Ein Ritt auf dem Nashorn

Wie sich Client und Server JavaScript-Code teilen können

Lukasz Plotnicki, Manuel Pütz

Mit Java 8 hat die JavaScript-Engine Rhino den Nachfolger Nashorn erhalten. Der Artikel beschreibt einen Anwendungsfall, der im ersten Moment ungewöhnlich erscheint: Das Ausführen in JavaScript implementierter Datenmigrationen in einer Java-Server-Anwendung.

Einleitung und Motivation

Zunächst beschreiben wir kurz das Szenario, in dem wir diesen Anwendungsfall gefunden haben. Unsere Anwendung wird zum Erstellen von Dokumenten verwendet und muss auch ohne zuverlässige Internet-Anbindung nutzbar sein. Selbst wenn mehrere Tage oder Wochen lang keine Internet-Verbindung besteht, soll das die Funktionalität nicht beeinträchtigen. Der Client ist eine Webapplikation (Single-Page App), die auf AngularJS basiert und Daten in einer IndexedDB speichert.

Der Server ist mit Java 8 und Spring MVC umgesetzt und speichert die Daten in einer MongoDB. Dabei wird die Struktur eines Dokuments im Client definiert und das Backend wird lediglich zur Synchronisierung zwischen verschiedenen Clients beziehungsweise als Backup verwendet. Damit ist die Client-Anwendung weitgehend unabhängig und dank Application Cache, IndexedDB und anderen HTML5-Programmierschnittstellen beziehungsweise Web-Storage-Technologien auch offline vollständig funktionsfähig. Sobald der Client eine Verbindung zum Backend aufgebaut hat, synchronisiert er die Dokumente.

In diesem Szenario haben wir zwei NoSQL-Datenbanken und folglich kein klassisches Datenbankschema. Es gibt aber ein implizites Schema (siehe auch [MART]), das im JavaScript-Code im Client lebt. Das Backend dagegen weiß wenig über die Struktur der Daten. Es muss lediglich einige Metadaten zur Erkennung von Konflikten beim Synchronisieren verwalten. Der große Vorteil ist, dass bei Änderungen an der Datenstruktur nur der Client, nicht aber das Backend angepasst werden muss.

Es stellt sich allerdings die Frage, wie wir mit Datenmigrationen umgehen. Bei dieser Architektur müssen jederzeit Client-Datenbanken in unterschiedlichen Versionen mit der Datenbank im zentralen Backend zusammenarbeiten können. Diese Versionsheterogenität ergibt sich aus den unterschiedlich langen Offline-Perioden einzelner Clients und der Anzahl an Releases, die in der Zwischenzeit veröffentlicht worden sind. Obwohl das Backend nur über eine grobe Kenntnis der Datenstruktur verfügt, wäre die Erstellung akkurater Berichte und jegliche Fehleranalyse erheblich schwerer, wenn die Daten nicht auch serverseitig migriert würden.

Hier kommt Nashorn ins Spiel, das es uns ermöglicht, die in JavaScript geschriebenen Client-Migrationen auch im Backend zu verwenden.

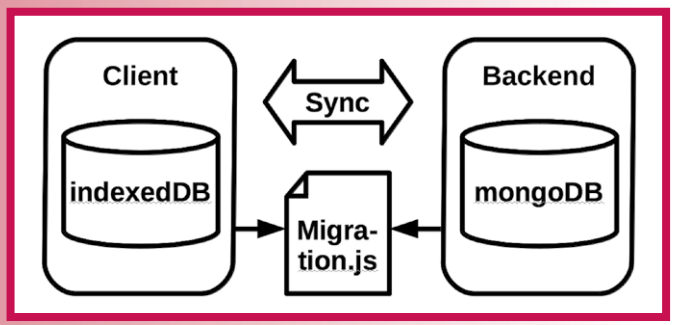

Abb. 1: Überblick

Nashorn

Nashorn ist eine mit JDK8 ausgelieferte JavaScript-Engine, die

- eine erheblich bessere Performance im Vergleich zum Vorgänger Rhino aufweist,
- mit dem ECMAScript 5.1-Standard kompatibel ist und
- eine einfache Ausführung von JavaScript-Code auf der JVM ermöglicht.

Die dynamische Natur von JavaScript wird somit mit dem reichen Java-Ökosystem verbunden und bietet eine sehr gute Alternative bei der Lösung vieler Problemstellungen: Skript-Erstellung, serverseitige Benutzung der JavaScript-Tools (z. B. Templating-Engines) oder Wiederverwendung des Client-Codes, die Einsatzmöglichkeiten sind breit. Der Einstieg ist einfach, da Nashorn mittels `jjs` [ORAC] eine interaktive REPL anbietet und gleichzeitig eine leichte Erstellung ausführbarer Skripte ermöglicht.

```
#!/usr/bin/jjs
var time = Packages.java.time;
var timeFormat = time.format.DateTimeFormatter.ofPattern('hh:mm');
var dateFormat = time.format.DateTimeFormatter.ofPattern('dd.MM.yyyy');
var currentTime = time.LocalTime.now().format(timeFormat);
var currentDate = time.LocalDate.now().format(dateFormat);
print('Hello Lukasz');
print("Aktuelle Zeit: ${currentTime} & Datum: ${currentDate}");
```

Listing 1: Shebang-Skript mit JavaScript und Java

Listing 1 zeigt den Aufruf von Java-Klassen aus JavaScript. Sollte die JDK-Programmierschnittstelle nicht ausreichen, kann der Classpath um weitere Java-Bibliotheken ergänzt werden: `#!/usr/bin/jjs -cp ./lib/commons-lang3-3.4.jar`.

Alles, was zum Ausführen des JavaScript-Codes aus Java nötig ist, kann in dem `javax.script`-Package gefunden werden. Listing 2 zeigt das Erzeugen einer Nashorn-Engine und wie mit dieser beliebige JavaScript-Dateien geladen und darin definierte Funktionen aufgerufen werden. Hierbei ist zu beachten, dass dabei keinerlei Überprüfung stattfinden kann, ob die genannte Funktion existiert oder nicht. Dementsprechend kann bei einem Fehler zur Laufzeit die `java.lang.NoSuchMethodException` ausgelöst werden.

```
import javax.script.Invocable;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;
// (..)
ScriptEngine nashorn =
  new ScriptEngineManager().getEngineByName("nashorn");
nashorn.eval(new FileReader("calculator.js"));

Double added =
  (Double) ((Invocable) nashorn).invokeFunction("add", 1, 2);
Double multiplied =
  (Double) ((Invocable) nashorn).invokeFunction("multiply", added, 2);
```

Listing 2: JavaScript-Ausführung mittels Nashorn-Engine

```
function add(one, two) {
  return one + two;
}
function multiply(one, two) {
  return one * two;
}
```

Listing 3: calculator.js

Nashorn bietet auch eine ganze Reihe an JavaScript-Syntax-Erweiterungen und speziellen Funktionen [NASH] an. Ein Beispiel ist `load()`, mit der man zur Laufzeit weitere JavaScript-Dateien laden und evaluieren kann. Diese Erweiterungen werden jedoch oft nur von Nashorn unterstützt und deren Benutzung führt dazu, dass der JavaScript-Code von anderen Engines nicht korrekt ausgeführt wird.

Vorstellung unserer Lösung

Aus dem zuvor beschriebenen Szenario ergibt sich, dass für den Migrationscode der größte gemeinsame Nenner zwischen den beiden JavaScript-Engines V8 (Client wird ausschließlich in Chrome verwendet) und Nashorn gefunden werden muss. Demnach kann beispielsweise die komplette Nashorn-Java-Programmierschnittstelle nicht verwendet werden. Auch für andere Nashorn-spezifische Funktionen wie `load()` zum Laden und Ausführen von Script-Dateien müssen Alternativen gefunden werden.

Wir versuchen, Abhängigkeiten zu JavaScript-Bibliotheken weitestgehend zu vermeiden. Auf *lodash* wollten wir dann aber doch nicht verzichten. Denn gerade beim Manipulieren großer verschachtelter Collections im Rahmen einer Migration erweisen sich *lodash*s Collection-Funktionen als sehr hilfreich. Die Alternative zu `load()` im JavaScript-Code wird im Java-Code wie in Listing 4 umgesetzt.

```
ScriptEngine engine =
  new ScriptEngineManager().getEngineByName("nashorn");
engine.eval(new InputStreamReader(
  this.getClass().getResourceAsStream("/lodash.min.js")));
engine.eval(new InputStreamReader(
  this.getClass().getResourceAsStream("/migrations.js")))
```

Listing 4: Initialisieren der Script-Engine

Die hier geladenen Dateien werden automatisch beim Erstellen einer neuer Version der Anwendung aus dem Client-Repository mittels eines Gradle-Tasks kopiert, sodass der ausgeführte Code auf beiden Seiten der gleiche ist. Listing 5 zeigt die Funktion an der Schnittstelle zum JavaScript-Code für das Migrieren eines einzelnen Datenobjekts aus Java.

```
function migrate(collectionName, data, targetVersion) {
  return JSON.stringify(Migrations.execute(collectionName,
    JSON.parse(data), targetVersion));
}
```

Listing 5: Schnittstelle zwischen Java und Migrationscode

Für das Backend sind die in der MongoDB gespeicherten Daten nur Datencontainer, die als Maps<> repräsentiert werden. Daher ist es naheliegend, JSON als Austauschformat zwischen Java und JavaScript zu wählen. Dies bedeutet, dass jedes aus der MongoDB ausgelesene Objekt entsprechend in eine JSON-Repräsentation umgewandelt werden muss, um nach einer erfolgreichen Migration wieder aus JSON deserialisiert zu werden. Für das JSON-Parsing in Java kommt Googles GSON-Bibliothek zum Einsatz.

Listing 6 zeigt die Schritte zum Migrieren aller Daten einer einzelnen Datenbank-Collection.

```
// 1: Erstelle einen Datenbank-Cursor,
// um die zu migrierenden Daten zu finden
DBCursor objects = collection.find(query);
while (objects.hasNext()) {
  DBObject oldData = objects.next();

  // 2: Wandele jedes einzelne Datenobjekt in seine JSON-Darstellung um
  String oldDataJson = gson.toJson(oldData.toMap());

  // 3: Verwende den JavaScript-Code zum Migrieren des Datenobjekts
  String migratedJson = (String) scriptEngine.invokeFunction(
    "migrate", collection.getName(), oldDataJson, targetSchemaVersion);

  // 4: Parse das JSON-Ergebnis für das Update der mongoDB
  Type typeOfMap = new TypeToken<Map<String, Object>>() {}.getType();
  Map<String, Object> migratedData =
    gson.fromJson(migratedJson, typeOfMap);
  migratedData.remove("_id");
  collection.update(oldData, new BasicDBObject(migratedData));
}
```

Listing 6: Migration einer Collection aus Java

Fehlerbehandlung und -Logging

Es stellt sich die Frage, wie bei solch kritischem Code mit Fehlern umgegangen werden soll. Da die eigentliche Migration in JavaScript immer nur mit genau einem Dokument aufgerufen wird, wird im Falle eines Fehlers dieses Dokument unverändert zurückgegeben und die Migration läuft weiter. Dazu werden die einzelnen Migrationen mit einer Kopie der Daten durchgeführt, damit, falls nötig, zum ursprünglichen Zustand zurückgerollt werden kann, siehe Listing 7 (Listing 8 zeigt die Funktion `migrateOneVersion`, die den Migrationscode aufruft). Da jedes Dokument auch seine Datenschemaversion enthält, sind bei der anschließenden Fehleranalyse alle nicht migrierten Dokumente leicht zu finden.

```
migrations.execute =
    function (collectionName, data, targetVersion, logService) {
  var migratedData = _.cloneDeep(data);
  var currentVersion = parseInt(data.schemaVersion) || 1;
```


```
function migrateOneVersion(migrationVersion) {
  var migrationVersionString = migrationVersion.toString();
  var migration = migrations[collectionName][migrationVersionString];
  if (migration) {
    try {
      migration(migratedData);
    } catch (e) {
      var cause = 'Could not migrate data: ' + JSON.stringify(data) +
         ', to version:' + targetVersion;
      logService.logError(e, cause);
      throw {name: 'MIGRATION_FAILED', message: cause};
    }
  } else {
    logService.logInfo('No migration present for version: ' +
      migrationVersionString);
  }
}
try {
  for (var migrationVersion = currentVersion + 1;
      migrationVersion <= targetVersion; migrationVersion++) {
    migrateOneVersion(migrationVersion);
  }
} catch (e) {
  logService.logInfo('Rolling back to version ' + currentVersion);
  migratedData = data;
}
return migratedData;
};
```

Listing 7: Fehlerbehandlung

Solides Logging ist hier unabdingbar. Aber wie vermittelt man vom JavaScript-Logger zum Java-Logger? Läuft die Migration im Client, dann ist ein logService als Angular-Service definiert, der alle Fehlerinformationen inklusive einer Client-ID an das Backend sendet, wo sie in Log-Dateien geschrieben werden.

Läuft die Migration im Backend, übernimmt ein anderer logService, der deutlich vereinfacht werden kann. Das Versenden eines Fehlerevents ist nicht nötig und Ausgaben auf System.out werden schon in Log-Dateien umgeleitet. Hier bietet es sich an, eine der zusätzlichen Nashorn-Funktionen zu verwenden: `print()` benutzt intern `java.lang.System.out.println()`. Listing 8 zeigt den vereinfachten logService für das Backend und der Migrationscode verwendet die gemeinsame Schnittstelle.

```
logService = logService || {
  logError: print,
  logInfo: print
};
```

Listing 8: Nashorn-Implementierung des JavaScript logService

Diskussion

Mit dem hier beschriebenen Ansatz ist es möglich, den Code für Datenmigrationen sowohl im Client als auch im Backend zu nutzen. So fällt der Entwicklungsaufwand nur einmal an. Der zusätzliche Aufwand, um JavaScript-Code in Java zu nutzen, ist gering. Den Umweg von MongoDB-Objekten über Map<> und JSON zu JavaScript-Objekten nehmen wir gerne in Kauf, denn der Umgang mit den erstellten Dokumenten ist dank der dynamischen Natur von JavaScript in dieser Sprache deutlich einfacher.

Wie man auch erkennt, wird unsere Migrationslogik einmalig ausgeführt (*Big Bang Migration*), indem alle Dokumente geladen und migriert werden, die der gewünschten Version nicht entsprechen. Reporting und Suchfunktionalität sind deutlich einfacher zu entwickeln und zu warten, wenn die gesamte Datenbasis im Backend die gleiche Datenstruktur aufweist. Dadurch dauert es bei einem Deployment gegebenenfalls deutlich länger, bis die Migration abgeschlossen ist und das gesamte System bereitsteht. Wer diesen Nachteil vermeiden will, könnte den gleichen Ansatz aber auch für *Rolling Migrations* verwenden. Sobald ein Objekt gelesen wird, findet eine Überprüfung bezüglich der Version statt und falls nötig wird die Migrationslogik via Nashorn angestoßen.

Wir haben die Möglichkeit des Code Sharing zwischen Client und Server in diesem Szenario sehr zu schätzen gelernt und überlegen, es auch in anderen Bereichen einzusetzen. Eine wichtige Funktion unserer Anwendung ist die Generierung von PDF-Dokumenten. Damit das auch ohne Verbindung zum Backend jederzeit möglich ist, haben wir die Logik mit JavaScript im Client implementiert. Nun wäre es denkbar, dass für Reports auch im Backend PDF-Dokumente erzeugt werden sollen. Mit Nashorn könnte auch dieser JavaScript-Code im Backend verwendet werden, sodass für alle zu erzeugenden PDFs eine gemeinsame Code-Basis verwendet werden kann.

Links

[MART] Schemaless – Implict schema,
`//http://martinfowler.com/articles/schemaless/#implicit-schema`
[NASH] Nashorn extensions,
`https://wiki.openjdk.java.net/display/Nashorn/Nashorn+extensions`
[ORAC] jjs,
`http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jjs.html`

Lukasz Plotnicki (Dipl.-Inform. Med.) ist Consultant und Softwareentwickler bei ThoughtWorks in Hamburg. Er ist begeistert von neuen Technologien - insbesondere HTML5 und funktionalen Programmiersprachen. Er arbeitet seit 2009 an Webapplikationen und sieht mit Freude, wie Web-Technologien reifer und professioneller werden.
E-Mail: lplotnic@thoughtworks.com

Manuel Pütz, B.Sc. Informatik, ist Softwareentwickler bei ThoughtWorks in Hamburg und interessiert sich besonders für offline-first Webapplikationen.
E-Mail: mpuetz@thoughtworks.com