Notizen zum Vorgehen Bachelorarbeit

# Backend

## WebAPI – WebServer – Zertifikat

1. Express Libary verwendet, um ein HTTP Endpoint lokal zu erstellen
2. EC2 Server (Amazon AWS) erstellt, um den Express Endpoint öffentlich zu machen
3. Server RSA Key (openssl) für HTTPS von Amazon AWS generieren lassen und lokal gespeichert
4. Aus dem RSA Key und einer Zertifikatsgenerierungsanforderung (Certificate Signing Request CSR Datei) mithilfe des GIT Repository Openssl eine Zertifikatsdatei erstellt.
5. Auf dem EC2 Server installieren ein selbst signiertes Zertifikat erstellt, damit eine sichere HTTPS Verbindung zunächst ermöglicht wird (Signierung einer öffentlichen Signierungsstelle steht noch aus bzw. ist derzeit nicht möglich).
6. ~~Elastische IP-Adresse dem EC2 Server zugewiesen.~~  🡸 Zurückgenomen aufgrund hoher Kosten.

## Setup EC2 Server Architektur

**Wie kann Express auf dem EC2 Server betrieben werden, sodass HTTPS Anfragen bearbeitet werden können?**

Zunächst laufen Node.js-Anwendungen im Allgemeinen nicht direkt auf Port 80 oder 443, die für HTTP- bzw. HTTPS-Verkehr reserviert sind, weil Ports unter 1024 als privilegierte Ports gelten und normalerweise nur von Prozessen mit Root-Zugriff verwendet werden können. Es ist aus Sicherheitsgründen nicht ratsam, Anwendungen mit Root-Berechtigungen auszuführen. In solchen Architekturen wird normalerweise die Node.js-Anwendung auf einem höheren, nicht privilegierten Port zum Laufen gebracht - in diesem Fall Port 3000. Theoretisch ist es möglich, dass die Anwendung so konfiguriert ist, dass direkt HTTPS-Anfragen auf dem Port 80 oder 443 angenommen werden, aber dann müsste für jede Anfrage SSL/TLS-Entschlüsselung in Node.js durchgeführt werden, was sehr ineffizient ist. Stattdessen wird oft ein Reverse-Proxy wie „Nginx“ verwendet. Nginx ist darauf ausgelegt, effizient eine große Anzahl von Verbindungen zu verwalten und kann daher besser mit dem HTTPS-Verkehr umgehen. In diesem Fall wird Nginx so konfiguriert, dass es auf Port 443 für eingehenden HTTPS-Verkehr lauscht. Es führt dann die SSL/TLS-Entschlüsselung durch und leitet die entschlüsselte Anfrage an die Node.js-Anwendung weiter, die auf Port 3000 lauscht. Diese Architektur hat mehrere Vorteile. Sie ermöglicht es die Anwendung ohne höhere Rechte laufen zu lassen, was sicherer ist, und entlastet die Anwendung von der Aufgabe, den SSL/TLS-Verkehr zu verarbeiten. Es ermöglicht auch andere Funktionen, wie das Lastverteilung von Anfragen auf mehrere Node.js-Instanzen, wenn größere Anwendungen betrieben werden.

## SSL-Certificate: Self-Signed

Hier muss eine Autorisierungsstelle ein Zertifikat ausstellen. Selbst signierte Zertifikate werden von vielen Anwendungen nicht akzeptiert oder als unsicher betrachtet. Certbot ist ein Python Packet mit dem es möglich ist über die Zertifizierungsstelle „Let‘s Ecrypt!“ ein Zertifikat zu generieren. Da der EC2 Server jedoch ein Amazon AWS Server ist, dessen Domain sich ändert, stellen bekannte Zertifizierungsstellen ein Zertifikat für AWS-Server aus. Aus diesem Grund wird zunächst ein selbst signiertes Zertifikat verwendet. Um das Problem zu umgehen (auch für AWS-Server) könnte in Zukunft eine Domain gemietet werden, welche auf die IP-Adresse des Amazon Servers mithilfe eines DNS-Eintrags verweist. Diese Domain kann dann ein Zertifikat erhalten, da sich diese nicht durchgehend ändert und somit „vertrauenswürdig“ ist.

## HTTPS in Node/Express-Anwendung

Dies kann sinnvoll sein, wenn die Anwendung Daten verarbeitet, welche Verschlüsselt übertragen werden sollen. Da aber der Reverse Proxy Nginx bereits HTTPS Anfragen verarbeitet, ist zusätzliche Rechenleistung für Ver- und Entschlüsselung nicht unbedingt notwendig. Normalerweise verwenden Node-Anwendungen ausschließlich HTTP.

## 1 Entwicklung Express.js REST-API mit CRUD-Operationen

Bei der Umsetzung der REST-API wurde zunächst wie folgt vorgegangen:

1. Proof-of-concept:
   1. Ein Endpoint, welcher „Hello World!“ ausgibt wurde mithilfe einer Express.js-Anwendung entwickelt und auf der Server Architektur hochgeladen. Nachdem der Endpunkt mithilfe von PostMan im Netz verfügbar war, konnte mit der Entwicklung einer richtigen REST-API Anwendung begonnen werden.
   2. Datenbankverbindung: Es wurde eine Verbindung an die MongoDB Datenbank im Backend entwickelt und getestet.
   3. Schemata: Es wurden verschiedene Schemata für die Erstellung von Dokumenten auf der Datenbank erzeugt.
2. Konzeption: Es wurde ein Konzept für die Express.js-Anwendung ausgearbeitet
3. Implementierung: Die in dem Konzept entwickelte Struktur der Express.js-Anwendung wurde implementiert.
   1. Bei der Implementierung wurden zunächst CRUD-Operationen (Create, Read, Update, Delete) für HTTP-Requests erstellt, um die Datenbank mit Testdaten zu befüllen und ein Standard Toolset für die weitere Entwicklung im Frontend Bereich zu haben. Des Weiteren konnte dadurch PostMan effizienter im gesamten Backend Bereich genutzt werden.
4. Testen: Die entwickelten Endpunkte wurden alle mit PostMan getestet (vergleichbar mit Unit tests).

## 2 Express.js: Security der REST-API und des Backends

Vorgehen:

Express.js REST API mit MongoDB:

1. Verwendung von HTTPS für alle API-Anfragen.
   * HTTPS wird mithilfe des reverse proxy Servers (nginx) Sichergestellt.
2. Passwörter werden mit bcrypt gehasht, bevor sie gespeichert werden.
   * bcrypt ist ein Node.js Modul, welches Hashing-Funktionen bereitstellt.
   * bcrypt erzwingt von Haus aus „salt“ Runden, um die Sicherheit von Hashes zu erhöhen. Es werden zunächst 10 salt Runden implementiert.
   * bcrypt verwendet den Blowfish cypher als algorithmus.
3. JWT-Token mit einem Ablaufdatum für die Authentifizierung.
   * JWT-Token werden beim Login für jeden User mithilfe eines privaten Schlüssels generiert und an den anfragenden zurückgeschickt.
   * Bei jeder Anfrage wird der JWT-Token auf Gültigkeit geprüft, bevor der payload an die Ziel-Route weitergeleitet wird.
4. Validierung und Desinfektion aller Benutzereingaben.
5. CORS-Middleware, um nur spezifische Ursprünge zuzulassen.

Quellen:  
(für 2.) <https://heynode.com/blog/2020-04/salt-and-hash-passwords-bcrypt/>

# Frontend

## Lazy loading / Caching für Bilder während des Ladeprozesses

Beschreibung wie dies umsetzbar wäre und möglicherweise auch konkrete Beispiele.