Hochschule Reutlingen

Herman-Hollerith-Zentrum Böblingen

Elective Cloud based Web Application Development

Dokumentation Einzelarbeit

Erstellung einer Hotel-Website in der Cloud

14. Juli 2024

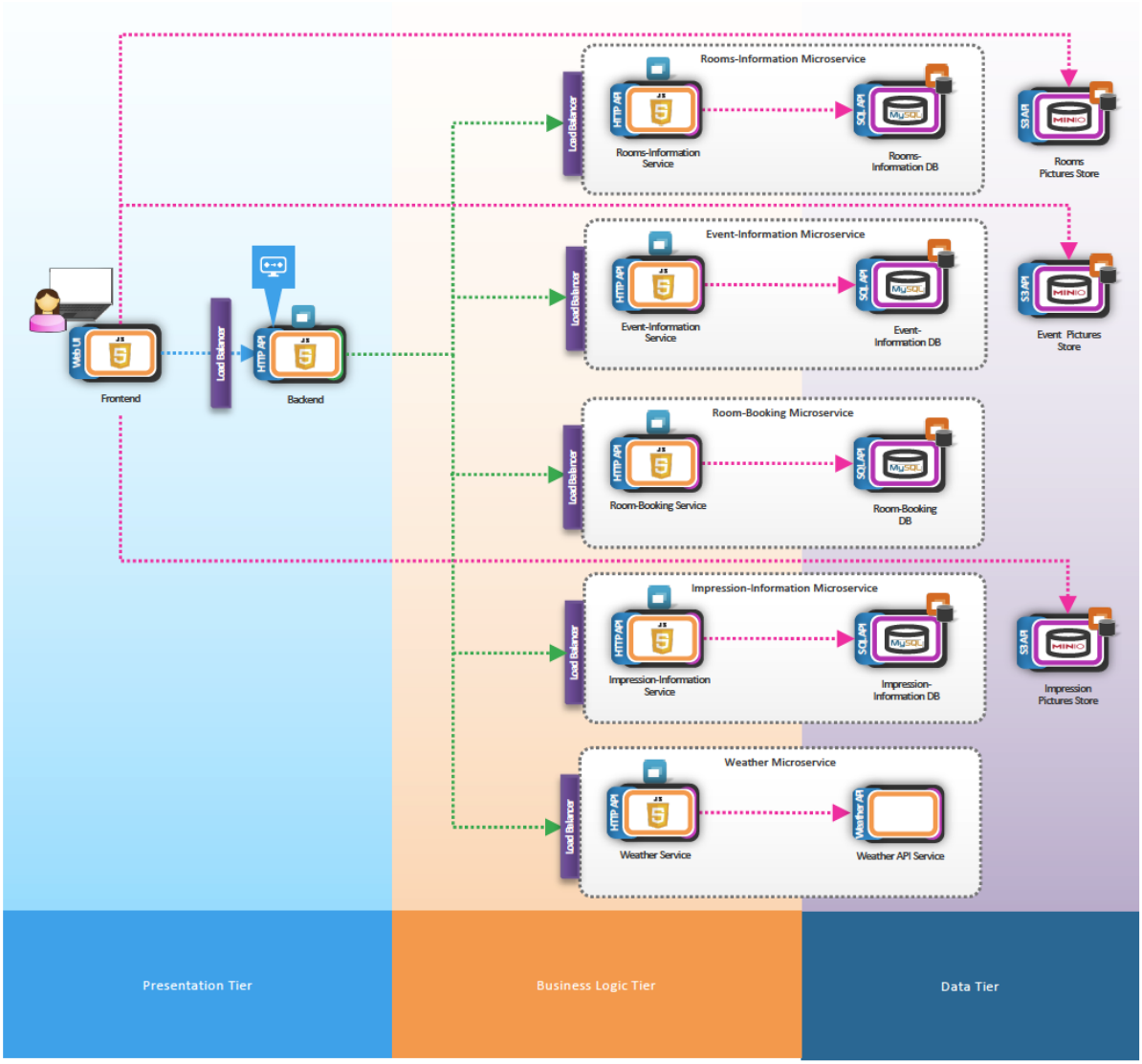
Marcel Thomas: 766829

Betreuer:

Prof. Uwe Breitenbücher

Table of Contents

I. Logical System Architecture



II. Architecture Component Descriptions

ACD1: Backend

Das Backend ist die zentrale Komponente zwischen Client und den Microservices. Primär ist dieses für das Routen und Verarbeiten von Anfragen des Clients verantwortlich. Entsprechend der Anfrage des Clients auf bestimme Komponenten der Anwendung, kann das Backend diese an die entsprechend verantwortlichen Microservices weiterleiten. Mittels der gewonnenen Informationen durch die Microservices kann dann das Rendering des Frontends stattfinden. Hierbei wird EJS als Technologie eingesetzt. In dieser Anwendung spricht man also von einem serverseitigen Rendering. Das Backend beinhaltet keinerlei Businesslogik, ist eine stateless Component und kann somit beliebig skaliert werden.

ACD2: Rooms-Information Microservice

Der Rooms-Information Microservice ist verantwortlich für die Informationen der verschiedenen Hotelzimmer des fiktiven Hotels. Dabei besitzt dieser Service eine Verbindung zu einer MySQL Datenbank, um hier die aktuellen Informationen zu den Zimmern zu erhalten. Der Service besitzt zwei Funktionalitäten, mit dieser er die Informationen an das Backend übergeben kann. Die erste gibt alle Hotelzimmer in der DB aus, während die zweite nur die ersten vier Treffer in der DB berücksichtigt. Hiermit können die Usecases des „Hompage-Previews“ mit nur vier exemplarischen Treffern und die „Hotelzimmer-Buchungsseite“ mit allen Zimmern ermöglicht werden. Die Datenlast für die Homepage kann somit verringert werden, da nicht alle Zimmer an das Backend übermittelt werden müssen. Der Datenaustausch erfolgt im JSON-Format.

ACD3: Event-Information Microservice

Der Event-Information Microservice ist verantwortlich für die Informationen der verschiedenen Events in der Nähe des fiktiven Hotels. Dabei besitzt dieser Service ebenfalls eine Verbindung zu einer MySQL Datenbank, um hier die aktuellen Informationen zu den Events zu erhalten. Der Service besitzt gleichermaßen zwei Funktionalitäten, mit dieser er die Informationen an das Backend übergeben kann. Die erste gibt alle Events in der DB aus, während die zweite nur die ersten vier Treffer in der DB berücksichtigt. Hiermit können die Usecases des „Hompage-Previews“ mit nur vier exemplarischen Treffern und die „Event-Übersichts-Seite“ mit allen Events ermöglicht werden. Die Datenlast für die Homepage kann verringert werden, da nicht alle Zimmer an das Backend übermittelt werden müssen. Der Datenaustausch erfolgt einheitlich im JSON-Format.

ACD4: Room-Booking Microservice:

Der Room-Booking Microservice wird für die Buchung eines Hotelzimmers verwendet. Dabei wird mittels POST-Request das jeweilige Zimmer ermittelt, welches gebucht werden soll und zusammen mit Kundeninformationen, wie des Namens und der E-Mail (exemplarische Umsetzung mit zwei Informationen) an den Service übermittelt. Die Kommunikation erfolgt hierbei vom Frontend über das Backend bis hin zum Microservice selbst. Dieser legt die Bestellinformationen in einer MySQL Datenbank ab und sichert somit die Bestellung. Bei erfolgreicher Sicherung der Bestellung wird über das Backend zurück zum Frontend eine erfolgreiche Bestätigung übermittelt, sodass der Kunde ein direktes Feedback zu seiner Bestellung erhält. Der Datenaustausch findet im JSON-Format statt. Diese Umsetzung ermöglicht es weiteren Services auf die Bestellinformationen zuzugreifen und damit weitere Implementierungen vornehmen zu können.

ACD5: Impression-Information Microservice:

Der Impression-Information Microservice ist verantwortlich für die Kundenrezensionen des fiktiven Hotels. Dabei besitzt dieser Service ebenfalls eine Verbindung zu einer MySQL Datenbank, um hier die aktuellen Informationen zu den Kundenrezensionen abzurufen. Der Datenaustausch erfolgt einheitlich im JSON-Format.

ACD6: Weather Microservice:

Der Weather Service ist für die Darstellung des heutigen Wetters verantwortlich und soll eine Wettervorhersage exemplarisch simulieren. Dieser stellt eine Verbindung zu einem existieren Wetter Service her, der von einem externen Provider zur Verfügung gestellt wird. Mittels dieses weatherapi-Services[1], der mittels HTTP-Request angesteuert werden kann, können somit Wetterdaten erhalten und an das Backend übermittelt werden. Die Kommunikation erfolgt über HTTP an den externen Service, während der interne Datenaustausch im JSON-Format stattfindet.

[1] Homepage weatherapi: https://www.weatherapi.com/

III. Architecture Decision Records

ADR1: Verwendung der Microservice-Architektur

Die Wahl der Microservice-Architektur ermöglicht viele Vorteile für dieses Fallbeispiel. Zum einen kann eine Aufteilung der Funktionen realisiert werden während zum anderen Komponenten unabhängig voneinander entwickelt, bereitgestellt und skaliert werden können. Dies hat eine Optimierung von Ressourcen zur Folge. Zudem können individuelle Komponenten wiederverwendet werden, ohne dabei zu einer unnötigen Duplizierung von Code zu führen. Im Falle der cloudbasierten Webanwendung können Verantwortlichkeiten getrennt und die Fehlertoleranz erhöht werden. Zudem bietet diese Architektur die Verwendung unterschiedlicher Technologien an. Es können je nach Microservice und Funktionalität verschiedene Programmiersprachen oder Technologien zur Geltung kommen und somit „das Beste aus allen Welten“ in einer Architektur zusammengeführt werden.

ADR2: Backend als API-Gateway

Das Backend dient in dieser Architektur als zentrales Bindeglied zwischen den Microservices und den Anfragen des Clients. Jede Anfrage wird ausschließlich über das Backend geleitet und von dort aus entsprechen an den richtigen Microservice navigiert. Hiermit wird ein API-Gateway simuliert, welches ausschließlich zum Routing benutzt und ohne Businesslogik implementiert wird. Es kann somit eine logische Trennung von Funktionalitäten erreicht und die Wartbarkeit des Systems verbessern werden. Änderungen in den Microservices stellen hierbei keine Korrelation zu anderen Services oder dem Backend dar. Die Kommunikation erfolgt ausschließlich über HTTP-Requests und ermöglicht die Kommunikation innerhalb des Systems. Hierbei wird zum Datenaustausch standardisiert das JSON-Format verwendet.

ADR3: Server-Side Rendering

Das serverseitige Rendering im Backend ermöglicht eine schnelle Darstellung der Benutzeroberfläche. HTML-Inhalte werden hierbei bereits fertig generiert an den Client geschickt. Dies reduziert die Ladezeiten und verbessert die Performance der Nutzer. Mittels EJS, bereits initialisierten UI-Templates und den Informationen der Microservices können fertige Inhalte an den Kunden übergeben werden. Dies ermöglicht eine Konsistent des UIs für verschiedene Nutzer und deren Endgeräte und bietet eine plausible Lösung für die Darstellung von dynamischen Inhalten. Für die Entwicklung sind Anpassungen nur an den EJS-Templates nötig, welche wenig Korrelationen aufweisen und modular untergleidert sind.

ADR4: Verwendung der MySQL Datenbank

Die Daten für die Microservices werden in einer relationalen MySQL Datenbank gespeichert. Diese Entscheidung wurde getroffen, da die Daten in einer relationalen Datenbank klar strukturiert sind und dies notwendig ist, um Abfragen über verschiedene Merkmale der Datensätze auszuführen sowie ggf. unterschiedliche Datensätze miteinander zu verknüpfen. MySQL bietet die erforderliche Flexibilität und Leistungsfähigkeit, um diese Anforderungen effizient zu erfüllen und eine Verwaltung der erforderlichen Daten für die Microservices zu gewährleisten.

ADR6: Benutzung des BLOB Storage MinIO

Die Wahl von MinIO als Blob Storage basiert auf dessen Fähigkeit, große Mengen unstrukturierter Daten effizient zu verarbeiten. Durch die Auslagerung der Bildspeicherung an MinIO können die Microservices entlastet werden, was zu einer verbesserten Performance führt, da der Zugriff auf Bilder direkt über das Frontend erfolgt. Die Bildreferenzen werden in den relationalen Datenbanken der Services gespeichert, die eine strukturierte und effiziente Verwaltung der Bilddaten ermöglichen.

ADR7: Horizontale Skalierung und Loadbalancing

Die strikte Trennung der Microservices ist entscheidend für das horizontale Skalieren. Services können unabhängig voneinander skaliert werden und damit unterschiedlichen Workloads gegenwirken. Der Cloud Provider kann bei steigender Last automatisiert zusätzliche Instanzen des Services bereitstellen, die über einen Load Balancer effizient ausgelastet werden. Unpredictable Workloads können somit abgefangen und die Verfügbarkeit der Services gewährleistet werden. Die Anwendung ist dynamisch für jegliche Zahlen an Nutzern vorbereitet und erreicht mittels Cloud Provider eine gesteigerte Availability.

ADR8: Database per Service

Die Implementierung des “Database per Service”-Musters unterstützt eine modulare und unabhängige Microservice-Architektur. Jeder Microservice verwaltet seine eigene Datenbank. Dies ermöglicht es jedem Service, autonom zu handeln und Änderungen an den Datenbanken durchzuführen ohne andere Services zu beeinträchtigen. Doppelte Schreiboperationen auf eine Datenbank können somit gezielt vermieden werden, was die Sicherheit und die Komplexität des Systems bei Fehler reduziert. Die Vorteile dieser Architektur sind, dass jeder Service isoliert ist und unabhängig entwickelt, bereitgestellt und skaliert werden kann. Fehler in einem Service beeinflussen nicht die Daten oder die Verfügbarkeit anderer Services und sind keine Gefährdung für die Applikation.

IV. Deployment Decision Records

DDR1: Micorservices / Backend auf AWS Elastic Beanstalk

Eine horizontale Skalierung für das Backend und die einzelnen Microservices ist für einen Deploy für eine Vielzahl an Kunden essentiell. AWS Elastic Beanstalk könnte hier eine geeignete PaaS Lösung sein, um Webanwendungen bereitzustellen, ohne dass die Infrastruktur selbstständig verwaltet werden muss. Denn Elastic Beanstalk kümmert sich um die Bereitstellung und Skalierung der Services automatisiert. Zudem bietet Elastic Beanstalk eine nahtlose Integration zu den anderen AWS-Services an, welche die Entwicklung und das Deployment weiterhin vereinfachen. Für einen Einstieg in den AWS Deploy mit der Hotelwebsite wäre Elastic Beanstalk eine nennenswerte Lösung.

DDR2: AWS Managed Services für Datenbanken

AWS Managed Services, wie Amazon RDS für MySQL-Datenbanken und S3 für MinIO Blob Storage bieten effiziente Lösungen für die Verwaltung von Datenbanken und Speicherlösungen an. Amazon RDS stellt verwaltete MySQL-Datenbanken bereit und übernimmt automatisch Aufgaben wie Bereitstellung, Konfiguration, Backups und Patching, während die Skalierbarkeit flexibel angepasst werden kann. Die Integration von RDS und S3 mit Elastic Beanstalk und anderen AWS-Services ermöglicht einen effizienten Umgang mit den Daten innerhalb der Webanwendung. Mittels dieser Managed Services kann sämtliche Verantwortung für die Inbetriebnahme und die Wartung der Datenbanken an den Cloud Provider abgeben werden.

V. Architekturentscheidungen für Improvements

Interessante Entscheidungspunkte im Projekt waren im Weather Service und im Order Service. Diese „Design-Entscheidungen“ sollen folgend analysiert und ein Ausblick gegeben werden.

Weather-Service

Im Wetterservice wurde die Entscheidung getroffen, dass pro Call auf der Wetter Page, ein API-Call an den Wetterservice stattfindet. Argumente könnten hierbei sein, dass jeder Kunde mit seinem Klick Live Daten von der API bekommen soll. Die API in der Free-version bietet allerdings nur wenige freie Aufrufe pro Tag an, was diese Designentscheidung hinfällig für einen Deploy machen würde. Bei einer großen Nutzermasse wäre die Anzahl an unnötigen API-Calls enorm. Hierbei gibt es einige Verbesserungen, die man im weiteren Verlauf einer Implementierung vornehmen könnte.

Eine Möglichkeit wäre ein smartes cachen der Wetterdaten. Bei einem Aufruf der Wetter API könnten hierbei die Daten in einem Cache abgelegt werden und für den Service zugänglich sein. Dieser könnte mit Businesslogik prüfen, ob sich Daten im Cache befinden oder ob dieser einen erneuten Call zur Wetter API betätigen muss. Ebenso ist eine Kombination mit einem Zeit Intervall möglich, sodass der Service alle 10 Minuten die neusten Daten erhält und diese für die Kunden statisch bereitstellt.

Für eine exemplarischen Implementierung war dieses Konzept zu spezifisch, für ein realistischen Deploy für den Kunden, definitiv eine Überlegung wert.

Was *definitiv nicht passieren darf*, ist dass die Daten im Service selbst abgelegt werden. Dies würde den Service stateful machen und dem Konzept eines skalierbaren Services widersprechen.

Order-Service

Momentan wird die ID des Hotelzimmers und die persönlichen Einträge des Kunden aufgenommen und mittels des Oder-Services in eine DB abgelegt. An dieser Stelle finden Datenbank Einträge auf Basis der Kundeneingaben statt. Kritisch könnten hier Fälle einer SQL-Injektion sein oder ein Manipulieren der Eingabe der Hotelzimmerdaten. Da die Zimmer ID nur frontendseitig gesetzt wird und nicht im Backend nach „Verfügbar“, „Buchbar“ oder „Frei“ geprüft wird, könnten sich hier Kunden für Hotelzimmer bewerben, die eigentlich zur Verfügung stehen würden. Hier wären backendseitige Validierungsschritte ratsam oder ein Abgleich im Order-Service mit „geschützten“ Daten. Für einen exemplarischen Entwurf funktional, für einen realistischen Deploy definitiv zu unsicher.