UE4 – JWT, Websockets, HTTPS, Deployment (25 Punkte)

Ziel dieses Übungsbeispiels ist es, weitere Technologien kennenzulernen, um die Kommunikation zwischen Client und Server zu verbessern. Dabei sollen für die vorhandenen Schnittstellen eine vorherige Authentifizierung mittels JSON Web Token (JWT) implementiert und die Kommunikation mit HTTPS verschlüsselt werden. Außerdem soll ein WebSocket verwendet werden, um vom Server kontinuierlich Daten senden zu können.

Deadline der Abgabe via TUWEL: **Sonntag, 03.06.2018 23:55 Uhr**

Nur ein Gruppenmitglied muss die Lösung abgeben!

Angabe

```
UE4-Angabe.zip
    client
        src
            app
                ...                 (von Ihnen teilweise zu erweiternde Dateien für
                app.module.ts       die Angular-App – siehe Übung 3)
                app-routing.module.ts
            environments            (Environment-Konfigurationen mit Server-URLs)
            ...                     (mehrere Ressourcen-Verzeichnisse)
            ...                     (mehrere weitere Dateien)
        .gitignore                  (u.a. wegen node_modules)
        README.txt                  (Informationen zu Angular CLI)
        ...                         (mehrere weitere Dateien)
    server
        resources                   (Server-Ressourcen)
        .gitignore                  (u.a. wegen node_modules)
        server.js                   (von Ihnen zu ergänzende Server-Applikation)
        simulation.js               (Simulation für Smart Production)
        ...                         (mehrere weitere Dateien)
    screenshots                     (possible production chains)
```

Bitte beachten Sie, dass nicht alle der bereitgestellten Dateien bearbeitet werden sollen (siehe *„Abgabemodalität“*). Nehmen Sie jedoch keinesfalls Änderungen an den verwendeten Bibliotheken vor. Alle für diese Aufgabe benötigten Bibliotheken sind bereits enthalten und entsprechend eingebunden und bedürfen daher keines Einschreitens Ihrerseits.

Im Ordner „screenshots“ finden Sie drei mögliche Kombinationen für eine Produktionskette (mehr Kombinationen sind möglich).

JSON Web Token

Die bereits vorhandenen Schnittstellen sollen nun auf authentifizierte User beschränkt werden. Dafür soll bei einer erfolgreichen Authentifizierung am Server ein Token generiert und zurückgegeben werden, der in weiterer Folge bei allen Requests vom Client mitgeschickt wird.

Die einfachste Form von Token ist eine zufällige Zeichenfolge, die dann serverseitig in einer Liste gespeichert wird und mittels dieser Liste validiert werden kann. Ein Beispiel für solche Tokens sind Session-IDs. Diese Methode hat jedoch den Nachteil, dass das System damit nicht mehr *stateless* ist – ob ein Token gültig ist hängt von der am Server vorhandenen Liste ab.

Für stateless Systeme sollte daher die Gültigkeit bereits vom Token selbst ableitbar sein. Ein solcher Token besteht aus (zumindest) zwei Teilen: Die eigentlichen Daten (oft als *body* oder *payload* bezeichnet) sind um eine *kryptografische Signatur* zur Sicherstellung der Integrität der Daten ergänzt. Ein (offener) Standard für signierte Token ist *JSON Web Token (JWT)*[1], die in dieser Übung verwendet werden.

Folgende Funktionalität soll implementiert werden:

- Nach erfolgreicher Authentifizierung soll der Server über die im Angabe-Template inkludierte JWT-Bibliothek[2] einen neuen Token generieren. Im Token soll zumindest der Username enthalten sein, außerdem soll der Token nur zeitlich begrenzt gültig sein.
- Im Client soll der Token in `LocalStorage` abgelegt werden. Die bereits von Übung 3 existierenden Services können dafür (mit entsprechenden Modifikationen) weiterverwendet werden.
- Beim Laden der Angular-App soll überprüft werden, ob ein eventuell in `LocalStorage` vorhandener Token (zeitlich) noch gültig ist. Dafür sollen keine JWT-Bibliotheken verwendet werden, sondern der Ablaufzeitpunkt entsprechend der JWT-Spezifikation direkt ausgelesen werden. Die Signatur muss im Client nicht validiert werden.
- Bei jedem Request soll der im `LocalStorage` vorhandene Token im Authorization-Header[3] mitgesendet werden. Verwenden Sie den Authorization-Typ *Bearer*. Implementieren Sie einen `HttpInterceptor`[4], um den Header transparent für alle Requests hinzufügen zu können.
- Der Server soll bei jedem Request den Token im Header überprüfen. Verwenden Sie dafür wie zum Signieren die inkludierte Bibliothek. Bei ungültigen Tokens soll mit einem passenden Statuscode abgebrochen werden.

Anmerkung: Üblicherweise werden für die Gültigkeitsdauer von JWT kleine Werte (z.B. 10 Minuten) gewählt, weil dann die Notwendigkeit einer serverseitigen *Revocation List* entfällt. Im Client sollte dann sichergestellt werden, dass im Hintergrund rechtzeitig ein neuer Token angefordert wird. Im Rahmen dieser Übung muss das jedoch nicht implementiert werden – wenn Tokens abgelaufen sind ist einfach ein neuerlicher Login nötig.

---

1 https://jwt.io/introduction

2 https://www.npmjs.com/package/jsonwebtoken

3 https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization

4 https://angular.io/api/common/http/HttpInterceptor

WebSockets

Die REST-Schnittstelle wurde im Angabe-Template so erweitert, dass nun Geräte und ihre Beziehungen untereinander ebenfalls auf den Server übertragen werden. Sie müssen daran keine Anpassungen vornehmen, sollten allerdings für das Abgabegespräch die Struktur der neuen REST-Ressourcen verstanden haben. Beachten Sie, dass grundsätzlich vor der Speicherung unbedingt eine serverseitige Validierung der Daten erfolgen muss. Zur Vereinfachung verzichten wir hier jedoch (großteils) darauf.

Ihre Aufgabe ist es nun, am Server einen *WebSocket* mittels *express-ws*[5] bereitzustellen sowie darüber Nachrichten mit dem Client auszutauschen:

- Updates in den Detail-Seiten der Geräte sollen per WebSocket an den Server übertragen werden. Serverseitig soll entsprechende der Wert im Objekt `devices` aktualisiert werden.
- Der Client soll, nachdem der WebSocket verbunden wurde, eine Subscribe-Nachricht senden, sodass in weiterer Folge Updates an diesen Client gesendet werden.
- Am Server läuft eine (einfache) Simulation auf Basis der hinzugefügten Geräte. Immer wenn die Simulation einen Wert verändert, wird die Methode `sendUpdatedValue` (mit dem Geräte-Index und dem neuen Wert als Parameter) aufgerufen. Diese Methode soll Updates an alle WebSocket-Clients schicken, die zuvor eine Subscribe-Nachricht gesendet haben. Der Client soll dann das Bild und Diagramm zum jeweiligen Gerät aktualisieren.

  Nicht alle Produktionsketten sind gültig – beachten Sie die Konsolen-Nachrichten des Servers!
- Der Counter *Produkte* soll die Summe der Produkte in allen Endlagern anzeigen.

Anmerkungen:

- Auf Nachrichten vom Client soll der Server mit einer Bestätigung oder Fehlermeldung antworten. Nachrichten vom Server an den Client werden nicht quittiert.
- Alle oben aufgeführten Nachrichten sollen über einen einzigen WebSocket übermittelt werden. Überlegen Sie sich daher ein (einfaches) auf JSON basierendes Protokoll, um die verschiedenen Typen zu unterscheiden und die Nutzdaten auslesen zu können.
- Nachrichten vom Client an den Server müssen mittels JWT authentifiziert werden. Beim Senden der Updates vom Server muss jedoch nicht überprüft werden, ob der Token zum jeweiligen Client noch gültig ist.
- Der WebSocket soll über eine gesicherte Verbindung erstellt werden (siehe nächster Abschnitt).
- Es wäre zwar sinnvoll, wenn Clients auch per WebSocket darüber informiert würden, wenn Geräte hinzugefügt etc. werden, allerdings werden wir das hier nicht implementieren, um den Aufwand niedrig zu halten.

5 https://www.npmjs.com/package/express-ws

HTTPS

Bisher wurden sämtliche Daten zwischen Browser und Server unverschlüsselt übertragen, wodurch die Daten (sobald der Server nicht mehr lokal am selben Rechner läuft) relativ einfach mitgelesen und manipuliert werden können. In weiterer Folge sollen die Daten daher mittels *Transport Layer Security (TLS)*[6] verschlüsselt werden. Dafür muss ein Zertifikat erstellt werden, das den Server gegenüber des Browsers identifiziert.

Für öffentlich zugängliche Websites müssen solche Zertifikate von einer *Certificate Authority (CA)*[7] signiert werden, damit die Gültigkeit überprüft werden kann. Für Intranet-Applikationen und Testserver ist das jedoch oft nicht zweckmäßig oder auch gar nicht möglich. Hier können *selbstsignierte Zertifikate* verwendet werden, für die im Browser zunächst manuell eine Ausnahme hinzugefügt werden muss.

Sie sollen daher nun drei Dateien für den Server erstellen:

- *Private Key*: Für reale Anwendungen sollten Private Keys klarerweise niemals weitergegeben werden. Im Zuge dieser Übung können Sie den Key jedoch dennoch im Repository mit den anderen Gruppenmitgliedern teilen. Inkludieren Sie den Key außerdem in der TUWEL-Abgabe.
- *Certificate Signing Request (CSR)*: Stellen Sie sicher, dass als *Subject* sinnvolle Werte angegeben werden!
- Zertifikat: Dafür soll das CSR-File mit dem zuvor erstellten Private Key signiert werden (selbstsigniert). Das Zertifikat soll für die Domain `localhost` gültig sein. Wählen Sie die Gültigkeitsdauer so, dass eine irrtümliche Verwendung über die Übung hinaus ausgeschlossen ist.
- Ändern Sie den Express-Server so ab, dass ein HTTPS-Server mit dem erstellten Zertifikat gestartet wird.

Sie können für die Erstellung von Key, CSR und Zertifikat beispielsweise *OpenSSL*[8,9] verwenden, das unter Linux üblicherweise vorinstalliert, aber auch für andere Betriebssysteme verfügbar ist.

Anmerkungen:

- Früher wurde die Domain meist im *Common Name* des Zertifikats angegeben, die Erweiterung *Subject Alternative Names (SAN)* war nur notwendig, wenn das Zertifikat für mehrere Domains gleichzeitig gelten sollte. Mittlerweile wird der Common Name jedoch ignoriert, die Subject Alternative Names sind verpflichtend. Folgen Sie daher in älteren Anleitungen den Schritten für mehrere Domains, obwohl das Zertifikat nur für `localhost` gelten soll.
- In Anleitungen kursieren oft weiterhin veraltete *Hashing-Algorithmen* und *Schlüssellängen*. Jedenfalls nicht mehr verwendet werden sollten die Algorithmen MD5 und SHA1 sowie Schlüssel-

---

6 Der Vorgänger *Secure Sockets Layer (SSL)* ist überholt, in der Praxis wird *SSL* jedoch oft als Synonym für *TLS* weiterhin verwendet.

7 Beispielsweise können über https://letsencrypt.org/ kostenlose Zertifikate bezogen werden.

8 https://fabianlee.org/2018/02/17/ubuntu-creating-a-self-signed-san-certificate-using-openssl/ deckt die für die Übung nötigen Schritte sehr gut ab.

9 https://gist.github.com/Soarez/9688998 bietet eine ausführlichere Erklärung und Anleitung.

längen unter 2048 Bit – solche Zertifikate werden von aktuellen Browsern üblicherweise gar nicht akzeptiert. Aktuelle OpenSSL-Installationen verwenden passende Default-Werte.

- Sie müssen die URL des Servers einmal direkt im Browser aufrufen, um das Zertifikat als vertrauenswürdig zu markieren. Andernfalls wird Angular nicht mehr mit dem Server kommunizieren können.

Zusatzaufgabe (3 Bonuspunkte):

Bei größeren Projekten ist es sinnvoller, eine eigene (z.B. firmen- oder teaminterne) CA einzurichten und diese im Browser als vertrauenswürdig zu importieren, sodass nicht für alle einzelnen Server Ausnahmen hinzugefügt werden müssen. Führen Sie daher stattdessen folgende Schritte durch:

- Erstellen Sie einen Private Key und ein Zertifikat für eine (selbstsignierte) Certificate Authority.
- Erstellen Sie einen Private Key und ein CSR für den REST-Server (wie oben).
- Erstellen Sie einen weiteren Private Key und CSR für den Angular-Server. Verwenden Sie dafür ebenfalls `localhost` als Domain.
- Signieren Sie die beiden Requests mit der zuvor erstellten CA.
- Ändern Sie die Konfiguration des Servers entsprechend (wie oben).
- Passen Sie die Konfiguration von Angular CLI in `.angular-cli.json` so an, dass standardmäßig ein HTTPS-Server gestartet wird.
- Dokumentieren Sie die durchgeführten Schritte in einer Textdatei kurz, sodass die anderen Gruppenmitglieder die Schritte nachvollziehen können.

Die obigen Anmerkungen gelten hier gleichermaßen. Setzen Sie insbesondere eine kurze Gültigkeitsdauer für das CA-Zertifikat.

Deployment und Optimierung

Der integrierte Angular-Server ist zwar während der Entwicklung praktisch, für das tatsächliche Deployment[10] der Applikation jedoch ungeeignet. Daher ist zunächst ein Build der Applikation erforderlich, der dann auf einem Server bereitgestellt wird. Führen Sie daher folgende Schritte durch:

- Generieren Sie einen *Development-Build*. Ein solcher Build hat den Vorteil, dass der Optimierungsschritt wegfällt und leichteres Debugging im Browser möglich ist, allerdings sind die Dateien erheblich größer. Benennen Sie das erzeugte Verzeichnis in `dist-dev` um, damit es im nächsten Schritt nicht überschrieben wird.
- Generieren Sie einen *Production-Build*. Ein solcher Build dauert etwas länger, dafür sind die Dateien erheblich kleiner, was für Web-Applikationen einen wichtigen Faktor darstellt. Vergleichen Sie die Gesamtgrößen der Development- und Production-Applikation!
- Die so gebaute Applikation benötigt zwar keine Node-Umgebung mehr, ein direkter Aufruf über das Filesystem ist jedoch trotzdem nicht möglich (aufgrund diverser Sicherheitsfeatures im Browser). Verwenden Sie daher einen (lokalen) Webserver, um die Applikation zu testen.
- Mit der Standard-Konfiguration üblicher Webserver sollte es direkt möglich sein, die Startseite aufzurufen und dann innerhalb der Applikation zu navigieren. Direktlinks auf z.B. `/overview` (oder ein Browser-Refresh auf Unterseiten) sind jedoch nicht möglich, weil diese Pfade serverseitig nicht existieren.

  Erstellen Sie daher ein Konfigurationsfile (wahlweise für *Apache* oder *nginx*), sodass für alle derartigen Requests `index.html` ausgeliefert wird und die Darstellung der richtigen Komponente wieder dem Angular-Router überlassen wird.

10 https://angular.io/guide/deployment

Hinweise

Ausführen der Applikation

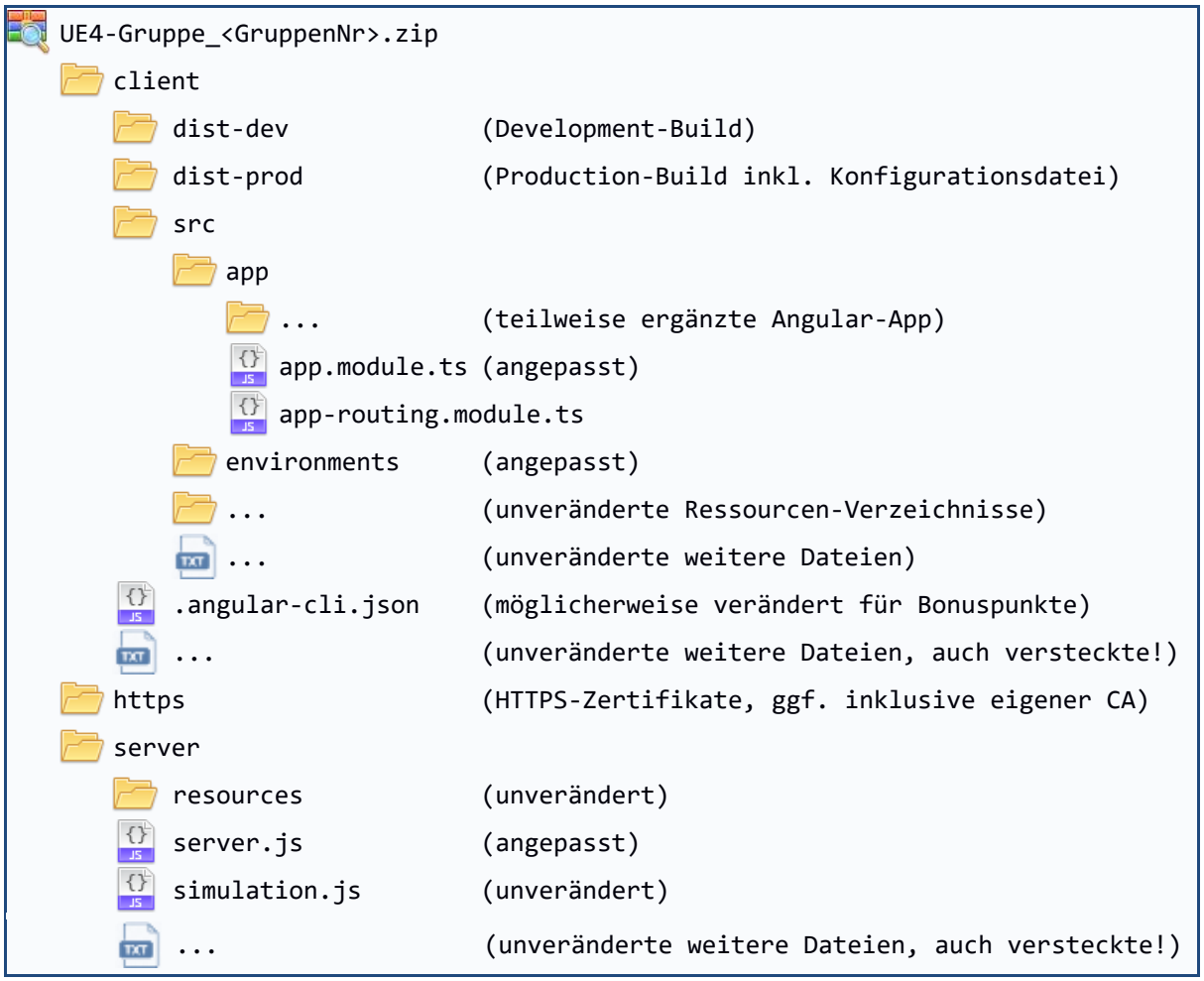
Sowohl Client- als auch Server-Applikation müssen mittels npm start ausgeführt werden können. Verändern Sie die mitgelieferten Bibliotheken nicht (auch nicht deren Versionen).

Validierung

Die Validität des von Angular generierten Codes muss nicht überprüft werden. Achten Sie aber darauf, möglichst validen Code zu erstellen und behalten Sie die Accessibility-Features der letzten Übung bei.

Abgabemodalität

Beachten Sie die allgemeinen Abgabemodalitäten des TUWEL-Kurses. Zippen Sie Ihre Abgabe, sodass sie die folgende Struktur aufweist:



- Alle Dateien müssen UTF-8 codiert sein!
- Nur ein Gruppenmitglied muss die Lösung abgeben!
- Geben Sie keinesfalls `node_modules` ab!
- Wird das Abgabeschema bzw. der Name der Zip-Datei (`UE4-Gruppe_<GruppenNr>.zip`) nicht eingehalten, kommt es zu Punkteabzügen!