ECHTZEITKOMMUNIKATION MIT (ROOM)SERVICE

Zimmerservice, bitte!

Das von Google Docs und Co. bekannte gemeinsame Arbeiten an einem Dokument nachzubauen ist aufwendig. (room)service will das ändern.

Mit dem Aufkommen des Web 2.0 haben sich die Möglichkeiten der Online-Zusammenarbeit stark erweitert. Stabiler und schneller Internetzugang, Technologien mit mehr Möglichkeiten und der Anspruch der Nutzer haben hier zu vielen neuen Entwicklungen geführt. Dieser Trend hält nicht nur bis heute an, sondern scheint sich zu beschleunigen beziehungsweise zu verstärken.

Wer den Begriff der Echtzeit-Kollaboration in den Mund nimmt, wird schnell an Anbietern wie Google und Microsoft gemessen, die mit Tools wie Google Docs und Office 365 Pflöcke im Bereich der Echtzeit-Kollaboration eingeschlagen haben. Wenn Daten und im Speziellen Dokumente online verfügbar sind, erwarten viele, dass diese auch gemeinsam bearbeitet werden können. Features wie mehrfache Cursor von verschiedenen Personen und konfliktfreies Mergen scheinen für viele einfach dazuzugehören.

Aus Sicht der Softwareentwicklung und der notwendigen Technologien sind diese Features allerdings nicht ohne. Es gibt zahlreiche Möglichkeiten, wie ein solches Vorhaben scheitern kann oder deutlich mehr Ressourcen zur Umsetzung braucht als vorher gedacht. Mit (room)service ist vor Kurzem eine Bibliothek auf der Bildfläche erschienen, die genau da ansetzt und die Anwendungen in Sachen Echtzeit-Kollaboration in 15 Minuten fit machen möchte. Wir schauen uns die Bibliothek an, die allerdings gerade erst in einer Alpha-Version verfügbar ist.

Was ist (room)service?

Die Bibliothek (room)service wirbt damit, von Figma [1], einem kollaborativen Design-Tool, oder von Google Docs bekannte Funktionen in eigene Anwendungen zu integrieren. Das Versprechen geht so weit, dass die eigene App innerhalb von 15 Minuten der Echtzeit-Kollaboration mächtig sein soll. Das ist ein ordentliches Versprechen und wird sich aktuell noch nicht auf Herz und Nieren überprüfen lassen.

Der Grund ist, dass (room)service in einer Alpha-Version verfügbar ist. Denn erschienen ist die Bibliothek noch nicht. Der Zugang muss über die Website [2] angefordert werden. Im Zuge dieses Artikels war der Zugang nach zwei bis drei Wochen freigeschaltet. Daher wird aus aktuellem Anlass in dieser Kolumne etwas außer der Reihe einmal eine JavaScript-Bibliothek vorgestellt.

Aufgrund der Alpha-Version von (room)service werden in diesem Artikel einige Grundlagen und erste Funktionen vorgestellt. In zukünftigen Artikeln begleiten wir die Entwicklung der Bibliothek aber weiter und halten Sie so auf dem Laufenden. Zudem ist der Autor mit dem Entwicklerteam in Kontakt – für zusätzliche spannende Einblicke hinter die Kulissen.

Installation

Die Installation findet über NPM statt, den Paketmanager für die Webwelt. Momentan wird bei (room)service zwischen verschiedenen Varianten unterschieden. Die einfachste ist sicherlich die Komponente browser. Der Name ist Programm, stecken hier doch die Funktionen drin, die für das Frontend einer Anwendung notwendig sind. Wer eine Website baut und die Bibliotheken recht einfach für den Austausch von Daten einsetzen will, wird mit diesem Paket glücklich. Insgesamt gibt es die folgenden Pakete:

- @roomservice/browser
- @roomservice/node
- @roomservice/react

Das Paket node ist im Backend wichtig. Die aktuellen Beispiele gehen darauf ein, wie mit Node und Express.js ein Endpunkt für die Authentifizierung bereitgestellt werden kann. Dazu aber später mehr. Zu guter Letzt bietet das Paket react die React Hooks an, was sinnvoll ist, wenn das Frontend-Framework React zum Einsatz kommt.

Zur Installation an sich bleibt nicht viel zu sagen. Einfach das *npm*-Kommando ausführen, zum Beispiel wie folgt:

```
npm install --save @roomservice/browser
```

Alternativ kann der Paketmanager Yarn genutzt werden. Nach der Installation kann es direkt losgehen.

Der Quelltext der Bibliotheken steht auf GitHub zur Verfügung, das meiste davon unter der MIT-Lizenz.

Features

Die Kernfunktionalität wird vom *RoomService*-Objekt bereitgestellt. Über das können sogenannte Räume bereitgestellt oder betreten werden. Ein Raum beinhaltet Daten, die im Sprachgebrauch der (room)service-Bibliothek *Dokument* genannt werden. Dieses Dokument kann aktualisiert/angepasst werden und alle, die einem Raum beigetreten sind, bekommen diese Änderung mit und können selbst ebenfalls Änderungen durchführen. Der Server stellt dann sicher, dass die Daten konsistent sind.

Bei Räumen wird zwischen öffentlichen und privaten Räumen unterschieden. Der Begriff öffentlich trifft es hier sehr

gut, denn ein öffentlicher Raum ist nicht nur für eine bestimmte Benutzergruppe öffentlich, die vorher spezifiziert werden könnte, sondern tatsächlich für alle, die (room)service als Bibliothek in ihren Projekten nutzen. Noch genauer: tatsächlich von der ganzen Welt.

Private Räume hingegen sind zunächst auch erst einmal öffentlich. Der Unterschied ist aber, dass die Bibliothek (room) service einen Auth-Endpunkt benutzt, den Sie in Ihren eigenen Anwendungen implementieren müssen. Dieser Endpunkt entscheidet dann, ob ein Benutzer einen Raum tatsächlich betreten darf oder abgelehnt wird. Private Räume sind demnach immer noch auf dem Server verfügbar, Sie stellen nur durch die eigene Anwendung einen Türsteher ab, der im Zweifelsfall den Zutritt verwehrt.

Die React Hooks erlauben es, (room)service etwas komfortabler in den Komponenten einer React-App zu nutzen.

Beim Auslesen von Daten geht (room)service den Weg über den eigenen Browser. Der speichert eine Kopie der Daten, die unter Umständen aber nicht konsistent mit denen auf dem Server ist. Denn es ist möglich, dass Teilnehmende eines Raums offline sind. Dann werden die Änderungen an einem Dokument lokal zwischengespeichert. Sobald der Weg ins Internet wieder frei ist, können die Änderungen mit allen anderen im Raum zusammengeführt und an alle Teilnehmenden verteilt werden.

Der entfernte Room Service hat nicht zwingend aktuellere Daten als der eigene Browser, weil erst beim Aktualisieren der Daten eines Dokuments sichergestellt wird, dass alle Änderungen berücksichtigt werden. Das Mergen von Daten geschieht völlig automatisch und im Hintergrund. Über das Prinzip des „conflict-free replicated data type“ (CRDT [3]) wird sichergestellt, dass bei allen Änderungen auch die Intention der Änderungen berücksichtigt bleibt.

Das bedeutet, dass eine Liste, die von mehreren Leuten verändert wird, nachher die korrekten Elemente besitzt, auch bezogen auf die Reihenfolge. Denn wenn ein Benutzer ein Element aus einer Liste mit vier Elementen löscht, etwa Nummer zwei, und ein weiterer Benutzer den Inhalt von Nummer vier anpasst, dann ist, technisch gesehen, die Nummer vier nicht mehr das letzte Element. Die Bibliothek stellt aber sicher, dass sich die Änderung auf das korrekte Listenelement bezieht.

Der erste öffentliche Raum

Jetzt geht es ans Eingemachte: Wir erstellen den ersten (öffentlichen) Raum. Per Definition ist jeder erstellte Raum öffentlich, außer es wird eine Authentifizierung genutzt, die im nächsten Beispiel vorgestellt wird. Da sich dieses Beispiel komplett im Frontend abspielt, reicht die Installation des Pakets browser:

```
npm install --save @roomservice/browser
```

Der API-Endpunkt lautet wie folgt:

```
https://api.roomservice.dev/public
  /proj_01E0W5HJY7NSQW3Y93THED7D8A
```

Der API-Key kann über das minimalistische Backend erstellt werden. Ein RoomService-Objekt wird wie folgt erstellt:

```
const client = new RoomService({
  authUrl: "<URL>"
});
```

Der öffentliche Raum ist nach der nächsten Programmzeile erstellt:

```
const room = client.room("public");
```

Wichtig ist, dass der Raum *public* genannt werden muss. Andere Möglichkeiten gibt es zurzeit nicht. Wer einem Raum beitreten möchte, realisiert das über die folgenden zwei Zeilen Code:

```
const room = client.room("<Raum>");
const { doc } = await room.init();
```

Dabei wird ein nicht veränderliches Dokument zurückgeliefert, das aus einem JSON-Objekt besteht. Dieses sollte nicht direkt modifiziert werden. Es dient vielmehr dazu, den aktuellen Stand des Raums darzustellen, zum Beispiel, um das UI einer Anwendung korrekt zu initialisieren. Wenn erforderlich, kann das Dokument eines Raums über die *client.room()*-Funktion initialisiert werden:

```
const room = client.room("<Raum>", {
  title: "Standard-Titel"
});
```

Die ID für den Raum, in den obigen Beispielen *<Raum>*, kann frei vergeben werden. Auf diese Weise sind viele Anwendungsfälle realisierbar.

Änderungen an einem Dokument

Um die Daten eines Dokuments eines Raums anzupassen, sollten nicht die Eigenschaften des JSON-Objekts direkt genutzt werden. Das kann zu Problemen/Inkonsistenzen führen. Die Bibliothek stellt dafür eigene Funktionen bereit, wie zum Beispiel *publishDoc*:

```
const newDoc = room.publishDoc(prevDoc => {
  prevDoc.title = "Neuer Titel...";
});
```

Das zurückgelieferte neue Dokument, hier die Variable *newDoc*, ist wieder eine Read-only-Version des Dokuments, wie bei der *init*-Funktion, dieses Mal aber mit den gerade durchgeführten Änderungen. Wenn für eine Aktualisierung der Daten eines Dokuments andere Daten desselben Dokuments notwendig sind, dann sollten immer die Daten der *prevDoc*-Variablen genutzt werden und nicht eine alte Kopie des Dokuments, die womöglich in einer Variablen schlummert. Das kann zu Inkonsistenzen führen, da nur die *prevDoc*-Variable Änderungen von anderen Benutzern berücksichtigt hat. ▶

Räume erstellen und Änderungen durchführen sind damit bereits beschrieben. Natürlich ist es noch wichtig, Änderungen von anderen Benutzern eines Raums mitzubekommen. Das funktioniert über die *onUpdate*-Funktion:

```
room.onUpdate(newDoc => {
  // ...
});
```

Die Variable *newDoc* stellt das neue Dokument mit Änderungen anderer Benutzer zur Verfügung, wie üblich als JSON-Dokument.

Authentifizierung

Die bisherigen Beispiele beziehen sich alle auf den erstellten Raum, der wie schon beschrieben öffentlich ist. Um die Öffentlichkeit auszusperren, muss die eigene Anwendung eine Prüfung durchführen. Dazu ist auf der eigenen Seite ein Endpunkt notwendig. Welcher URL dafür genutzt wird, ist beliebig. Laut Dokumentation ist */auth/roomservice* gebräuchlich. Listing 1 zeigt einen Ausschnitt des notwendigen Codes.

Was das Listing nicht zeigt, sind die Import-Anweisungen, die Implementierung der *AllowedAccess*-Funktion und das Aufsetzen des Express.js-Servers. Deutlich wird aber, dass die Rückgabe des *Response*-Bodys geparst werden muss. Einen Beitrittswunsch eines Benutzers abzulehnen funktioniert über einen Aufruf von *client.reject()*, wohingegen *client.authorize()* für die Annahme zuständig ist. Dabei wird die Referenz des Raums weitergegeben, die sich hinter der Eigenschaft *room.reference* verbirgt.

Sowohl das Schlüsselwort Referenz als auch die mitgelieferte Zeichenkette sind bewusst vage und generisch gehalten. Auf diese Weise können unterschiedliche Anwendungsszenarien über die Räume abgebildet werden, wie schon zuvor angedeutet. Zum Beispiel können unterschiedliche Rollen oder Abteilungen eines Unternehmens über *unternehmen/**, *unternehmen/dev* und *unternehmen/sales* abgebildet werden. Der Benennung sind dabei keine Grenzen gesetzt.

React Hooks

Die React Hooks sind für alle interessant, die React als Frontend-Framework einsetzen. Daher gehen wir im weiteren Verlauf nur kurz auf dieses Feature ein. Wichtig sind die folgenden zwei Pakete:

Listing 2: Integration in eine React-Komponente

```
function MyComponent() {
  const [sharedState, setSharedState] =
    useSharedState(client, "<Raum>");
  function onChange() {
    setSharedState(prevState => {
      prevState.myOption = "<Daten>";
    });
  }
  // ...
}
// ...
```

```
npm install --save @roomservice/react
```

```
npm install --save @roomservice/browser
```

Das Erstellen eines Raums oder das Betreten eines vorhandenen funktioniert wie oben angedeutet. Anschließend ist der folgende Import notwendig:

```
import { useSharedState } from "@roomservice/react";
```

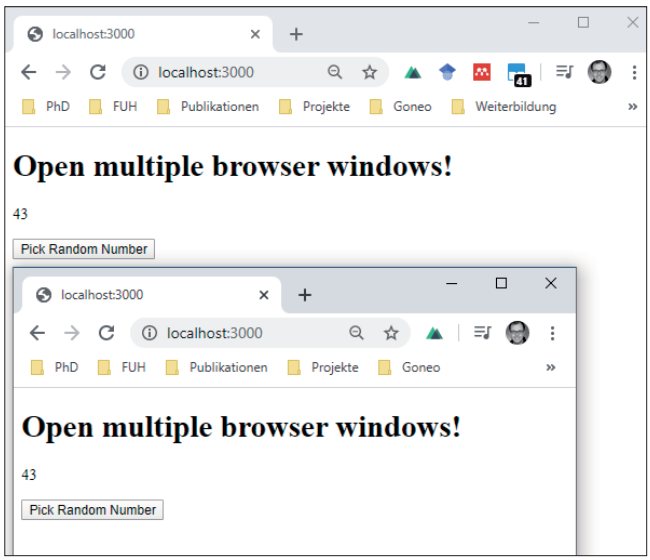
Das Listing 2 zeigt, wie dieser Shared State in einer Komponente genutzt wird. Gegenüber den bisherigen Beispielen haben sich nur die Funktionen geändert, das Prinzip bleibt identisch. Auch die bisher erläuterten Regeln gelten weiterhin. *setSharedState* akzeptiert nur eine Funktion als Argument, kein Objekt. Des Weiteren darf *setSharedState* nichts zurückgeben. Das Dokument/Objekt muss direkt über die vorhandenen Eigenschaften angepasst werden. Zu guter Letzt dürfen nur JSON-Primitive genutzt werden. Objekte führen zu einem unerwarteten Verhalten.

Daten mergen

Etwas weiter oben wurde kurz das Konzept CRDT angesprochen. Durch geeignete Datenstrukturen ist sichergestellt, dass verteilte und zeitlich unabhängige Änderungen zu einem konsistenten Dokument und damit zu konsistenten Daten führen.

Listing 1: Die Authentifizierung für Räume über einen eigenen Endpunkt

```
// ...
const client = new RoomService("YOUR_API_KEY");
app.post("/auth/roomservice", (req, res) => {
  const { room } = client.parse(req.body);
  const userId = "your-user-id";
  if (!isUserAllowedToAccessRoom(req.user, room)) {
    return client.reject();
  }
  return client.authorize(res, {
    guest: req.user.id, // must be a string
    room: room.reference
  });
});
// ...
```



Die Daten einer durch (room)service erweiterten Web-App werden korrekt verteilt (Bild 1)

Noch einmal sei angemerkt, dass es sich dabei um eine eventuelle Konsistenz handelt. Das bedeutet, dass die eigenen Daten nicht zwingend alle Änderungen beinhalten müssen. Erst wenn alle Teilnehmer sicher ihre Daten übertragen haben, zum Beispiel weil sie zwischendurch offline waren, ist das Dokument semantisch auf dem Stand, wie ihn alle Teilnehmenden eines Raums haben wollen.

Die Intention von Änderungen über Teilnehmende hinweg hängt vom Zeitpunkt der Änderungen ab. Das stellt (room)service sicher. Wichtig ist, dass alle Operationen in der *publishDoc*-Funktion durchgeführt werden. So auch das Löschen aus einer Liste oder anderweitige Anpassungen:

```
room.publishDoc(doc => {
  delete doc.todos[1];
});
```

Auf diese Weise ist sichergestellt, dass alle Änderungen berücksichtigt werden und das Dokument am Ende aller Operationen nicht nur technisch in Ordnung ist, sondern auch semantisch aus Sicht aller Teilnehmenden eines Raums.

Der aktuelle Stand

Auf GitHub stehen weitere Beispiele für React und Vue.js zur Verfügung. Die Beispielanwendung zu Vue.js verdeutlicht, wie eine zufällig ausgewählte Zahl in einem Raum zur Verfügung gestellt werden kann, um die Daten dann in verschiedenen Browserfenstern anzuzeigen (siehe Bild 1).

Sobald der Zugang zur Alpha-Version freigeschaltet ist, wird ein Dokument zur Verfügung stehen, das den aktuellen und kurzfristig geplanten Entwicklungsstand beschreibt. Aktuell sind JSON-Dokumente umgesetzt, wie die bisherigen Beispiele auch gezeigt haben, und das mit der Unterstützung für JSON CRDT, Offline-Support und automatische Updates. Das lässt vermuten, dass in Zukunft auch andere Formate neben JSON unterstützt werden sollen.

Eine Liste verrät zudem, was verbessert werden könnte. Hier ein Ausschnitt:

- Ein Update soll zurückgewiesen werden können (reject), wenn es mit dem Datenschema in Konflikt steht.
- Die Latenz bei den JSON-Dokumenten und hoher Last kann verbessert werden.
- Mehrere Dokumente per Raum.
- Lesen und Schreiben per Raum.
- Rechtesystem auf Basis von Eigenschaften eines Dokuments

Darüber hinaus arbeitet das (room)service-Team an den Awareness-Funktionen, sodass Maus- und Text-Cursor und andere Eigenschaften übertragen werden können, inklusive einer ordentlichen Integration in Frontend-Frameworks wie React und Vue.js. Die Integration in vorhandene Texteditoren ist ebenfalls geplant – aber noch in weiter Ferne.

Fazit

Die Bibliothek (room)service adressiert einen wunden Punkt in vielen Plattformen und Anwendungen, die den Anspruch haben, unterschiedliche Benutzer auf den gleichen Daten arbeiten zu lassen. Diese Art der Kollaboration ist technisch nicht einfach zu realisieren, insbesondere wenn Awareness-Funktionen dazukommen sollen, zu denen zum Beispiel gehört, wo im Dokument sich gerade ein Benutzer befindet. Das alles ist prinzipiell mit der in diesem Artikel vorgestellten Bibliothek möglich.

Da sich (room)service aktuell in einer frühen Alpha-Phase befindet, ist es schwer abzuschätzen, wohin die Reise am Ende wirklich geht. Das Potenzial ist definitiv vorhanden und gebraucht wird die Bibliothek auch. Zum jetzigen Zeitpunkt hat sich (room)service ein „Gut“ und eine Empfehlung verdient, sich die Bibliothek auf jeden Fall einmal anzuschauen.

Wie in der Einleitung bereits angekündigt, bleibt dotnetpro an der weiteren Entwicklung dran. Dem Autor hatte beim Schreiben des Artikels etliche Ideen zu Aspekten, die von einem gut funktionierenden, verteilten und kooperativen Arbeiten profitieren würden. Die Liste mit den geplanten Funktionen liest sich auf jeden Fall schon vielversprechend. ■

[1] Die Website zu Figma, www.figma.com
[2] Die Website zu (room)service, www.roomservice.dev
[3] Der Begriff CRDT auf Wikipedia, www.dotnetpro.de/SL2004Frameworks1

Fabian Deitelhoff
promoviert am Graduiertenkolleg „User-Centred Social Media“ im Themenumfeld der Code Comprehension. Zudem ist er Autor, Entwickler, Trainer und Gründer von www.brickobotik.de. Erreichbar über www.fabiandeitelhoff.de oder **@FDeitelhoff**.

dnpCode A2004Frameworks