Smart Lamp Control Dilmand Saado (MtrNr. 9511296), Tom Weber (MtrNr. 1705171), Felix Erhard (MtrNr. 3394927) - Architektur und Funktionsweise

Gewählte Architektur

Für unsere Smart Lamp Control-Anwendung haben wir eine vereinfachte Mikroservice-Architektur mit Message-Oriented Middleware gewählt. Die Anwendung besteht aus drei Hauptkomponenten:

1. **Consumer**: Steuert die physische Lampe und verarbeitet Befehle
2. **Frontend/UI**: Bietet die Benutzeroberfläche im Browser und kommuniziert direkt mit RabbitMQ
3. **RabbitMQ**: Fungiert als zentraler Message Broker für die gesamte Kommunikation

Diese Architektur zeichnet sich durch den Verzicht auf einen klassischen Backend-Producer aus. Stattdessen kommuniziert das Frontend direkt über WebSockets mit RabbitMQ.

Gründe für diese Architektur:

- **Maximale Entkopplung**: Frontend und Consumer sind vollständig voneinander entkoppelt und kommunizieren ausschließlich über Messages.
- **Reduzierte Komplexität**: Durch den Wegfall einer zusätzlichen Backend-Komponente wird die Architektur vereinfacht.
- **Zuverlässigkeit**: RabbitMQ speichert Nachrichten und stellt sie zu, auch wenn Komponenten temporär nicht verfügbar sind.
- **Unmittelbare Reaktionsfähigkeit**: Direkte WebSocket-Verbindung vom Browser zu RabbitMQ ermöglicht Echtzeit-Updates ohne Umwege.
- **Event-Driven Design**: Status-Updates werden als Events veröffentlicht, auf die Clients direkt reagieren können.
- **Containerisierung**: Die Docker-basierte Infrastruktur ermöglicht einfaches Deployment und Skalierung.

Kommunikationsmuster:

Wir verwenden zwei zentrale Kommunikationsmuster:

1. **Frontend → Consumer**: Befehle werden über die Queue "lamp-commands" an den Consumer (Lampe) gesendet.
2. **Consumer → Frontend**: Status-Updates werden über den Exchange "lamp-status" veröffentlicht und vom Frontend empfangen.

WebSocket-Integration mit RabbitMQ

Eine Besonderheit unserer Architektur ist die Integration von WebSockets direkt mit RabbitMQ durch das RabbitMQ Web STOMP Plugin. Dadurch wird eine eigene WebSocket-Implementierung im Producer überflüssig. Das Plugin bietet uns somit:

- WebSockets direkt mit dem RabbitMQ Message Broker zu verbinden

- STOMP-Protokoll-Support über WebSocket
- Eine direkte Verbindung zwischen Browser-Clients und der Message Queue

Diese Integration bietet mehrere Vorteile:

- **Reduzierter Code**: Der Producer muss nicht als WebSocket-Proxy fungieren
- **Skalierbarkeit**: Die WebSocket-Verbindungen werden direkt von RabbitMQ verwaltet
- **Standardisierung**: Nutzung des etablierten STOMP-Protokolls für Messaging
- **Erhöhte Zuverlässigkeit**: Direkter Zugriff auf die Robustheit von RabbitMQ

Retry-Mechanismus

Das Backend überprüft in regelmäßigen Abständen (30s), ob eine Verbindung zu einer Lampe besteht. Wenn ein Lampe nicht verfügbar sein sollte, wird ein MockDevice erstellt um den Status der Lampe über Events im UI darstellen zu können. Dabei wird der Retry-Mechanismus weiterhin durchgeführt, sodass sobald eine Lampe verfügbar ist, diese als Device verwendet werden kann. Für den Fall, dass die Verbindung erneut geschlossen wird, wird wieder auf das MockDevice gewechselt.

Funktionsweise der Anwendung

1. Systemstart

Bei Systemstart werden drei Docker-Container gestartet:

- RabbitMQ: Stellt den zentralen Message Broker bereit
- Consumer: Verbindet sich mit der physischen Lampe und RabbitMQ
- Webserver: Stellt lediglich die statischen Frontend-Dateien bereit

2. Benutzeraktion → Lampensteuerung

1. **Benutzerinteraktion**: Der Benutzer interagiert mit der Weboberfläche (z.B. Ein-/Ausschalten, Helligkeit ändern).
2. **WebSocket-Verbindung**: Das Frontend sendet Befehle direkt über WebSocket an RabbitMQ.
3. **Message Routing**: RabbitMQ leitet die Nachricht an die "lamp-commands" Queue weiter.
4. **Command-Verarbeitung**: Der Consumer empfängt den Befehl und verarbeitet ihn mittels des Strategy Patterns:
    - Für jeden Befehlstyp (on, off, brightness, color, morse) gibt es eine Strategy-Klasse.
    - Die entsprechende Strategie wird ausgewählt und ausgeführt.
5. **Lampensteuerung**: Der Consumer steuert die physische Lampe über die TP-Link API.

3. Lampenänderung → UI-Update

1. **Statusänderung**: Nach jeder Änderung des Lampenzustands wird ein Event generiert.
2. **Event-Veröffentlichung**: Der Consumer veröffentlicht den aktuellen Status über den RabbitMQ "lamp-status" Exchange.
3. **WebSocket-Empfang**: Das Frontend empfängt das Status-Event direkt von RabbitMQ über die WebSocket-Verbindung.
4. **UI-Aktualisierung**: Die Weboberfläche aktualisiert die Darstellung der Lampe und den angezeigten Status.

4. Spezialfunktion: Morse-Code

Eine besondere Funktion unserer Anwendung ist die Morse-Code-Übertragung:

1. Der Benutzer gibt einen Text in das Morse-Eingabefeld ein und sendet ihn ab.
2. Das Frontend sendet einen `morse`-Befehl mit dem Text als Wert an RabbitMQ.
3. Der Consumer wandelt den Text in Morse-Code um (Punkte und Striche).
4. Die Lampe blinkt entsprechend dem Morse-Code:
    - Kurzes Blinken für Punkte (.)
    - Längeres Blinken für Striche (-)
    - Pausen zwischen Zeichen und Wörtern
5. Während der Übertragung werden Fortschritts-Updates über den "lamp-status" Exchange veröffentlicht.
6. Die UI zeigt den Fortschritt der Morse-Code-Übertragung an.

Vorteile unserer Implementierung

1. **Einfachheit**: Durch den Verzicht auf einen separaten Producer-Service ist die Architektur schlanker und einfacher zu verstehen.
2. **Echtzeitfähigkeit**: Direkte WebSocket-Verbindung zwischen Frontend und Message Broker für unmittelbare Updates.
3. **Erweiterbarkeit**: Neue Befehlstypen können einfach durch Hinzufügen neuer Strategien im Consumer implementiert werden.
4. **Robustheit**: Fehler in einer Komponente beeinträchtigen nicht das gesamte System.
5. **Multiuser-Unterstützung**: Mehrere Benutzer können gleichzeitig die Lampe steuern, da alle Clients die gleichen Status-Updates erhalten.
6. **Geringer Ressourcenverbrauch**: Weniger Services bedeuten weniger Overhead und einfachere Wartung.