Technische Hochschule
Brandenburg
University of
Applied Sciences

Implementierung von Prozessen
-
Zum Thema Reisebüro

Fachbereichs Wirtschaft der
Technischen Hochschule Brandenburg

vorgelegt von:
Franck Derrick Kateu Fonga
Naura Mbigha
Mattias Le Prince Tamko
Nadia Ouertani
Zaur Abakarov

2. Semester

Betreuer: Prof. Dr. Kai Jander

Brandenburg/H., den 21. July 2023

Inhaltsverzeichnis

Abbildungsverzeichnis III

Tabellenverzeichnis IV

Abkürzungsverzeichnis V

1 Einleitung 1

2 Implementierung 3
2.1 Softwareauswahl 3
2.2 Prozessaufnahme 3
2.3 Prozessmodellierung 4

3 Anpassung des Serverdienstes und Test mit Postman 5
3.1 Zimmer im Hotel Buchen 5
3.2 Fahrzeug und Bahnreise buchen 7
3.3 Buchung stornieren bzw. Löschen. 9
3.4 Bonitätsprüfung 10
3.5 Statuts der Gebuchte Zimmer Ändern 11
3.6 Gesamte Rechnung durchführen 12
3.7 Zusammenfassung der Implementierte Endpoints 14

4 Herausforderungen 15

5 Zusammenfassung 16

Literaturverzeichnis 17

Anhang 18

Abbildungsverzeichnis

Abbildung 1: Darstellung des Reisebüros 4
Abbildung 2: Buchungen starten 5
Abbildung 3: Zimmer in einem Hotel buchen und speichern. 6
Abbildung 4: Test für die Buchung eines Zimmers. 7
Abbildung 5: Fahrzeug buchen..... 8
Abbildung 6:Test für die Buchung des Fahrzeuges..... 8
Abbildung 7: Buchung löschen..... 9
Abbildung 8: Test fürs Löschen einer Buchung 10
Abbildung 9: Code zur Bonitätsprüfung..... 11
Abbildung 10: Test für einen User über seine Bonität 11
Abbildung 11:Code zur Änderung des Status eines gebuchten Zimmers 12
Abbildung 12: Test über Änderung des Zimmerstatus 12
Abbildung 13: Totalpreis Berechnung 13
Abbildung 14: Test für Totalpreis in Postman..... 13

Tabellenverzeichnis

Tabelle 1: Implementierte Endpoints 14
Tabelle 2: Herausforderungen über das Projekt 15

Abkürzungsverzeichnis

API	Application Programming Interface
BPMN	Business Process Modelling Notation
CRUD	Create Read Update Delete
REST	Reprentational State Transfer

1 Einleitung

In der heutigen Geschäftswelt ist die effiziente Gestaltung und Implementierung bzw. Automatisierung von Geschäftsprozessen von entscheidender Bedeutung, um Wettbewerbsvorteile zu erlangen und den Erfolg eines Unternehmens sicherzustellen. Business Process Model and Notation (BPMN) ist eine weit verbreitete und standardisierte Methode zur Modellierung von Geschäftsprozessen. Camunda ist eine Open-Source-Plattform, die BPMN zur Prozessmodellierung und Workflow-Management verwendet.

Ziel dieses Projekts ist es, die Implementierung von Prozessen mit BPMN auf Camunda und die Entwicklung einer Serveranwendung mit einer REST-API in Node.js zu untersuchen. Die Kombination dieser Technologien ermöglicht es Unternehmen, ihre Geschäftsprozesse zu modellieren, zu automatisieren und zu überwachen, während sie gleichzeitig eine flexibel erweiterbare Serveranwendung bereitstellen, die über eine REST-API zugänglich ist.

Die Verwendung von BPMN in Kombination mit Camunda bietet eine Reihe von Vorteilen. Erstens ermöglicht es die visuelle Modellierung von Geschäftsprozessen mit BPMN eine klare Darstellung der verschiedenen Schritte, Entscheidungen und Aktivitäten, die in einem Prozess involviert sind. Dies erleichterten das Verständnis und die Zusammenarbeit zwischen Geschäfts- und IT-Teams. Camunda bietet Funktionen wie das Task-Management, die Zuweisung von Aufgaben an Benutzer oder Gruppen, die Bearbeitung von Formularen und die Integration von externen Systemen über Serviceaufrufe. Mit Camunda können Prozesse automatisiert werden, um eine schnellere Durchführung, geringere Fehlerquoten und eine effiziente Ressourcennutzung zu erreichen.

Die Entwicklung einer Serveranwendung mit einer REST-API in Node.js ergänzt die BPMN-Implementierung. Node.js ist eine leistungsstarke Plattform, die auf der JavaScript-Programmiersprache basiert und eine effiziente Verarbeitung von Anfragen ermöglicht. Durch die Bereitstellung einer REST-API kann die Serveranwendung nahtlos mit anderen Systemen und Anwendungen kommunizieren und ermöglicht so die Integration und den Austausch von Daten.

Im Rahmen dieses Projekts werden wir uns mit der Modellierung von Geschäftsprozessen mit BPMN befassen, diese Prozesse in Camunda implementieren und die Serveranwendung mit einer REST-API in Node.js entwickeln. Wir werden untersuchen, wie BPMN-Prozesse in Camunda ausgeführt und deployt werden, wie Aufgaben verwaltet und delegiert werden und wie die Kommunikation mit externen Systemen über die REST-API erfolgt.
Für dieses Projekt wurde ein vierköpfiges Team gebildet, das sich bereits im ersten Semester mit zwei sehr wichtigen Modulen befasste, nämlich Advanced Software Engineering und Prozessmodellierung.
Diese beiden Module befassten sich jeweils mit der Entwicklung einer REST API, die mit einer Webanwendung gekoppelt wurde, und der Analyse, Modellierung und Optimierung von Prozessen. Diese beiden Module bilden die notwendige Grundlage für die Durchführung dieses Projekts.
Bei der Durchführung haben wir gelernt, Prozess + Server + API (zum Ausführen der Methoden vom Server) zu kombinieren.

2 Implementierung

2.1 Softwareauswahl

Die Wahl der Software ist bei der Umsetzung von IT-Projekten von entscheidender Bedeutung. Bei diesem Projekt haben wir uns entschieden, mit **Node.js** auf **Visual Studio Code** zu arbeiten.
Für die Modellierung und das Deployment des Prozesses empfahl uns der Lehrer hingegen vor allem die Verwendung der Software **Camunda-Modeller** Version 5 und die entsprechende Plattform auf Version 7. Zum Testen alle Endpoint wurde **Postman** als angepasste Software ausgewählt.

2.2 Prozessaufnahme

Da die Konzeption des Projekts prozessorientiert und mit Verbesserungsmöglichkeiten versehen war, gingen wir auch bei den Prozessaufnahmen schrittweise vor.
Während der Modellierung des Prozesses wurden mehrere Änderungen vorgenommen, da die Programmierung von server.js unnötig komplex sein sollte. Nach der Prozessaufnahme stellt sich im Wesentlichen heraus, dass es in diesem Projekt um einen Kunden geht, der mit dem Ticket eines Reisebüros ein Hotelzimmer in einer Stadt buchen will, in der er gerne Tourismus machen möchte. Im Buchungsprozess gibt es die Möglichkeit, auch ein Auto oder ein Zugticket zu buchen, monatlich oder auf einmal zu bezahlen. Die monatliche Zahlung wird vom Filialleiter genehmigt. Wenn es ein Problem gibt, kann der Prozess interomputiert werden, der Kunde hat die Möglichkeit, verschiedene Zimmertypen in verschiedenen Hotels in verschiedenen Städten zu buchen.

2.3 Prozessmodellierung

Wie bereits erwähnt, wurden einige Details berücksichtigt, was zu einigen Veränderungen auf der Prozessebene führte.
Der endgültige Prozess sieht wie folgt aus:

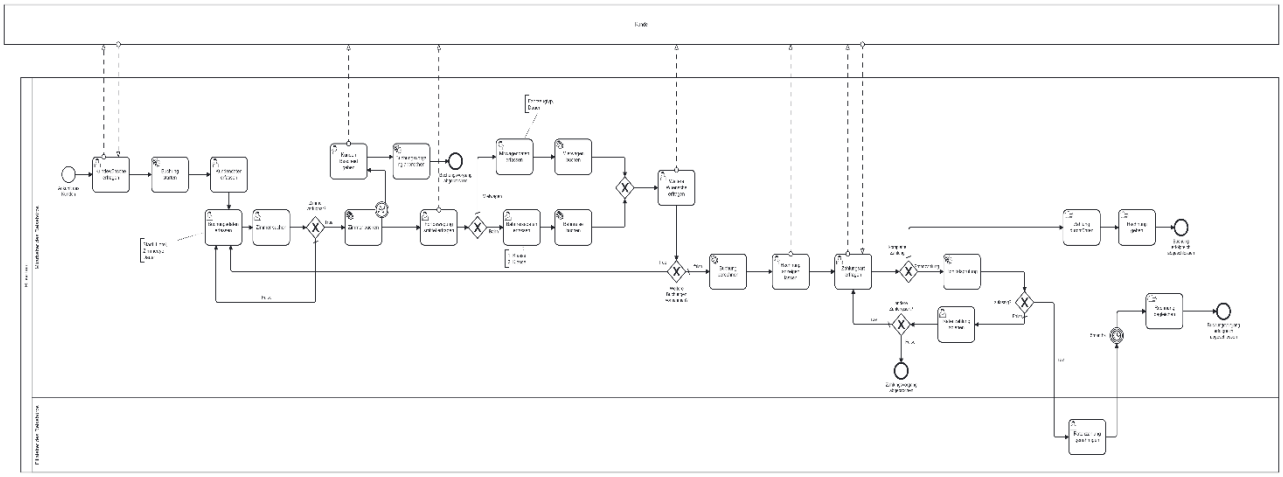


Abbildung 1: Darstellung des Reisebüros

3 Anpassung des Serverdienstes und Test mit Postman

Im Rahmen dieses Projekt wurde eine Rest API vorgegeben und es müsste nach den Funktionalitäten des Prozesses angepasst werden.
Die modellierten Funktionen sind meisten mit dem POST-Methoden verbunden.

Buchung starten
Diese Anfragen löst unseren Prozess aus, in dem der Mitarbeiter des Reisebüros den Kundenwünschen, Kunden Daten im System erfasst. Der Code Teil sieht wie folgendes aus.

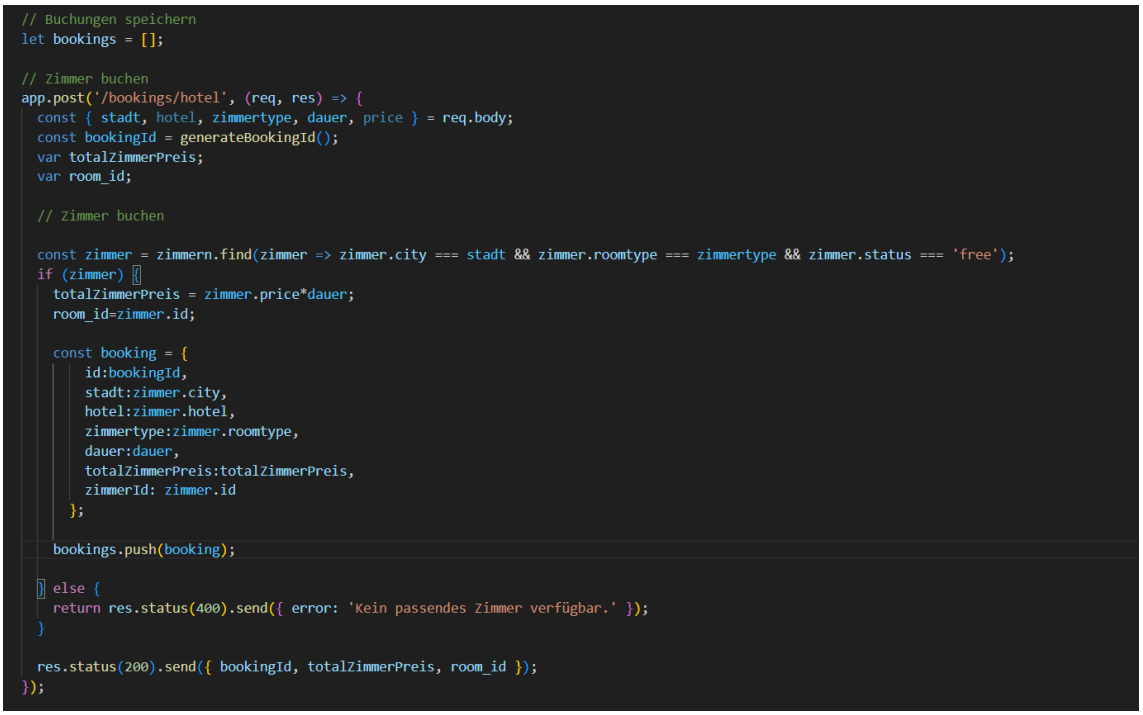
```
// Buchung starten
app.post('/booking/start', (req, res) => {
    let bookingRef = uuidv4.uuid()
    bookings.set(bookingRef, [])
    return res.json({ bookingRef: bookingRef })
})
```

Abbildung 2: Buchungen starten

Bookings ist ein Array, indem es alle Buchungen hinzugefügt werden und am Ende daraus eine Berechnung für die Rechnung des Kunden.

3.1 Zimmer im Hotel Buchen

Die Zimmer werden ersten nach der Stadt, und Hotelname initialisiert und anschließend gebucht. Es besteht die Möglichkeit entweder ein Einzelzimmer, Doppelzimmer oder eine Suite zu buchen. Die Preise sind beliebig vorgegeben. Die Parameter zum initialisieren von Zimmer sind in eine Array *zimmer[ ]* gespeichert. Diese Parameter sind (id, Roomtype, preis, Dauer, Hotel und Stadt bzw. City). Code Teil dafür ist in folgende Abbildung zu sehen.



```
// Buchungen speichern
let bookings = [];

// Zimmer buchen
app.post('/bookings/hotel', (req, res) => {
  const { stadt, hotel, zimmertype, dauer, price } = req.body;
  const bookingId = generateBookingId();
  var totalZimmerPreis;
  var room_id;

  // Zimmer buchen

  const zimmer = zimmern.find(zimmer => zimmer.city === stadt && zimmer.roomtype === zimmertype && zimmer.status === 'free');
  if (zimmer) {
    totalZimmerPreis = zimmer.price*dauer;
    room_id=zimmer.id;

    const booking = {
        id:bookingId,
        stadt:zimmer.city,
        hotel:zimmer.hotel,
        zimmertype:zimmer.roomtype,
        dauer:dauer,
        totalZimmerPreis:totalZimmerPreis,
        zimmerId: zimmer.id
      };

    bookings.push(booking);

  } else {
    return res.status(400).send({ error: 'Kein passendes Zimmer verfügbar.' });
  }

  res.status(200).send({ bookingId, totalZimmerPreis, room_id });
});
```

Abbildung 3: Zimmer in einem Hotel buchen und speichern.

Zum Testen, ob diese POST-Anfrage funktioniert, wurde Postman verwendet. Der Body wurde als JSON eingegeben und das Ergebnis auch ist als JSON-Datei im System bzw. Array gespeichert. Der Client sendet eine POST-Anfrage an die API mit den erforderlichen Daten für die Zimmerbuchung, einschließlich Stadt, Hotel, Zimmertyp und Dauer. Die API validiert die Daten, überprüft die Verfügbarkeit des gewünschten Zimmers und führt die Buchung durch. Der gebuchte Raum wird als "belegt" markiert und die Buchungsdaten werden gespeichert. Die API sendet eine Antwort mit der Buchungs-ID und dem Gesamtpreis an den Client.

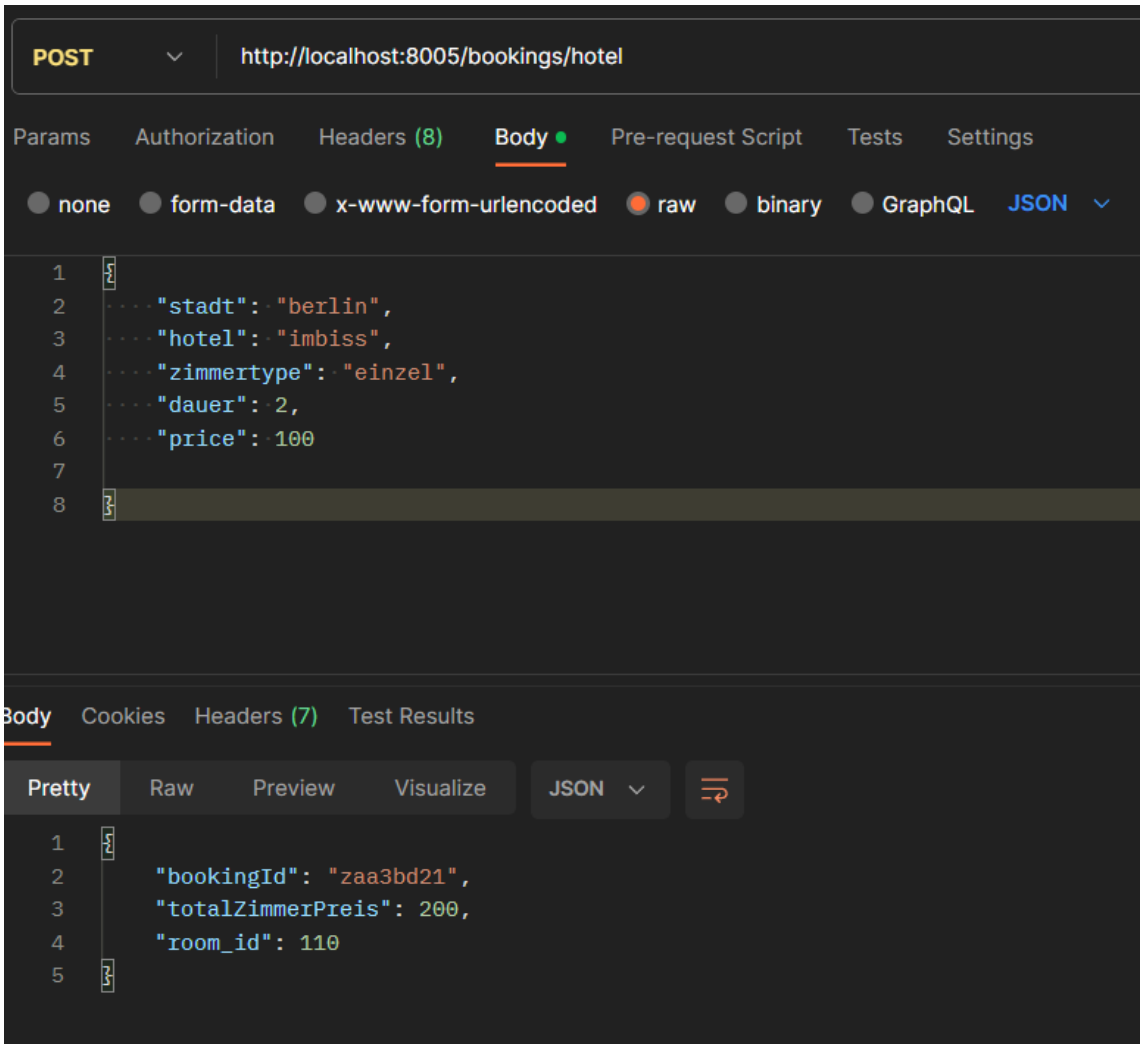


Abbildung 4: Test für die Buchung eines Zimmers.

3.2 Fahrzeug und Bahnreise buchen

Das Prinzip ist das gleiche, also das heißt das die bahn und Fahrzeuge Parameter werden initialisiert und gespeichert. Folgende Abbildung zeigt den Test mit eine POST-Anfrage und es ist auch zu sehen das die Anfrage funktioniert und ein freies Fahrzeug wurde für 5 Tage bzw. für 80€ pro tag für den Kunde mit eine entsprechende bookingId gebucht.

```
app.post('/bookings/fahrzeug', (req, res) => {
  const { fahrzeugTyp, dauer } = req.body;
  const bookingId = generateBookingId();
  var totalPrice;
  var fahrzeugId;

    // Fahrzeug mieten
    const fahrzeug = fahrzeuge.find(fahrzeug => fahrzeug.type === fahrzeugTyp && fahrzeug.status === 'free');
    if (fahrzeug) {
        const booking = {
            id: bookingId,
            fahrzeugID: fahrzeug.id,
            fahrzeugTyp: fahrzeug.type,
            dauer: dauer,
            price: fahrzeug.price,
            totalPrice: fahrzeug.price * dauer
        };

       // fahrzeug.status = 'free';
        fahrPreis = fahrzeug.price * dauer;
        bookings.push(booking);
        console.log(bookings);
    } else {
        return res.status(404).send({ error: 'Kein passendes Fahrzeug verfügbar.' });
    }

    res.status(200).send({ bookingId, totalPrice, fahrzeugId });
});
```

Abbildung 5: Fahrzeug buchen

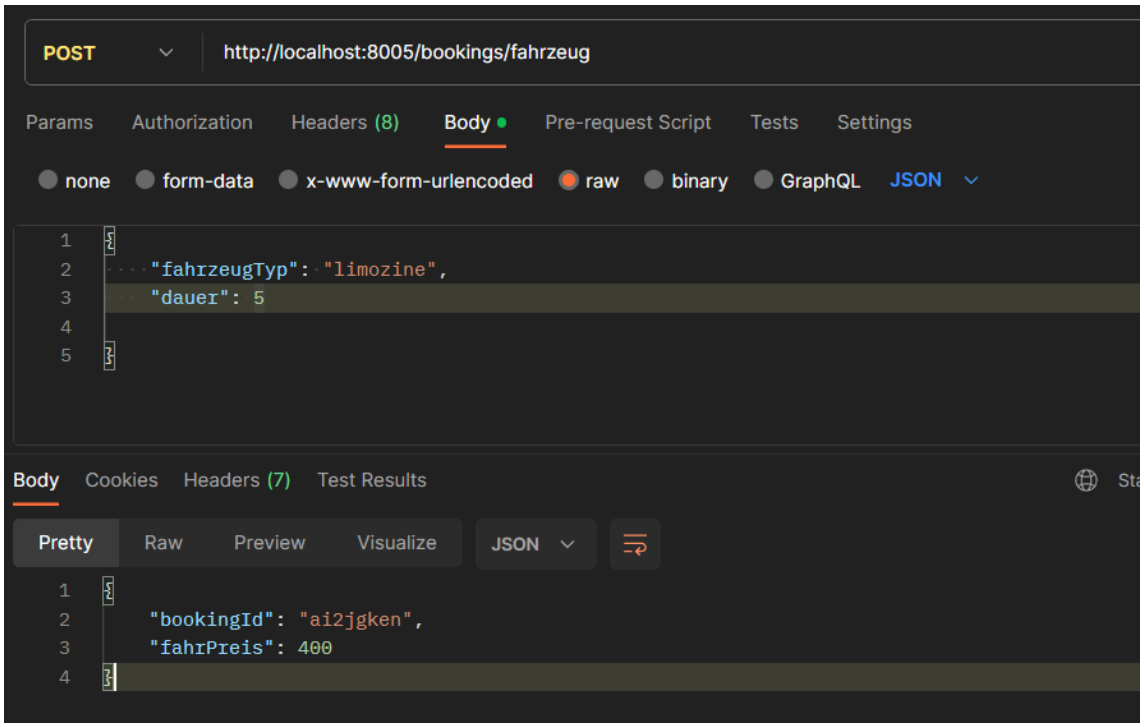


Abbildung 6:Test für die Buchung des Fahrzeuges

3.3 Buchung stornieren bzw. Löschen.

Es wurde auch im Rahmen dieses Projekt über weitere Implementierungsmöglichkeiten überlegt. Der Kunden könnte selbst auf alle Buchungen verzichten und keine Buchungen mehr vornehmen, in dem falls wäre eine Funktion zum Löschen diese Buchungen notwendig. Dafür sollte die entsprechende bookingId's bekannt sein.
Folgende Code-Teil ist zum Löschen der Buchungen vorgesehen.

```
// Buchung stornieren
app.delete('/bookings/:bookingId', (req, res) => {
  const { bookingId } = req.params;

  const index = bookings.findIndex(booking => booking.id === bookingId);

  if (index !== -1) {
    const booking = bookings[index];

    if (booking.zimmertype) {
      const zimmer = zimmern.find(zimmer => zimmer.id === booking.zimmerId);
      if (zimmer) {
        zimmer.status = 'free';
        const bookingIndex = zimmer.bookings.indexOf(bookingId);
        if (bookingIndex !== -1) {
          zimmer.bookings.splice(bookingIndex, 1);
        }
      }
    }

    if (booking.fahrzeugTyp) {
      const fahrzeug = fahrzeuge.find(fahrzeug => fahrzeug.id === booking.fahrzeugId);
      if (fahrzeug) {
        fahrzeug.status = 'free';
      }
    }

    bookings.splice(index, 1);
    res.status(200).send({ message: 'Buchung erfolgreich storniert.' });
  } else {
    res.status(404).send({ error: 'Buchung nicht gefunden.' });
  }
});
```

Abbildung 7: Buchung löschen

Für den Test dieser Anfrage ist kein Body erforderlich.

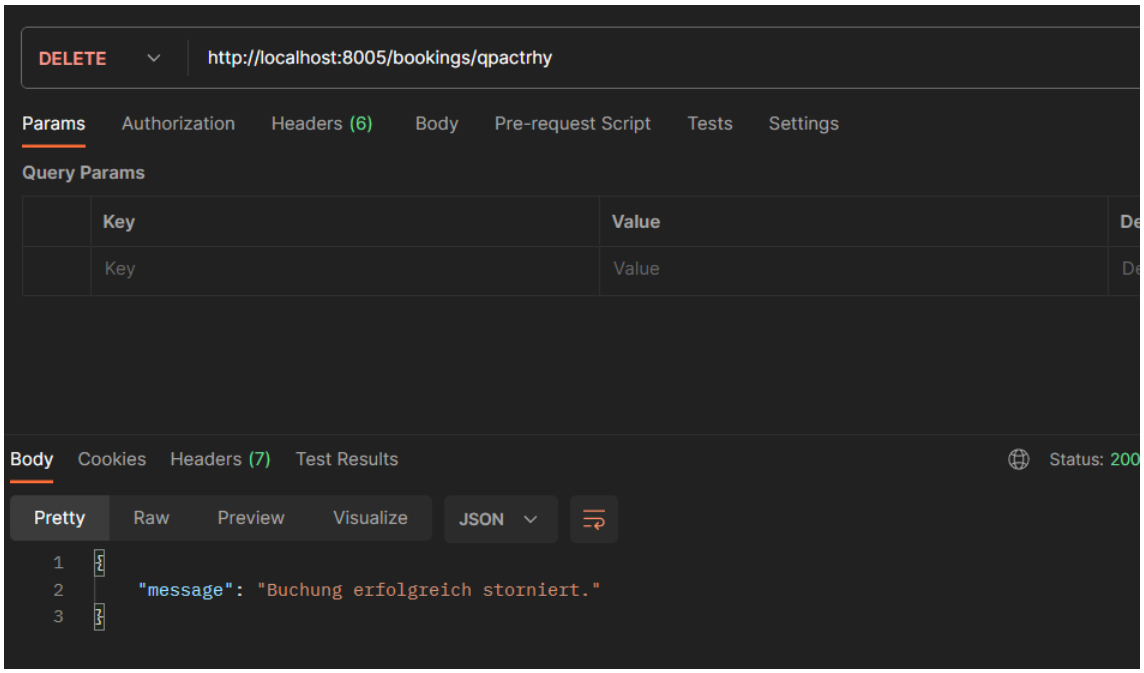


Abbildung 8: Test fürs Löschen einer Buchung

3.4 Bonitätsprüfung

Der Code enthält ein Array initialisierten Users und ihrer Bonität. Dieser Array kann durch eine echte Datenbank ersetzt werden, je nachdem, wie deine Anwendung aufgebaut ist.

Der API-Endpunkt ***/bookings/checkbonitaet/:username*** nimmt den Nachname des Users entgegen und ruft die Bonität des Benutzers aus der Datenbank ab. Der parseInt-Aufruf wird verwendet, um sicherzustellen, dass die ID des Benutzers eine Ganzzahl ist. Die Array.prototype.find-Methode wird verwendet, um den Benutzer in der Datenbank anhand seiner nachname zu finden. Wenn der Benutzer nicht gefunden wird, wird eine Fehlermeldung mit dem Statuscode 404 zurückgegeben. Die Bonität des Benutzers wird aus dem gefundenen Benutzerobjekt abgerufen. Anhand der Bonität wird eine Bonitätsprüfung (Abgleich) durchgeführt und eine entsprechende Antwort zurückgegeben. In diesem Beispiel werden verschiedene Meldungen je nach Bonität zurückgegeben. Die Antwort wird mit dem entsprechenden Statuscode (200) und einer JSON-Nachricht zurückgegeben.

Diese JSON-Antworte wurden auch eben in Postman getestet und folgendes sehen die Ergebnisse mit entsprechendem Code aus.

```
app.get('/bookings/checkbonitaet/:username', (req, res) => {
const user = users.find(user => user.user_nachname === req.params.username);

  if (user) {
    res.status(200).send({ bonitaet: user.bonitaet });

  } else {
    res.status(404).send({ message: 'User with Username ' + req.params.username + ' not found ' });
  }

});
```

Abbildung 9: Code zur Bonitätsprüfung

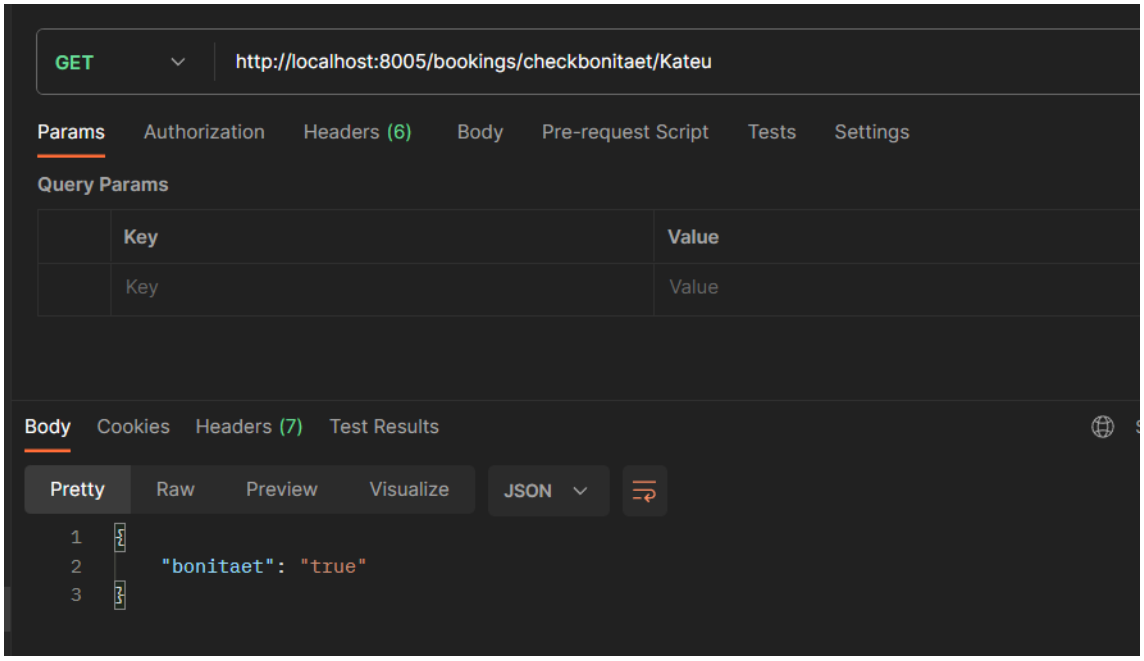


Abbildung 10: Test für einen User über seine Bonität

3.5 Statuts der Gebuchte Zimmer Ändern

Dieser Endpunkt erwartet eine PUT-Anfrage an die Route */bookings/:bookingId/status,* wobei *:bookingId* die ID der Reservierung ist, für die der Status des Zimmers geändert werden soll. Der Code durchsucht die Buchung mithilfe der Buchungs-ID und findet das entsprechende Zimmer mithilfe der Zimmer-ID in der Buchung. Anschließend wird der Status des Zimmers auf "belegt" gesetzt. Eine erfolgreiche Antwort mit dem Statuscode 200 und einer Erfolgsmeldung wird zurückgesendet. Wenn die Reservierung

oder das Zimmer nicht gefunden wird, wird ein Fehler mit dem entsprechenden Statuscode zurückgegeben.

```
app.put('/bookings/:bookingId/status', (req, res) => {
  const bookingId = req.params.bookingId;
  // Finde die Buchung anhand der Buchungs-ID
  const booking = bookings.find(booking => booking.id === bookingId);
  if (!booking) {
    return res.status(404).send({ error: 'Buchung nicht gefunden.' });
  }
  // Finde das zugehörige Zimmer anhand der Zimmer-ID in der Buchung
  const zimmer = zimmern.find(zimmer => zimmer.id === booking.zimmerId);

  if (!zimmer) {
    return res.status(404).send({ error: 'Zimmer nicht gefunden.' });
  }
  // Setze den Status des Zimmers auf "belegt"
  zimmer.status = 'belegt';
  res.status(200).send({ message: 'Zimmerstatus erfolgreich auf "belegt" gesetzt.' });
});
```

Abbildung 11:Code zur Änderung des Status eines gebuchten Zimmers

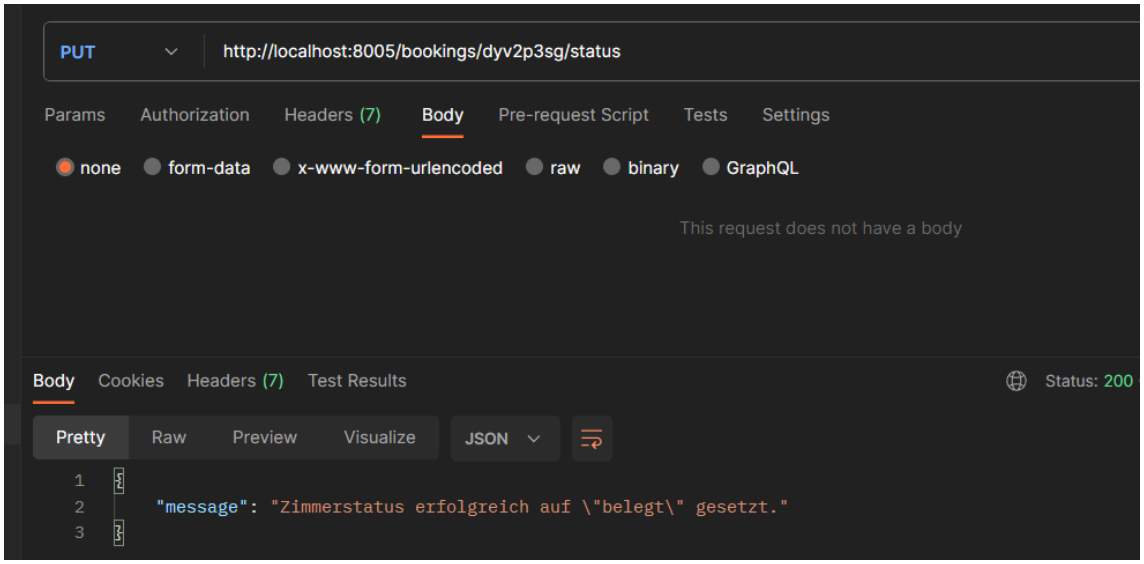


Abbildung 12: Test über Änderung des Zimmerstatus

3.6 Gesamte Rechnung durchführen

Da bei jedem Endpunkt, der implementiert wurde, d. h. bei jeder Reservierung, die vorgenommen wurde, andere Parameter ins Spiel kommen, die auf den Fotos oben zu sehen sind, ist es wichtig, dass Sie sich an die Regeln halten. Die Gesamtrechnung wird also auf der Grundlage des gewählten Pfades und der Wahl des Kunden erstellt und die Rechnung wird ebenfalls vom Kunden abhängen.

Der nachfolgende Test auf Postman zeigt, dass die Arrays, die für die verschiedenen Reservierungen deklariert wurden, funktionieren.

```
345
346  app.get('/bookings/invoice', (req, res) => {
347    let totalPrice = 0;
348
349    bookings.forEach(booking => {
350      totalPrice += booking.totalPrice;
351    });
352
353    res.status(200).send({ totalPrice });
354  });
```

Abbildung 13: Totalpreis Berechnung

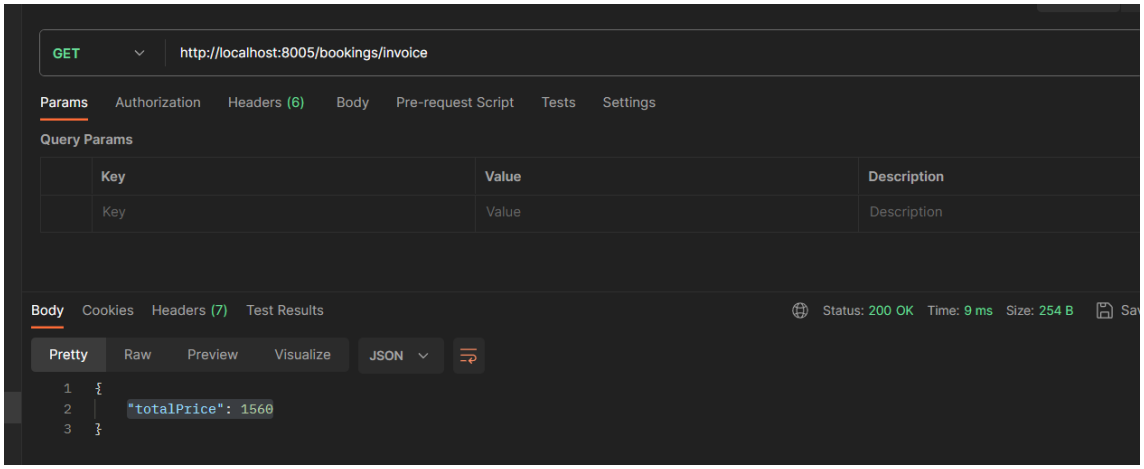


Abbildung 14: Test für Totalpreis in Postman

3.7 Zusammenfassung der Implementierte Endpoints

Im Folgenden finden Sie eine Übersicht der implementierten Endpunkte.

1- POST	http://localhost:8005/booking/start	Buchung Starten
2- POST	http://localhost:8005/booking/hotel	Zimmmer Buchung
3- POST	http://localhost:8005/booking/bahn	Bahnreise Buchung
4- POST	http://localhost:8005/booking/fahrzeug	Fahrzeug Buchung
5- POST		Buchung Stornieren
6- DELETE	http://localhost:8005/bookings/bookingId	Buchung Löschen
7- PUT	http://localhost:8005/rooms/:room_id/status	Status des gebuchten Zimmers ändern
8- GET	http://localhost:8005/bookings/checkbonitaet/:username	Bonitätsprüfung
9- GET	http://localhost:8005/bookings/invoice	Gesamtpreis Berechnung

Tabelle 1: Implementierte Endpoints

4 Herausforderungen

Im Rahmen dieses Projekts war es nur einfach, die im ersten Semester erlernte REST-Methode und die in diesem Semester erlernte BPMN zu kombinieren. Die Kombination der beiden erfordert viel Arbeit, um ein gutes Ergebnis zu erzielen. Im Großen und Ganzen gibt die Tabelle unten einen Überblick über die Schwierigkeiten, die wir während des Projekts hatten. Die Gruppe lernte mehr über die Art von Technologien, nämlich Camunda-Modeller, Node.Js, Express.Js, Camunda-Plattform und wie eine Kombination dieser Technologien möglich ist.

1	Modellierung des Prozesses
2	Services Task in BPMN mit Konnektoren gut konfigurieren
3	Code mit BPMN einbinden
4	Abspielen lassen und Test
5	Einzelne Teste mit Postman durchzuführen
6	Das gebuchte Zimmern auf „Besetzt“ setzen
7	Bonitätsprüfung
8	GUI fürs Anzeigen der Bekommene Werten

Tabelle 2: Herausforderungen über das Projekt

Es war schwierig für die Gruppe, eine Benutzeroberfläche für die Werte im Zusammenhang mit der Abrechnung und Buchung zu erstellen, daher haben wir uns für einen Test mit der Postman-Software entschieden, um zu sehen, ob unsere Endpoints in der Camunda-Plattform funktionieren.

5 Zusammenfassung

Das Projekt zielt darauf ab, eine Lösung zur Automatisierung des Buchungsprozesses für Zimmer und Fahrzeugmieten in der Tourismusbranche zu entwickeln. Dabei wird die BPMN (Business Process Model and Notation)-Methodik verwendet, um den Prozess zu modellieren und zu optimieren, während eine Server-API in JavaScript die erforderlichen Funktionen bereitstellt. BPMN-Modellierung für die gewünschte Funktionen werden mithilfe der BPMN-Notation modelliert. Die Prozessmodelle beschreiben den Ablauf der Buchungen, die beteiligten Aktivitäten und die Daten, die zwischen den Schritten ausgetauscht werden.

Server-API: Eine Server-API wird in JavaScript implementiert, um die erforderlichen Endpunkte für die Funktionen bereitzustellen. Die API verwendet Express.js als Framework und bietet RESTful-Endpunkte für die erforderiche Buchungen und andere damit verbundene Aktionen. In diesem Projekt wurden Arrays verwendet, um die verfügbaren Zimmer und Fahrzeuge zu speichern. Bei Buchungen werden die Daten aktualisiert und entsprechende Validierungen durchgeführt.

Gesamtrechnung: Die Server-API berechnet die Gesamtrechnung basierend auf die vorgenommenen Buchungen. Die Preise und Dauer der Buchungen werden berücksichtigten, um den endgültigen Betrag zu ermitteln.

Der Workflow für das Zimmerbuchungsszenario könnte folgendermaßen aussehen:

Zum Schluss kombiniert das Projekt die Modellierungskraft von BPMN mit der Flexibilität und Skalierbarkeit einer Server-API in JavaScript. Es ermöglicht eine nahtlose Automatisierung des Buchungsprozesses in der Tourismusbranche und bietet eine robuste Lösung für die Verwaltung und Berechnung von Buchungsdaten

Literaturverzeichnis

Anhang

- Source code: Server.js
- BPMN-Prozess