. So kann die erste Operation im Code eventuell erst nach der zweiten abgeschlossen werden. Es gibt Entwurfsmuster, mit denen eine garantierte Reihenfolge ermöglicht wird. Wir haben darüber schon in Kapitel 3 gesprochen, wollen aber hier nochmals darauf eingehen.

Stellen Sie sich vor, Sie wollen eine Datei erst lesen und dann löschen (Übung 4-18). Geschieht das Löschen (unlink) vor dem Lesen, dann werden Sie den Inhalt der Datei nicht mehr auslesen können.

Übung 4-18: Asynchrones Lesen und Löschen einer Datei – Falscher Weg

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

fs.readFile('warandpeace.txt', function(e, data) {
  console.log('Krieg und Frieden: ' + data);
});

fs.unlink('warandpeace.txt');
```

Beachten Sie, dass wir die asynchronen Methoden verwenden. Wir haben dabei zwar Callbacks geschrieben, aber nirgendwo definiert, in welcher Reihenfolge sie aufgerufen werden. Das ist häufig für Programmierer ein Problem, die die Arbeit mit Eventschleifen nicht gewohnt sind. Oberflächlich sieht der Code korrekt aus und manchmal wird er auch funktionieren – manchmal aber auch nicht. Wir brauchen also ein Muster, in dem wir die Reihenfolge der Aufrufe festlegen. Es gibt dabei ein paar mögliche Vorgehensweisen. Eine besteht darin, verschachtelte Callbacks zu verwenden. In Übung 4-19 wird der asynchrone Aufruf zum Löschen der Datei in der asynchronen Funktion eingebettet, die die Datei liest.

Übung 4-19: Asynchrones Lesen und Löschen einer Datei mit verschachtelten Callbacks

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

fs.readFile('warandpeace.txt', function(e, data) {
  console.log('Krieg und Frieden: ' + data);
  fs.unlink('warandpeace.txt');
});
```

Dieses Vorgehen ist für eine begrenzte Menge an Operationen sehr effektiv. In unserem Beispiel mit zwei Operationen ist der Code gut lesbar und verständlich, aber das Muster kann potenziell auch aus dem Ruder laufen.

Buffer

Node arbeitet zwar mit JavaScript, aber es setzt JavaScript nicht in der gewohnten Umgebung ein. So braucht der Browser zum Beispiel JavaScript zum Ausführen vieler Funktionen, das Bearbeiten binärer Daten gehört eher selten dazu. JavaScript unterstützt zwar bitweise Operationen, aber es besitzt keine native Repräsentation binärer Daten. Zusammen mit den Einschränkungen des JavaScript-Typsystems für Zahlen ergibt das eine ausgesprochen unerfreuliche Kombination, da man Binärdaten ansonsten auf diesem Umweg hätte repräsentieren können. Node führt die Klasse `Buffer` ein, um all diese Probleme zu umgehen, wenn Sie mit Binärdaten arbeiten.

Buffer sind eine Erweiterung zur V8-Engine und bringen somit ihre eigenen Probleme mit sich. Buffer ermöglichen tatsächlich ein direktes Zuweisen von Speicher. Das kann sehr große oder sehr kleine Auswirkungen haben – abhängig davon, wie groß Ihre Erfahrungen mit Sprachen sind, die den Computer auf sehr niedriger Ebene ansprechen können. Anders als die Datentypen in JavaScript, die einige der Unschönheiten beim Speichern

von Daten wegabstrahieren, stellt `Buffer` einen direkten Speicherzugriff mit all seinen Vor- und Nachteilen bereit. Wenn ein `Buffer` einmal erstellt wurde, hat er eine feste Größe. Wenn Sie Daten hinzufügen möchten, dann müssen Sie den `Buffer` in einen größeren `Buffer` klonen. Auch wenn manche dieser Features frustrierend sein können, ermöglichen Sie `Buffer`, so schnell arbeiten zu können, wie es für viele Datenoperationen auf dem Server notwendig ist. Es war eine bewusste Designentscheidung, bei der klar war, dass man aus Gründen der Performance die Bequemlichkeit von Programmierern einschränkt.

Ein schneller Überblick zu Binärdaten

Wir haben uns gedacht, dass es sehr hilfreich ist, hier einen schnellen Überblick zur Arbeit mit Binärdaten zu geben. So können Programmierer, die noch nicht viel mit solchen Dateien gearbeitet haben, oder bei denen das schon lange her ist (wie es bei uns während der Anfänge von Node der Fall war), besser einsteigen. Computer funktionieren – wie fast jeder weiß – dadurch, dass sie mit den Status »an« und »aus« arbeiten. Wir nennen dies einen »*binären Status*«, da es nur diese beiden Möglichkeiten gibt. Alles andere beim Umgang mit Computern baut darauf auf, daher kann die direkte Arbeit mit Binärdaten oft die schnellste Möglichkeit sein. Für komplexere Operationen fassen wir »Bits« (die jeweils einen einzelnen binären Status repräsentieren) in Achtergruppen zusammen, die als *Oktett* oder meist als *Byte* bezeichnet werden.[3] So können wir größere Zahlen als Kombination aus 0 und 1 darstellen.

Durch das Erstellen von Sets aus 8 Bits können wir jede Zahl zwischen 0 und 255 repräsentieren. Das Bit ganz rechts steht für die 1 und wir verdoppeln den Wert mit jedem Bit, das wir uns nach links bewegen. Um herauszufinden, welche Zahl ein Byte darstellt, addieren wir einfach die Zahlen in den Spaltenköpfen (Übung 4-20).

Übung 4-20: 0 bis 255 in einem Byte darstellen

```
128 64 32 16 8 4 2 1
--- -- -- -- - - - -
0   0  0  0  0 0 0 0 = 0

128 64 32 16 8 4 2 1
--- -- -- -- - - - -
1   1  1  1  1 1 1 1 = 255

128 64 32 16 8 4 2 1
--- -- -- -- - - - -
1   0  0  1  0 1 0 1 = 149
```

Sie werden auch die hexadezimale Schreibweise (»Hex«) häufig sehen. Da sich Bytes auf einfache Weise darstellen lassen müssen, und ein String aus acht Nullen und Einsen nicht sehr handlich ist, hat sich die Hex-Darstellung weit verbreitet. Eine Binärnotation wird zur Basis 2 beschrieben, weil es zwei mögliche Status pro Ziffer gibt (0 oder 1). Hex nutzt

3 Es gibt keine »Standard«-Größe für ein Byte, aber de facto nutzt heutzutage jeder 8 Bit. Daher sind Oktetts und Bytes äquivalent und wir werden den allgemein üblichen Begriff *Byte* verwenden, wenn wir uns auf ein Oktett beziehen.

die Basis 16, und jede Ziffer kann dabei einen Wert von 0 bis F haben, wobei die Buchstaben A bis F (oder die entsprechenden Kleinbuchstaben) für die (Dezimal-)Werte 10 bis 15 stehen. Das Praktische an Hex ist die Tatsache, dass wir ein ganzes Byte mit zwei Ziffern darstellen können. Die rechte Ziffer steht für die 1er, die linke für die 16er. Wenn wir die Dezimalzahl 149 darstellen möchten, ist dies `(16 x 9) + (1 x 5)`, oder der Hex-Wert 95.

Übung 4-21: 0 bis 255 in Hex-Darstellung

```
Hex nach Dezimal:

0 1 2 3 4 5 6 7 8 9 A  B  C  D  E  F
- - - - - - - - - - -- -- -- -- -- --
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


In Hex zählen:

16 1
-- -
0  0 = 0

16 1
-- -
F  F = 255

16 1
-- -
9  5 = 149
```

In JavaScript, können Sie eine Zahl aus einem Hex-Wert erstellen, indem Sie vor den Wert »0x« schreiben. So entspricht zum Beispiel `0x95` der Dezimalzahl 149. In Node werden Sie immer wieder `Buffer` als Hex-Werte in der Ausgabe von `console.log()` oder Node REPL finden. Übung 4-22 zeigt, wie Sie drei Oktett-Werte (zum Beispiel einen RGB-Farbwert) als einen `Buffer` anlegen können.

Übung 4-22: Einen 3-Byte-Buffer aus einem Array mit Oktetts erzeugen

```
> new Buffer([255,0,149]);
<Buffer ff 00 95>
>
```

In welcher Beziehung stehen nun Binärdaten zu anderen Arten von Daten? Nun, wir haben gesehen, wie man Zahlen durch Binärdaten darstellen kann. In Netzwerkprotokollen ist es üblich, eine bestimmte Anzahl von Bytes anzugeben, um Informationen zu übermitteln. Dabei stehen dann Bits an bestimmten Positionen für die Information zu spezifischen Aspekten. In einem DNS-Request werden zum Beispiel die ersten beiden Bytes als Zahl für eine Transaktions-ID verwendet, während im nächsten Byte die einzelnen Bits genutzt werden, um anzuzeigen, dass bestimmte DNS-Features in diesem Request verwendet werden.

Das andere sehr wichtige Einsatzgebiet ist die Repräsentation von Strings. Die beiden gebräuchlichsten Kodierungen für Strings sind ASCII und UTF (meist UTF-8). Diese Kodierungen definieren, wie die Bits in Zeichen umgewandelt werden sollten. Wir werden nicht allzu sehr in die Details gehen, aber im Prinzip funktioniert die Kodierung mit einer Lookup-Tabelle, in der das Zeichen als eine bestimmte Zahl in Bytes abgebildet wird. Umgekehrt muss der Computer nur die Zahl in ein Zeichen umwandeln, indem er in eine Konvertierungs-Tabelle schaut.

ASCII-Zeichen (von denen einige unsichtbare »Steuerzeichen« wie zum Beispiel der Zeilenumbruch sind) sind immer genau sieben Bits lang, so dass Sie durch Werte von 0 bis 127 dargestellt werden können. Das achte Bit wird häufig genutzt, um den Zeichensatz zu erweitern, so dass er unterschiedlichste Gruppen internationaler Zeichen repräsentieren kann (wie zum Beispiel ȳ oder □).

UTF ist ein wenig komplexer. Sein Zeichensatz besitzt viel mehr Zeichen, unter anderem viele sprachspezifische Zeichen. Jedes Zeichen wird in UTF-8 durch mindestens ein Byte dargestellt, manchmal aber auch durch bis zu vier Bytes. Im Prinzip sind die ersten 128 Werte das gute alte ASCII, während die anderen Zeichen weiter nach hinten rutschen und durch größere Werte dargestellt werden. Wird ein seltener gebrauchtes Zeichen referenziert, dann findet sich im ersten Byte eine Zahl, die den Computer anweist, das nächste Byte anzuschauen, um zusammen die wirkliche Adresse des Zeichens auf der zweiten Seite der Tabelle herauszufinden. Befindet sich das Zeichen nicht auf der zweiten Seite, wird der Computer durch das zweite Byte auf die dritte Seite gelenkt und so weiter. Das heißt, dass die Länge eines Strings in UTF-8 in Zeichen nicht unbedingt auch der Länge in Bytes entspricht, wie dies bei ASCII der Fall ist.

Binärdaten und Strings

Denken Sie daran: Sobald Sie etwas in einen `Buffer` kopieren, wird dies dort in Binärdarstellung abgelegt. Sie können diese Binärdarstellung aus dem Buffer später auch wieder zurückkonvertieren – so auch in Strings. Ein `Buffer` ist nur durch seine Größe definiert, nicht durch eine Kodierung oder eine andere Kennzeichnung seiner Bedeutung.

Davon ausgegangen, dass ein `Buffer` nicht transparent ist, kann man sich fragen, wie groß er sein muss, um einen bestimmten Eingabestring abzuspeichern. Wie wir schon gesagt haben, kann ein UTF-Zeichen bis zu vier Byte einnehmen. Um sicherzugehen, sollten Sie also einen `Buffer` so definieren, dass er das Vierfache der längsten Eingabe in UTF-Zeichen aufnehmen kann. Es gibt teilweise die Möglichkeit, diesen Wert zu verringern – wenn Sie nur mit europäischen Sprachen arbeiten, reicht es, zwei Bytes pro Zeichen zu reservieren (wobei das Euro-Zeichen in UTF-8 drei Bytes benötigt).

Buffer verwenden

`Buffer` können mit drei möglichen Parametern erzeugt werden: der Länge des `Buffer` in Bytes, ein Array mit Bytes, die in den `Buffer` kopiert werden sollen, oder einen String, der

in den Buffer kopiert wird. Die erste und die letzte Methode sind die am häufigsten verwendeten. Es kommt nur selten vor, dass Sie ein JavaScript-Array mit Bytes haben.[4]

Das Erstellen eines Buffer mit einer bestimmten Größe ist ein recht häufiges und einfaches Szenario. Beim Erstellen geben Sie einfach die Anzahl der Bytes als Argument von Buffer mit (Übung 4-23).

Übung 4-23: Einen Buffer über die Byte-Länge erstellen

```
> new Buffer(10);
<Buffer e1 43 17 05 01 00 00 00 41 90>
>
```

Wie Sie in diesem Beispiel sehen, erhalten wir beim Erstellen eines Buffer die passende Anzahl an Bytes. Aber da dem Buffer einfach ein Bereich des Speichers zugewiesen wird, sind diese Bytes *nicht initialisiert* und in dem Speicherbereich finden sich die Reste dessen, was dort vorher abgelegt war. Das ist anders als bei allen nativen JavaScript-Typen, die ihren Speicher initialisieren, so dass beim Erstellen einer neuen einfachen Variable oder eines Objekts nicht das zugewiesen wird, was sich vorher dort befand. Stellen Sie sich einmal ein Cafe vor. Es ist viel los und Sie wollen sich gerne setzen. Am schnellsten geht das, indem Sie sich einfach an einen Tisch setzen, sobald die Vorgänger aufgestanden sind. Das ist zwar schnell, aber das schmutzige Geschirr und die Essensreste sind auch noch da. Vielleicht wäre es besser, darauf zu warten, dass die Bedienung abräumt und einmal den Tisch abwischt, bevor Sie Platz nehmen. Das ist in etwa der Unterschied zwischen Buffer und den nativen Typen. Buffer tun sehr wenig dafür, Ihnen das Leben zu erleichtern, dafür erhalten Sie aber direkten und schnellen Zugriff auf den Speicher. Wenn Sie eine nette, saubere Anzahl von Nullen im Speicher wünschen, dann müssen Sie das selbst erledigen (oder eine Hilfsbibliothek finden).

Buffer werden über eine Bytelänge vor allem dann erzeugt, wenn es um Netzwerk-Transportprotokolle mit sehr genau definierten Strukturen geht. Wenn Sie exakt wissen, wie groß die Daten sein werden (oder wie groß sie werden können) und Sie einen Buffer reservieren und aus Performancegründen auch wiederverwenden wollen, ist das die beste Vorgehensweise.

Am allerhäufigsten wird ein Buffer mit einem String aus ASCII- oder UTF-8-Zeichen erzeugt. Ein Buffer kann zwar beliebige Daten enthalten, aber er ist besonders nützlich für I/O mit Zeichendaten, da durch die bekannten Limitierungen die Operationen mit dem Buffer viel schneller werden können als bei normalen Strings. Wenn Sie also hochskalierbare Anwendungen erstellen, lohnt es sich häufig, Buffer zu nutzen, um Strings zu speichern. Das gilt insbesondere dann, wenn Sie die Strings nur durch die Gegend schieben, ohne wirklich mit ihnen zu arbeiten. Auch wenn also Strings als Primitive in JavaScript existieren, werden sie in Node häufig in Buffern abgelegt.

Wenn wir einen Buffer aus einem String erstellen, wie in Übung 4-24 gezeigt, ist die Standardkodierung UTF-8, sofern Sie keine andere angeben. Das heißt nicht, dass Buffer

4 Unter anderem wird der Speicher dadurch sehr ineffizient genutzt. Wenn Sie jedes Byte als Zahl abspeichern, verwenden Sie 64 Bit für 8 Bit.

den Speicherbereich so groß wählt, dass überall beliebige Unicode-Zeichen (mit jeweils vier Byte pro Zeichen) möglich sind, sondern nur, dass keine Zeichen abgeschnitten werden. In diesem Beispiel sehen wir, dass der `Buffer` bei einem englischsprachigen String nur aus Kleinbuchstaben bei jeder Kodierung die gleiche Bytestruktur nutzt, da alle zum gleichen Ergebnis führen. Haben wir aber ein »é«, dann wird dies mit UTF-8 (implizit oder explizit definiert) mit zwei Byte kodiert. Wenn wir ASCII angeben, wird das Zeichen auf ein einzelnes Byte zurechtgestutzt.

Übung 4-24: Buffer durch Strings erzeugen

```
> new Buffer('foobarbaz');
<Buffer 66 6f 6f 62 61 72 62 61 7a>
> new Buffer('foobarbaz', 'ascii');
<Buffer 66 6f 6f 62 61 72 62 61 7a>
> new Buffer('foobarbaz', 'utf8');
<Buffer 66 6f 6f 62 61 72 62 61 7a>
> new Buffer('é');
<Buffer c3 a9>
> new Buffer('é', 'utf8');
<Buffer c3 a9>
> new Buffer('é', 'ascii');
<Buffer e9>
>
```

Mit Strings arbeiten

Node stellt eine ganze Menge Funktionalität bereit, um die Arbeit mit Strings und `Buffern` zu vereinfachen. So brauchen Sie zum Beispiel die Länge eines Strings nicht zu berechnen, bevor Sie einenentsprechenden `Buffer` erzeugen – übergeben Sie einfach beim Erstellen des `Buffer` den String als Argument. Stattdessen können Sie auch die Methode `Buffer.byteLength()` nutzen. Man übergibt dieser Methode einen String und eine Kodierung und erhält die Länge des Strings in Byte (und nicht in Zeichen, wie dies bei `String.length` der Fall ist).

Sie können auch einen String in einen bestehenden `Buffer` schreiben. Die Methode `Buffer.write()` schreibt einen String an eine bestimmte Stelle in einem `Buffer`. Ist der `Buffer` groß genug, wird der String vollständig ab der angegebenen Position geschrieben, ansonsten werden Zeichen am Ende des Strings abgeschnitten. In beiden Fällen gibt `Buffer.write()` die Anzahl der geschriebenen Bytes zurück. Bei UTF-8-Strings kann es passieren, dass ein Zeichen, das mehr als ein Byte belegt, mittendrin zerschnitten würde. In diesem Fall wird es gar nicht geschrieben. In Übung 4-25 ist der `Buffer` so klein, dass selbst ein einzelnes Nicht-ASCII-Zeichen nicht hineinpasst, daher wird es gar nicht geschrieben.

Übung 4-25: Buffer.write( ) und zu lange Zeichen

```
> var b = new Buffer(1);
> b
<Buffer 00>
```

```
> b.write('a');
1
> b
<Buffer 61>
> b.write('é');
0
> b
<Buffer 61>
>
```

In einem `Buffer`, der nur ein Byte groß ist, kann man das Zeichen »a« einfügen und man erhält den Wert `1` zurück, weil ein Byte geschrieben wurde. Versuchen wir aber, das Zeichen »é« zu schreiben, schlägt dies fehl (da es zwei Byte groß ist) und die Methode gibt `0` zurück, weil nichts geschrieben wurde.

`Buffer.write()` ist allerdings noch ein bisschen komplexer. Beim Schreiben von UTF-8-Strings versucht `Buffer.write()` den String mit einem NUL-Zeichen zu beenden, sofern noch ausreichend Platz ist.[5] Das sollte man vor allem dann im Hinterkopf haben, wenn man mitten in einen größeren `Buffer` schreibt.

In Übung 4-26 wird ein `Buffer` mit fünf Byte erstellt, danach schreiben wir das Zeichen `f` in den gesamten `Buffer` (was wir natürlich auch direkt beim Anlegen des Buffers hätten tun können). `f` hat den Zeichencode 0x66 (oder dezimal 102). So sieht man sehr gut, was passiert, wenn wir die Zeichen »ab« an der Indexposition 1 in den `Buffer` schreiben. Das nullte Zeichen ist weiterhin ein `f`. An den Positionen 1 und 2 werden die Zeichen selbst geschrieben – 61, gefolgt von 62. Dann fügt `Buffer.write()` einen Terminator ein – in diesem Fall das Null-Zeichen 0x00.

Übung 4-26: Einen String mit einem Terminator in einen Buffer schreiben

```
> var b = new Buffer(5);
> b.write('fffff');
5
> b
<Buffer 66 66 66 66 66>
> b.write('ab', 1);
2
> b
<Buffer 66 61 62 00 66>
>
```

console.log

Der einfache Befehl `console.log` ist aus dem Firebug des Firefox entliehen und ermöglicht Ihnen, Texte an stdout zu schicken, ohne auf irgendwelche Module zurückgreifen zu müssen (Übung 4-27). Netterweise gibt er Objekte ansprechend formatiert aus, so dass man deren innere Struktur gut erkennen kann.

5 Das ist eine binäre 0.

Übung 4-27: Ausgabe mit console.log

```
> foo = {};
{}
> foo.bar = function() {1+1};
[Function]
> console.log(foo);
{ bar: [Function] }
>
```