Hochschule Konstanz
Fakultät Elektrotechnik
und Informationstechnik

REST API, POST/PUT

Verteilte Systeme

Autoren:

Tobias Marzini	Ma. Nr. 298070
Sebastian Albrecht	Ma. Nr. 298242
Sascha Timm	Ma. Nr. 297782

15.07.2021

Inhaltsverzeichnis

Abbildungsverzeichnis

Einleitung

Die Aufgabe des Projektes belief sich auf die Implementierung einer RESTful API auf dem Microcontroller TexasInstruments TM4C129EXL. Dabei sollte auf eine Beispielanwendung via Postman und einem Python-Skript zugegriffen werden. Darüber hinaus war die Kommunikation zwischen Client und Server mit TLS zu verschlüsseln. Ebenfalls musste die Beispielanwendung mehrere Ressourcen umfassen, die über POST- und PUT-Anfragen angesprochen werden können. Diese Dokumentation befasst sich zuerst mit der Theorie zu den eingesetzten Technologien. Im Anschluss folgt ein Teil, der sich mit der Recherche befasst. Daraufhin wird der Laborversuch beschrieben und zum Schluss werden die wichtigen Codeteile nochmals erklärt.

Lightweight IP

lwIP steht für lightweight IP. IP bedeutet dabei Internetprotokoll, d.h. lwIp ist eine unabhängige Implementierung der TCP/IP-Protokolle.[1] Es wird in eingebetteten Systemen eingesetzt und steht Open-Source zur Verfügung. Lightweight kommt daher, dass der Programmcode nur 40KB ROM befasst. Es wurde entwickelt, um den Ressourcenverbrauch zu reduzieren und trotzdem sicherzustellen, dass der ganze TCP-Protokollstack zur Verfügung steht. Deswegen wird es in diesem Projekt eingesetzt und als Basis herangezogen. Auf der Vermittlungsschicht wird das Internet Control Message Protocol (ICMP) für Netzwerkwartungen und das Internet Group Management Protocol (IGMP) für das Management von Multicast benutzt. Allgemein wird das ICMP zur Übertragung von Statusinformationen und Fehlermeldungen in IP-Netzwerken genutzt. Daher ist es auch für Netzwerkwartungen geeignet. Es schickt Fehler- oder Informationsmeldungen wie z.B. Berichte über das Problem an die Quelle zurück. IGMP ist, wie der Name schon sagt, ein Hilfsprotokoll, das zur Gruppenkommunikation benutzt wird und daher eben auch vor allem für den Multicast in IP-Netzwerken. In der Transportschicht steht das vebindungsorientierte TCP-Protokoll und das verbindungslose UDP-Protokoll zur Verfügung. Für die Anwendungsschicht gibt es das Domain Name System (DNS), das Simple Network

[1] Doxygen: „lwIP 2.1.0“, in: “https://www.nongnu.org/lwip/2_1_x/index.html”, 01.08.2013, Zugriff: 03.04.2021

Management Protocol (SNMP) und das Dynamic Host Configuration Protocol (DHCP), das in diesem Projekt zum Einsatz kommt.

Sockets

Ein Socket ist ein vom Betriebssystem bereitgestelltes Objekt.[2] Um eine Kommunikation zu ermöglichen, wird er erstellt und dient als Verbindungspunkt in einem TCP/IP-Netzwerk. Ein Socket wird durch eine IP-Adresse vom Rechner oder System und einer Port-Nummer definiert. Es gibt verschiedene Services auf einem Rechner, daher gibt es auch verschiedene Ports. Ein Port wird durch eine 16-Bit Zahl repräsentiert. Beispielsweise gibt es System-Ports von null bis 1023. Die Sieben steht dabei für echo und die 13 für daytime. Die weiteren Ports von 1024 bis 65535 sind User-Ports und daher User-spezifisch. Der Klient startet die Kommunikation durch Anlegen einer Socket-Verbindung. Bzw. das Programm fordert einen Socket vom Betriebssystem an. Es hat die Aufgabe, alle benutzten Sockets und zugehörigen Verbindungsinformationen zu verwalten. Der Server wartet auf die Anfrage an seinem Port. Dadurch kann ein Datenaustausch mit anderen Programmen erfolgen. Dabei kann das Programm auf demselben Rechner sein, dann wird es Interprozesskommunikation genannt oder auf einem anderen übers Netzwerk erreichbaren Rechner befinden. Daher kann die Socket-Kommunikation bei verteilten Systemen eingesetzt werden. Die Sockets bilden eine plattformunabhängige und standardisierte Schnittstelle zwischen Netzwerkprotokoll-Implementierungen des Betriebssystems und der Anwendungssoftware.

REST

REST ist die Abkürzung für Representational State Transfer und wurde im Jahr 2000 von Roy Filding in seiner Dissertation vorgestellt.[3][4] Es ist ein Paradigma der Softwarearchitektur von verteilten Systemen und legt sechs Einschränkungen fest, die auch als Constraints bekannt sind. Sie schränken den Webservice dabei lediglich ein.

[2] Birkhölzer, Thomas: „Software Engineering Version 4.2“, in: “ https://moodle.htwg-konstanz.de/moodle/pluginfile.php/300185/mod_resource/content/1/se_skript.pdf”, 01.03.2021, Zugriff: 04.04.2021, S.77f.
[3] Böck, Boris: „Distributed Systems – 05 REST“, in: “https://moodle.htwg-konstanz.de/moodle/pluginfile.php/290964/mod_resource/content/3/VS_05_REST_en.pdf”, (o.D.), Zugriff: 05.05.2021
[4] Fielding, Roy: “CHAPTER 5 Representational State Transfer (REST)”, in: “https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm”, 2000, Zugriff: 05.05.2021

Sie beschreiben nicht, wie im Detail ein RESTkonformer Service zu implementieren ist. Ein REST Webservice wird als RESTful bezeichnet, wenn alle sechs Einschränkungen eingehalten werden. Das entspricht dem Wunsch, dass es einheitliche Schnittstellen geben soll. Das Protokoll dient dabei der Maschinen-zu-Maschinen-Kommunikation. Es werden standardisierte Verfahren verwendet. Beispielsweise sind sie HTTP, HTTPS, JSON und XML. Eine Ressource wird durch eine URI angegeben, der ihr Ort und Name angibt, nicht jedoch die Funktionalitäten, die auf ihr zur Verfügung stehen durch dessen Schnittstellen. Sie kann dadurch durch verschiedene Medientypen dargestellt werden. Man spricht in dem Zusammenhang auch von der Repräsentation der Ressource. REST Services erlauben dem anfragenden System auf sie zuzugreifen und zu manipulieren.

REST-API

Eine REST-API ermöglicht den Austausch von Informationen, wenn diese auf unterschiedlichen Systemen verteilt sind.[5] Sie ist eine Schnittstelle, die das REST-Protokoll nutzt und sich dabei an den Paradigmen und Verhalten des World Wide Webs orientiert. Sie nutzt einheitliche, vordefinierte und zustandslose Operationen, die hauptsächlich HTTP-Anfragen sind. API steht dabei für Application Programming Interface, d.h. sie ist eine Programmierschnittstelle. REST stellt dabei eine Alternative zu SOAP, WSDL und RPC dar. Durch sie kann eine Ressource abgefragt, neu angelegt, gelöscht oder geupdatet werden. Der abstrakte Verlauf einer Anfrage verläuft, wie in Abbildung eins dargestellt.

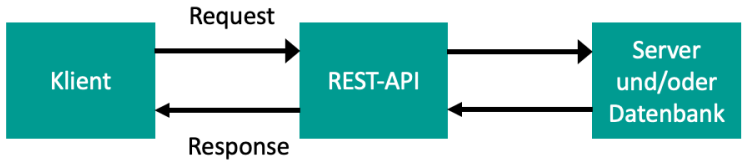


Abbildung 1: Ablauf einer REST-API-Anfrage[6]

[5] Dazer, Michael: „RESTful APIs- Eine Übersicht“, in: „https://www.snet.tu-berlin.de/fileadmin/fg220/courses/WS1112/snet-project/restful-apis_dazer.pdf“, (o.D.), Zugriff: 07.05.2021

[6] Screen shot Microsoft PowerPoint 16.0.13901.20366

Der Klient stellt eine Anfrage zur Ressource und schickt sie an die REST-API. Dazu stellt sie auf der Ressource verschiedene Methoden zur Verfügung. Der Methodenkatalog besteht aus GET, POST, PUT, DELETE, HEAD, PATCH und OPTIONS. Dabei werden ein Header und ein Body mitgegeben. Darin stehen zum Beispiel Anfrageparameter und Schlüssel zur Autorisierung. Die Schnittstelle reagiert auf die eingehende Anfrage, wenn die Autorisierung erfolgreich war, falls eine benötigt wurde. Sie wird verarbeitet und weitergegeben. So wird beispielsweise durch eine GET-Anfrage die angegebene Ressource in der Datenbank gesucht und zurückgegeben, sodass die REST-API sie in der Antwort an den Klienten schicken kann. Es gibt verschiedene Antworten. Sie gehen über informationelle oder erfolgreiche Antworten hin zu Fehlernachrichten mit Auslöser Klient oder der Server selbst. Da REST hauptsächlich HTTP-Anfragen verwendet, werden sie in fünf Kategorien eingeteilt, die durch HTTP-Statuscodes repräsentiert werden. Die Anfrage ist vollständig abgeschlossen, wenn der Klient dessen Antwort erhalten hat.

Constraints

Es gibt sechs Constraints, die auch als Architekturprinzipien bezeichnet werden. Sie schränken nur ein und geben nicht an, wie ein konformer Service zu implementieren ist. Um RESTkonform zu sein, muss der Service alle sechs Eigenschaften erfüllen. Es gibt das Klient-Server-Modell, die Zustandslosigkeit, einheitliche Schnittstellen, Cache, mehrschichtige Systeme und Code-on-Demand.

Klient-Server-Modell

Der Server stellt einen abfragbaren Dienst über dessen API zur Verfügung. Das Modell legt überhaupt die grundsätzliche Trennung in Klienten und Server fest (siehe Abb. zwei), um in einem Netzwerk Aufgaben und Dienstleistungen zu verteilen.

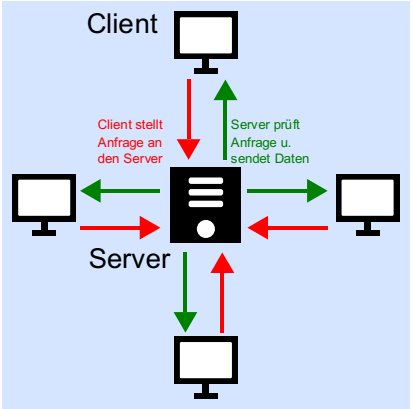


Abbildung 2: Client-Server-Modell[7]

So kann ein Klient einen Service bei dem Server anfragen. Der Server kann sich dabei auf dem gleichen oder auf einem anderen Rechner im Netzwerk befinden. Er beantwortet die Anfrage. Zudem kann ein Server für mehrere Klienten arbeiten.

Zustandslosigkeit

Die Kommunikation zwischen Klienten und Server ist zustandslos. Die Anfrage des Klienten enthält alle nötigen Informationen zur Beantwortung durch den Server, ohne dabei eine Abhängigkeit zu einer vorherigen Anfrage zu haben. Deswegen ist die Anfrage in sich geschlossen. Bei Wiederholung wird dieselbe Antwort vom Server erwartet. Keine Informationen zu den Klienten werden dabei zwischen den Anfragen gespeichert.

[7] Sarman, Marcel: "Das Client-Server-Modell erklärt", in: „https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.einfache-internetseiten.de%2Fdas-client-server-modell-erklaert%2F&psig=AOvVaw3FiVHvAt9WLeTxgGCeQya_&ust=1626360551995000&source=images&cd=vfe&ved=0CAoQjRxqFwoTCMDtv6rn4vECFQAAAAAdAAAAABAD", (o.D.), Zugriff: 05.05.2021

Einheitliche Schnittstellen

Das Ziel der einfachen Nutzung der Schnittstelle erfordert einheitliche Schnittstellen. Es ist das Hauptunterscheidungsmerkmal zu allen anderen APIs und besteht aus vier weiteren Eigenschaften. Sie sind die Adressierbarkeit von Ressourcen, die Repräsentation zur Veränderung von Ressourcen, die selbst-schreibenden Nachrichten und das Hypermedia as the Engine of Application State Merkmal.

Die Adressierbarkeit von Ressourcen:
Jede Information mit einer URI ist eine Ressource. Jeder RESTkonforme Server hat eine eindeutige Adresse für eine Ressource. Sie steht daher in einer URL. Dadurch wird eine konsistente Adressierbarkeit ermöglicht.

Die Repräsentation zur Veränderung von Ressourcen
Eine Ressource ist unter einer Adresse zugänglich. Die dadurch bereitgestellten Daten können in unterschiedlichen Darstellungsformen angesprochen bzw. abgefragt werden. Die Repräsentation betrifft das Format, die Sprache, Beschreibung oder Dokumentation des Dienstes. Die Daten können beispielsweise in ein JSON-, einem XML- oder HTML-File geschrieben und zurückgegeben werden. Die Veränderung einer Ressource sollte nur über ihre Repräsentation erfolgen!

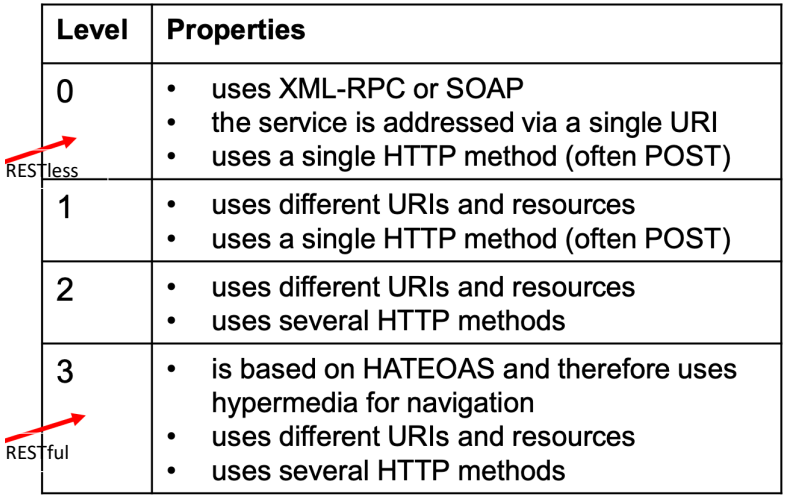
Die Selbst-schreibende Nachrichten
Jede Nachricht enthält ausreichend Informationen, um zu beschreiben, wie sie verarbeitet werden kann. Beispielsweise wird mitangegeben, in welchem Format die enthaltenen Daten stehen. Das Merkmal geht aus der Zustandslosigkeit hervor, da es eine Voraussetzung ist, da sonst die Nachrichten nicht alle dafür nötigen Informationen enthalten. Es werden Standardmethoden wie die von HTTP zur Ressourcenmanipulation verwendet.

Hypermedia As The Engine Of Application State (HATEOAS)
Das Merkmal ist die wichtigste Eigenschaft von REST, jedoch wird es oft missachtet. Das Mitliefern von URIs bei der Repräsentation der Ressource ermöglicht eine dynamische Bearbeitung der Ressource. Die URLs können direkt aus der vorhergehenden Antwort übernommen werden. Durch REST-Anfragen erfolgt eine Zustandsänderung des Systems. Es verläuft wie folgt: Der erste Klient greift auf eine

anfängliche URL der REST-Anwendung zu. Der Klient sollte nun vom Server die möglichen Links bekommen, bei denen Anfragen möglich sind, dadurch kann er ebenfalls alle verfügbaren Ressourcen sehen und die auswählen, die er benötigt. Nach Anfrage der Ressource antwortet der Server mit Text, in dem unter anderem die momentane Ressource über Hypermedia enthalten ist. D.h. in dem Text stehen Hyperlinks zur Ressource. Die Zustandsänderungen bzw. -übergänge durch die REST-Anfragen erfolgen durch den Transfer der Daten, die den nächsten Zustand repräsentieren. Daher wird es auch Representational State Transfer genannt.

Richardson Maturity Model (RMM)

Das Richardson Maturity Model (siehe Abb. drei) ist ein von Leanoard Richardson entwickelter Maßstab, wie strikt ein Service REST implementiert. Das Level null gilt dabei noch als RESTless. Es steigt bis hin zu Level drei. Erreicht eine Implementierung ein solches Level, so gilt er als RESTfull und beachtet alle sechs Einschränkungen.



Level	Properties
0	• uses XML-RPC or SOAP • the service is addressed via a single URI • uses a single HTTP method (often POST)
1	• uses different URIs and resources • uses a single HTTP method (often POST)
2	• uses different URIs and resources • uses several HTTP methods
3	• is based on HATEOAS and therefore uses hypermedia for navigation • uses different URIs and resources • uses several HTTP methods

Abbildung 3: Richardson Maturity Modell[8]

[8] Screen shot Böck (o.D.), S.16

HTTP-Anfragen und dessen Statuscodes

Da REST hauptsächlich HTTP nutzt, ist es wichtig, dessen Anfragemethoden zu kennen. Die sieben Wichtigsten sind GET, POST, PUT, HEAD, PATCH, DELETE und OPTIONS.[9] Sie sind in der Tabelle der Abbildung vier näher beschrieben.

GET	Anfordern der angegebenen Ressource vom Server; Sollten keine Nebeneffekte oder Zustandsänderung am Server hervorrufen
POST	Einfügen einer neuen (Sub-) Ressource unterhalb der Angegebenen; Adressiert die übergeordnete Ressource, da für sie noch keine eigene URI gibt; In Antwort kann URI der angelegten Ressource stehen
PUT	Die angegebene Ressource wird unter angegebener URI angelegt; Falls sie schon existiert, wird sie verändert
HEAD	Anfordern des Nachrichtenheaders der Ressource; Anfordern der Metadaten zur angegebenen Ressource
PATCH	Ändern eines Teils der angegebenen Ressource; Nebeneffekte sind erlaubt; → Update
DELETE	Löschen der angegebenen Ressource beim Server
OPTIONS	Prüfen, welche Methoden auf der Ressource zur Verfügung stehen

Abbildung 4: Übersicht über die HTTP-Anfragemethoden

Um den Erfolg der Anfrage in der Antwort direkt zu identifizieren, gibt es spezifische Statuscodes von HTTP. Sie werden extra mitangegeben. Es gibt fünf Kategorien (siehe Abb. fünf).[10]

1xx	Informational
2xx	Success
3xx	Redirection
4xx	Client Error
5xx	Server Error

Abbildung 5: Übersicht der HTTP Status Codes

Die Codes bestehen aus drei Ziffern. Die Erste gibt dabei die Kategorie an. 1xx steht für Informationell. Dabei wird angegeben, die Bearbeitung der Information dauert noch an. 2xx bedeutet, dass die Anfrage vom Klienten erfolgreich war. 200 steht dabei für

[9] Fielding, et al.: “Hypertext Transfer Protocol – HTTP/1.1”, in: “https://datatracker.ietf.org/doc/html/rfc2616”, (o.D.), Zugriff: 07.05.2021
[10] Fielding, et al.: “10 Status Code Definitions”, in: “https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html”, (o.D.), Zugriff: 07.05.2021

OK, 201 für Created, 202 für Accepted, 203 für Non-Authotitative Information und 204 für No Content. Es wird dementsprechend kein weiterer Code benötigt, da die Anfrage erfolgreich war. Der Body enthält eine Entität, die der angefragten Ressource oder dem Ergebnis entspricht. Bei 201 wurde die Ressource erstellt oder angelegt. 202 bedeutet, dass die Anfrage angenommen wurde, aber die angefragte Aktion noch länger dauert. 204 wird gesendet, wenn die REST-API es abweist, eine Statusnachricht zu senden, die Anfrage aber erfolgreich war oder wenn die Ressource angelegt ist, aber es keine URI oder andere Repräsentation davon gibt. Durch die dritte Kategorie wird ausgesagt, dass der Klient zusätzliche Aktionen vornehmen muss, dass die Anfrage erfolgreich oder vervollständigt wird. Die Vierte bedeutet, dass ein Fehler auf Klientenseite vorliegt. 400 steht für Bad Request, 401 für Unathorized, 403 für Forbidden, 404 für Not Found und 405 für Method Not Allowed. 400 ist ein generischer Fehlerstatus, der eintritt, wenn kein anderer aus der Kategorie zutrifft. Beispielsfälle sind falsch ausgeführte Anfragesyntax, -parameter oder ein irreführendes Anforderungsrouting. Bei 401 ist die benötigte Autorisierung falsch oder gar nicht angegeben in der Anfrage. Wenn die Anfrage korrekt, aber der Zugriff nicht erlaubt ist, wird 403 gesendet. 404 tritt auf, wenn hinter der URI aus der Anfrage keine Ressource steht, sie also zu keiner passt. Bei 405 ist die gesendete Methode auf der Ressource nicht erlaubt. Die fünfte Kategorie zeigt, dass beim Server ein Fehler vorliegt. Er übernimmt die Verantwortung. 500 bedeutet Internal Server Error. Der Statuscode wird gesendet, wenn in dem Anfrage-Handler-Code eine Exception ausgelöst wird. Bei 501 (Not Implemented) erkennt der Server die Anfragemethode nicht oder kann sie nicht ausführen. Die genannten Codes stellt nur ein Ausschnitt der HTTP-Statuscodes und zugleich die Wichtigsten dar!

Request-Aufbau (von HTTP)

Eine HTTP-Anfrage und demnach auch eine REST-Anfrage hat einen bestimmten Aufbau. Zuallererst wird eine Uniform Ressource Locator (URL) benötigt. Darin wird der Endpunkt festgelegt, hinter der eine Ressource auch steht. Sie kann Parameter enthalten, die mit einem Fragezeichen oder Und-Zeichen angehangen werden. Der nächste Teil ist der Header. In ihm stehen die Metainformationen zur Ressource und zur Anfrage. Beispielsweise wird der Content-Type angegeben. Im Header können ebenfalls Eigenschaften zur Autorisierung stehen. Unter anderem sind mögliche

Autorisierungen Tokens, API-Keys oder Basic Authorization. Die Letztere erfolgt via Benutzername und Passwort, die der Anfrage mitgegeben werden.

Vorteile

REST ist simple und standardisiert. Es ist aufgrund der einheitlichen Schnittstellen einfach zu nutzen. Des Weiteren werden standardisierte Verfahren wie JSON und HTTP/S benutzt. Außerdem muss sich nicht um die Formatierung der Ressource gekümmert werden.

Es wird eine erhöhte Zustandslosigkeit erreicht und die Anfragen sind zustandslos. Da die Anfragen in sich geschlossen sind, können sie im Zuge der Lastverteilung auf verschiedene Systeme verteilt werden. Dadurch entsteht eine bessere Balance. Kein Server muss den Zustand vom anderen wissen, um die Anfrage umzusetzen. Die Anwendungsinfos sind ausschließlich auf Klientenseite z.B. durch Cookies vorgehalten.

Es wird eine höhere Performance durch Caching erreicht. Das entlastet die Datenbank und den Server. Bei wiederholter Anfrage kann die zwischengespeicherte Information, die auf Klientenseite ist, aufgerufen werden. Somit muss der Server nichts nochmal abrufen.

Eine verbesserte Portabilität der Benutzeroberfläche über mehrere Plattformen hinweg durch das Klienten-Server-Modell wird erreicht. Eine Trennung der Datenschicht von der Benutzerschnittstelle durch einheitliche Schnittstellen erfolgt. Der Klient befasst sich nicht mit der Datenspeicherung, die intern auf dem Server verbleibt. Deswegen hat man eine verbesserte Portabilität. Des Weiteren führt die Trennung zur möglichen unterschiedlich schnellen Entwicklung der Komponenten.

REST stellt gut strukturierte Dienste durch die Einschränkungen zur Verfügung. Außerdem unterstützt es die Verwendung von Clean URLs. Sie enthalten lesbare Wörter anstelle von technischen Kürzeln der Datenbank-IDs. Daraufhin kann der Benutzer die URLS bezüglich ihrer Relevanz schneller bewerten. Er/Sie kann sie sich einfacher merken.

TLS

In diesem Abschnitt wollen wir uns zunächst etwas genauer mit der Frage beschäftigen, was TLS/SSL ist und wie es funktioniert.

TLS Basics

Transport Layer Security (TLS) ist ein Verschlüsselungsprotokoll, das zur Sicherung der Datenübertragung im Internet entwickelt wurde. Es ist üblicherweise beim Zugriff auf Websites über HTTPS zu finden, kann aber auch an anderen Stellen eingesetzt werden, zum Beispiel bei der Kommunikation mit webbasierten Servern oder MQTT-Brokern.

TLS ist der Vorgänger des Secure Socket Layer (SSL) Protokolls. Dabei ist zu beachten, dass SSL-Zertifikate nicht dasselbe sind wie das SSL-Protokoll. Während das TLS-Protokoll sicherer ist als SSL - da es eine aktualisierte Version ist, die Exploits und Sicherheitsprobleme des alten Protokolls behebt - verwenden wir immer noch SSL-Zertifikate für die Authentisierung.

Die Protokollkommunikation besteht aus zwei wesentlichen Teilen:

1. Der TLS Handshake - der Aufbau einer sicheren Verbindung.
2. Der TLS Record - der verschlüsselte Transport von Daten.

Es gibt zwei Möglichkeiten, TLS zu verwenden:

1. Verwendung von selbstsignierten Zertifikaten: In diesem Fall würde man einen privaten Schlüssel für die Entschlüsselung und ein Server-Zertifikat erstellen, das selbst signiert wäre, ohne dass eine sogenannte "Zertifizierungsstelle" (eng. Certificate Authority) existiert, welche die Gültigkeit des Zertifikats bestätigen könnte.
2. Verwendung einer Zertifizierungsstelle: Erstellen einer Certificate Authority, die unser Server-Zertifikat signieren und validieren kann.

Für dieses Projekt wurde die erste Variante gewählt. Im folgenden Abschnitt werden die notwendigen Schritte zum Aufbau einer sicheren TLS-Verbindung erläutert.

TLS Handshake

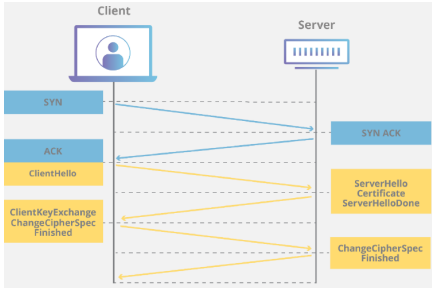
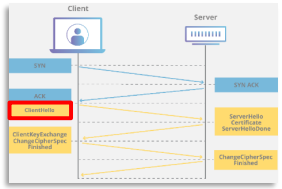


Abbildung 6: TLS Handshake

Die Abbildung zeigt die notwendigen Schritte, um eine sichere Verbindung aufzubauen. Es handelt sich um den sogenannten TLS Handshake (durch die gelben Pfeile gekennzeichnet), der nach einem regulären TCP Handshake (durch die blauen Pfeile gekennzeichnet) erfolgt.



Schritt 1:

Der Client sendet eine "ClientHello"-Nachricht an den Server, die eine Auswahl an unterstützten TLS-Versionen und so genannten "Cipher Suites" enthält, welche von dem Client ebenfalls unterstützt werden. Außerdem sendet er eine zufällige Folge von Bytes, die "Client Random" genannt wird.

Cipher Suites sind eine Sammlung von Verschlüsselungsalgorithmen. Jede Suite ist eine Kombination aus[11]:

- Key Exchange Algorithmen (RSA, DH, ECDH, DHE, ECDHE, PSK)
- Authentication/Digital Signature Algorithmen (RSA, ECDSA, DSA)
- Bulk Encryption Algorithmen (AES, CHACHA20, Camellia, ARIA)
- Message Authentication Code Algorithmen (SHA-256, POLY1305)

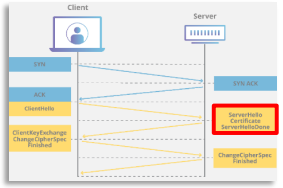
Damit könnte eine Cipher-Suite wie folgt aussehen:

TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

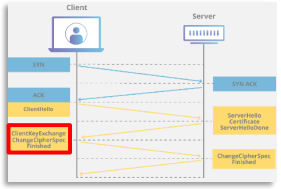
[11] Quelle: https://www.thesslstore.com/blog/cipher-suites-algorithms-security-settings/

Die farbliche Markierung zeigt hierbei die Zuordnung zu den verschiedenen Algorithmen.

Schritt 2:



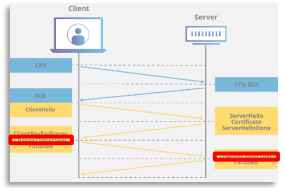
Als Antwort auf das ClientHello sendet der Server eine "ServerHello"-Nachricht zurück an den Client. Mit dieser Nachricht sendet er sein SSL-Zertifikat, um seine Identität zu verifizieren, die TLS-Version und die Cipher-Suite, die der Server aus der Auswahl, die er vom Client erhalten hat, ausgewählt hat, und eine zufällige Folge von Bytes, die "Server Random" genannt wird.



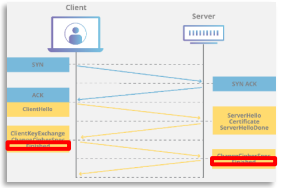
Schritt 3:

Der Client prüft das SSL-Zertifikat, das er vom Server erhalten hat, bei der Zertifizierungsstelle, die es signiert hat. Bei selbstsignierten Zertifikaten gäbe es keine Zertifizierungsstelle, gegen die er sich authentifizieren könnte. Die Verbindung würde trotzdem aufgebaut und verschlüsselt werden, aber sie würde als unsicher angesehen werden. Aus diesem Grund muss man die Verbindung ausdrücklich zulassen.

Schritt 4:



Wenn die Authentifizierung erfolgreich war, sendet der Client eine weitere zufällige Bytefolge namens "Premaster-Secret" und verschlüsselt sie mit dem öffentlichen Schlüssel, der im SSL-Zertifikat des Servers enthalten war.
Der Server kann das Premaster-Secret mit dem Private Key entschlüsseln. Client und Server können nun den Client-Random, den Server-Random und das Premaster-Secret verwenden, um einen Session-Key zu erstellen, der zur Verschlüsselung der Verbindung verwendet wird.



Schritt 5:

Server und Client sollten denselben Session Key erzeugen. Um zu testen, ob alles funktioniert hat und beide zur gleichen Lösung gekommen sind, senden Server und Client jeweils eine mit dem Sitzungsschlüssel verschlüsselte "Finish"-Nachricht.

Schritt 6:

Die Kommunikation ist nun verschlüsselt und kann mit dem Sitzungsschlüssel verschlüsselt weitergeführt werden.

Recherche

Die erste Herausforderung bei der Umsetzung einer REST-API mit verschlüsselter Kommunikation, war die Internetrecherche nach geeignetem Quellcode, der diese Funktionalität mit möglichst geringem Aufwand möglich macht. Im Nachfolgenden werden Bibliotheken vorgestellt, die diese Funktionalitäten bereitstellen.

Mongoose – Embedded Networking Library:

Bei der Suche nach einer passenden Bibliothek, die sowohl die nötigen Funktionen bietet als auch auf dem vorgegebenen Entwicklungsboard lauffähig ist, viel die Wahl nach eingehender Recherche auf Mongoose. Die Hauptgründe für diese Wahl, waren die zahlreichen Beispiele, ein bereits existierender Port speziell für den TM4C129exl und die sehr gute Dokumentation. Bei Mongoose handelt es sich um eine Netzwerkbibliothek für die Programmiersprachen C und C++. Sie bietet eventgesteuerte APIs für TCP, UDP, HTTP, WebSocket und MQTT. Für die Implementierung einer REST-API war hierbei vor allem die Unterstützung für TCP und HTTP entscheidend. Darüber hinaus verfügt die Bibliothek über eingebaute Funktionalitäten für Verschlüsselte Kommunikation per TLS/SSL durch die native Unterstützung von mbedTLS oder OpenSSL und für die Unterstützung von Netzwerk-Stacks wie LWIP und FreeRTOS-Plus-TCP. Der Einzige Negativpunkt belief sich darauf, dass eine ältere Version der Bibliothek benutzt werden musste, da die neueren Versionen das verwendete Entwicklungsboard nicht mehr unterstützen. Da aber schlussendlich die Vorteile deutlich dem negativen überwogen, viel die Wahl auf diese Bibliothek.[12]

MbedTLS

Da die Kommunikation zwischen Client und Server mithilfe von TLS verschlüsselt werden sollte, wurde eine Bibliothek benötigt, die mit möglichst wenigen Ressourcen auskommt und kompatibel mit Mongoose ist. Da Mongoose OpenSSL und MbedTLS direkt unterstützt, war schnell klar, dass die Entscheidung auf eine dieser beiden Bibliotheken fällt. Beide Bibliotheken erfüllen die Anforderungen und so war der einzige Unterschied, der Portierungs-Aufwand. Nach der Einbindung beider

[12] https://github.com/cesanta/mongoose

Bibliotheken in das Projekt, wurde schnell klar, dass die Portierung von MbedTLS viel einfacher sein würde als die von OpenSSL. Welche Codeteile genau angepasst werden mussten, wird im Abschnitt „Wichtige Codeteile“ genauer erläutert. MbedTLS bietet folgende Features:

- Server API und Client API unterstützen SSL Version 3 und TLS 1.0, 1.1 und 1.2
- Schlüssel-Austausch Methoden: RSA, DHE-RSA, ECDHE-RSA, ECDH-RSA, ECDHE-ECDSA, ECDH-ECDSA, PSK, DHE-PSK, ECDHE-PSK, RSA-PSK
- Symmetrische Verschlüsselungsalgorithmen: AES, Blowfish, Triple-DES, DES, ARC4, Camellia, XTEA
- Hash-Algorithmen: MD2, MD4, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, RIPEMD-160

Aus dieser Liste wird ersichtlich, dass MbedTLS trotz der ressourcensparenden Implementierung eine Vielzahl an Technologien unterstützt und sich somit perfekt für den Einsatz in diesem Projekt eignet[13].

Mjson

Da die POST/GET Anfragen und Antworten, JSON enthalten sollten, war es nötig eine Bibliothek einzubinden, die in der Lage ist, JSON zu verstehen. Mjson ist eine Bibliothek, die genau diese Funktionalität bietet und mit weniger als 1000 Zeilen Code für Embedded Anwendungen sehr gut geeignet ist[14].

Laborübung

Versuchsaufbau

Der Laborversuch bestand aus zwei Teilen. In Teil a) sollten die Studierenden zuerst das Projekt herunterladen und mithilfe des Quickguides in CCS importieren. Anschließend sollte die Funktionalität der bereits vorprogrammierten REST Endpunkte mithilfe von Postman oder eines Python Skriptes getestet werden.

[13] https://github.com/ARMmbed/mbedtls
[14] https://github.com/cesanta/mjson

Das Projekt wurde voreingestellt mit TLS ausgeliefert. Da dafür selbstsignierte Zertifikate verwendet wurden, war es wichtig, in Postman vor dem Testen der Endpunkte „SSL certificate verification“ zu deaktivieren.

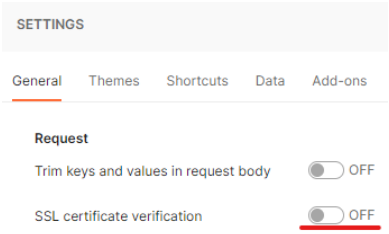


Abbildung 7: Postman Einstellungen

Wie bereits im TLS Abschnitt beschrieben gibt es bei selbstsignierten Zertifikaten keine Certificate Authority, wo man das SSL Zertifikat verifizieren könnte. Das Python Skript sah wie folgt aus:

```python
import requests
import json

# Hier eigene URL eintragen (http, wenn SSL deaktiviert ist)
url = "https://192.168.178.55/display"

payload = json.dumps({
  "display": "hello world"
})
headers = {
  'Content-Type': 'application/json'
}

response = requests.request(
    "POST", url, headers=headers, data=payload, verify=False)

print(response.text)
```

Abbildung 8: Python POST Skript

Es handelte sich um eine einfache POST Methode, die über die Übermittlung einer JSON Nachricht eine Ausgabe am Display des Mikrokontrollers erzeugt. Auch hier war wieder wichtig mit „verify=False“ die Verschlüsslung über selbstgezeichnete Zertifikate zu erlauben.
Im Teil b) der Übung sollte ein zusätzlicher Endpunkt im Programm gesetzt werden, der bei einem POST request eine RGB LED am Mikrokontroller in der entsprechenden Farbe an- bzw. ausschalten sollte.

Versuchsdurchführung

Teil a)

Der Import des Projektes erfolgte wie gewohnt. Die Studierenden bekamen einen .zip File,der das Projekt enthielt. Dieses musste über den Import Dialog importiert werden:

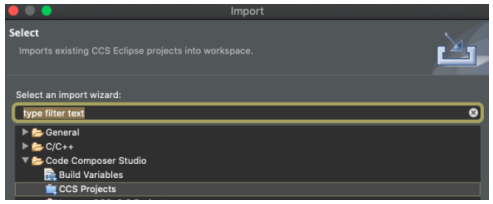


Abbildung 9: Auswahl des Import formats

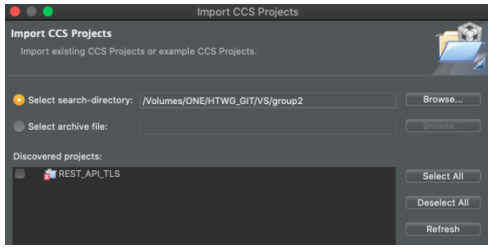


Abbildung 10: Projekt Auswahl

Um die vorhandenen Endpunkte zu testen musste anschließend TM4C Controller über USB und Ethernet Kabel mit dem PC verbunden werden. Nach Auswahl der Debug Funktion musste vor dem Start ein Terminal für den USB Port geöffnet werden. Nach Start des Debug Prozesses sollte anschließend im Terminal die Host Adresse für die REST Endpunkte angezeigt werden.

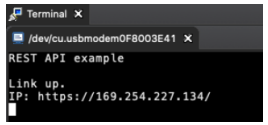


Abbildung 11: USB Terminal

Die Studierenden hatten zusätzlich die Moglichkeit die TLS Verschlüsslung im Projekt zu deaktivieren.

```
33  #ifdef MG_ENABLE_SSL
34  #undef MG_ENABLE_SSL
35  //activate (1) /deactivate (0) SSL
36  #define MG_ENABLE_SSL 1
37  #endif
```

Abbildung 12: TLS Aktivierung/Deaktivierung

Im File „REST_API_TLS.c“ musste dafür einfach die Variable „MG_ENABLE_SSL“ auf 0 gesetzt werden.

Nach Start des Debugging Prozesses konnten die Studierenden folgende Endpunkte testen:

GET Temperature API: Gibt Temperaturwerte vom Temperatursensor in °C und F zurück.

GET	<IP Adresse>/temperature
Request Body	Leer
Response Body	`{ "temperature": [ { "°C":"..." }, { "°F":"..." } ] }`

POST UART API: Gibt die Angaben, die vom REST Request kommen, über UART in der Konsole aus.

POST	<IP Adresse>/UART
Request Body	`{ "UART":"<deine Eingabe>" }`
Response Body	`{ "success":"<true|false>" }`

POST Display API: Gibt Angaben, die vom REST Request kommen, auf dem Display des Controllers aus.

POST	<IP Adresse>/display
Request Body	`{ "display":"<deine Eingabe>" }`
Response Body	`{ "success":"<true|false>" }`

Teil b)

Ziel war die Implementierung des folgenden Endpunktes:

POST LED API: Wechselt die Farbe der Controller LED entsprechend der RGB einstellung.

- state:0 schaltet die LED ab, in diesem Fall muss keine Farbe im Body mitgeschickt werden.
- state:1 aktiviert die LED, für die Farbe muss jeweils ein Wert von 0 bis 255 gewählt werden.

POST	<IP Adresse>/led
Request Body	{ "state":<1|0>, "color":{ "r": <0 bis 255>, "g": <0 bis 255>, "b": <0 bis 255> } }
Response Body	{ "success":"<true|false>" }

Dafür musste folgender Code zum „helper_functions.c“ File hinzugefügt werden:

```
57      // Practice - Create new endpoint here
58
59      else if(mg_vcmp(&hm->uri, "/led") == 0){
60          if(mg_vcmp(&hm->method, "POST") == 0){
61              if(check_for_json(hm)){
62                  if(led_request(hm)){
63                      send_response_success(nc, hm, addr);
64                  }
65                  else{
66                      send_response_bad_request(nc, hm, addr);
67                  }
68              }
69              else{
70                  send_response_bad_request(nc, hm, addr);
71              }
72          }
73      }
```

Abbildung 13: Lösung für Aufgabenteil b)

Die Studierenden konnten sich dabei an den bereits implementierten APIs orientieren, da diese in ähnlicher Weise geschrieben wurden. Die eigentliche Funktion „led_request“ zum interpretieren des Request Bodys musste dabei nicht geschrieben werden und war bereits implementiert.

Wichtige Codeteile

Eventhandler Funktion – Mongoose

Die Eventhandler Funktion wird aufgerufen, sobald eine Verbindung zum Server hergestellt wird. Es können unterschiedliche Events zu der switch-case hinzugefügt werden. In diesem Fall werden vier Events behandelt:

- MG_EV_POLL: Iteration der mg_mgr_poll Funktion
- MG_EV_ACCEPT: Die Verbindung wurde akzeptiert
- MG_EV_HTTP_REQUEST: Eine HTTP-Anfrage wurde an den Server gestellt
- MG_EV_CLOSE: Die Verbindung wurde geschlossen

```
void ev_handler(struct mg_connection *nc, int ev, void *ev_data) {

  if (ev == MG_EV_POLL) return;
  char addr[32];

  switch (ev) {
    //Message accepted
    case MG_EV_ACCEPT: {
      mg_sock_addr_to_str(&nc->sa, addr, sizeof(addr),
                          MG_SOCK_STRINGIFY_IP | MG_SOCK_STRINGIFY_PORT);
      UARTprintf("%p: Connection from %s\r\n", nc, addr);
      break;
    }
    case MG_EV_HTTP_REQUEST: {
      struct http_message *hm = (struct http_message *) ev_data;

      if(mg_vcmp(&hm->uri, "/display") == 0){
          if(mg_vcmp(&hm->method, "POST") == 0){
              if(check_for_json(hm)){
                  if(display_request(hm)){
                      send_response_success(nc, hm, addr);
                  }
                  else{
                      send_response_bad_request(nc, hm, addr);
                  }
              }
              else{
                  send_response_bad_request(nc, hm, addr);
              }
          }
      }
```

```
    else if(mg_vcmp(&hm->uri, "/UART") == 0){
        if(mg_vcmp(&hm->method, "POST") == 0){
            if(check_for_json(hm)){
                if(uart_request(hm)){
                    send_response_success(nc, hm, addr);
                }
                else{
                    send_response_bad_request(nc, hm, addr);
                }
            }
            else{
                send_response_bad_request(nc, hm, addr);
            }
        }
    }
    else if(mg_vcmp(&hm->uri, "/led") == 0){
        if(mg_vcmp(&hm->method, "POST") == 0){
            if(check_for_json(hm)){
                if(led_request(hm)){
                    send_response_success(nc, hm, addr);
                }
                else{
                    send_response_bad_request(nc, hm, addr);
                }
            }
            else{
                send_response_bad_request(nc, hm, addr);
            }
        }
    }


    else if(mg_vcmp(&hm->uri, "/temperature") == 0){
        if(mg_vcmp(&hm->method, "GET") == 0){
            send_response_temperature(nc, hm, addr);
        }
        else{
            send_response_bad_request(nc, hm, addr);
        }
    }
    else{
        send_response_not_found(nc, hm, addr);
    }
    break;
  }
  //Close event
  case MG_EV_CLOSE: {
    UARTprintf("%p: Connection closed\r\n", nc);
    break;
  }
 }
}
```

Aus dem Codeabschnitt wird ersichtlich, dass in der switch-case Verknüpfung, diese vier Events abgeprüft werden. Besonders interessant ist hierbei der MG_EV_HTTP_MESSAGE Case. In diesem wird der entsprechende Endpunkt auf

Basis der URI abgeprüft. Vier verschiedene Endpunkte wurden implementiert (/display, /UART, /led, /temperature). Wenn einer dieser Endpunkte mit der entsprechenden Request-Methode (GET oder POST) angesprochen wird, wird eine Funktion aufgerufen, die diese Anfrage behandelt und anschließend eine Antwort sendet. Der MG_EV_HTTP_MESSAGE Case wird in Abb(?) nochmals in einem Float-Chart veranschaulicht.

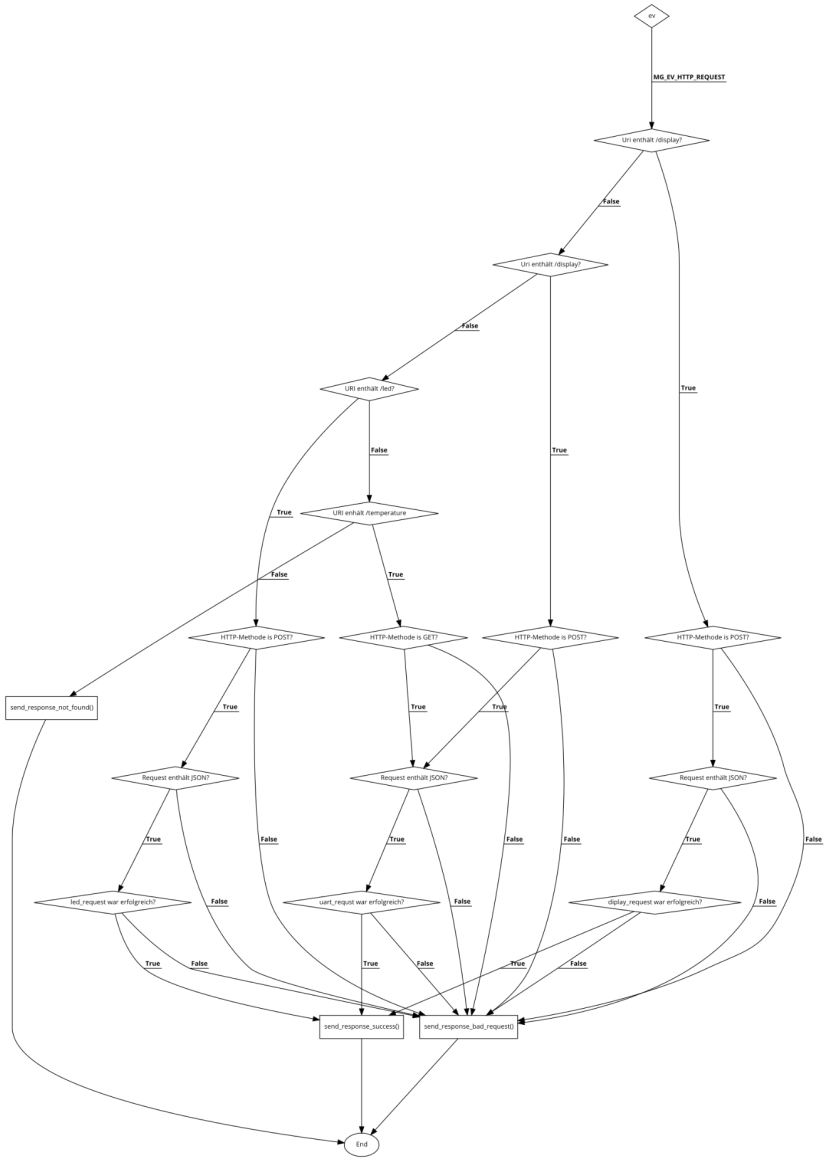


Abbildung 14: Ablauf des MG_EV_HTTP_MESSAGE Case

Einlesen des selbstgezeichneten Zertifikats in Mongoose

Da auf dem verwendeten Entwicklungsboard keine SD-Karte oder anderer geeigneter Speicher zur Verfügung stand, mussten das Zertifikats- und Keyfile als Array in einem Headerfile gespeichert werden. Um diesen Prozess so einfach wie möglich zu gestalten, wurde hierfür das enet_io Beispiel, welches in der TivaWare enthalten ist, herangezogen. Dieses beinhaltet ein CMD-File, welches erlaubt genau ein solches Headerfile zu erzeugen. Die im Folgenden gezeigte Funktion (*mbedts_pk_load_file*) wurde im pkparse.c File in MbedTLS angepasst. Sie ruft die parse_file_from_fs Funktion mit dem entsprechenden Array (*data_server_pem* oder *data_server_key*) auf.

```
int mbedtls_pk_load_file(const char *path, unsigned char **buf, size_t
*n )
{
    if(strcmp(path, "server.pem") == 0){
        size_t fs_size = sizeof(data_server_pem)/sizeof(uint8_t);
        parse_file_from_fs(data_server_pem, fs_size,buf, n);
    }
    else if(strcmp(path,"server.key") == 0){
        size_t fs_size = sizeof(data_server_key)/sizeof(uint8_t);
        parse_file_from_fs(data_server_key, fs_size, buf, n);
    }
    else{
        return MBEDTLS_ERR_PK_FILE_IO_ERROR;
    }
    return( 0 );
}
```

Die Funktion *parse_file_from_fs*, schreibt das Zertifikat oder den Key, aus dem Headerfile in einen Buffer, welcher von MbedTLS benutzt wird.

```
void parse_file_from_fs(const uint8_t *fs_file, size_t fs_size,
unsigned char **buf , size_t *n){

    long path_size;

    for (int i = 0 ; i < fs_size; ++i)
    {
        if(fs_file[i] == 0x00){
            path_size = i + 1;
            break;
        }
    }

    *n = fs_size - path_size;
    *buf = mbedtls_calloc(1, *n + 1);

    for (int i = 0 ; i < *n ; ++i)
    {
        (*buf)[i] = fs_file[i+path_size];
    }

    (*buf)[*n] = '\0';

    if( strstr( (const char *) *buf, "-----BEGIN " ) != NULL )
        ++*n;
}
```

Der obige Code-Abschnitt zeigt die Implementierung der Funktion *parse_file_from_fs*. Diese besteht hauptsächlich aus zwei aufeinanderfolgenden for-loops. Im ersten for-loop wird, der tatsächliche Anfang des Fileinhalts gefunden. Das Array enthält zu Beginn noch den Variablennamen, welcher in diesem Kontext nicht benötigt wird. Durch die erste Schleife wird geprüft, wann zum ersten Mal der Wert 0x00 im Array auftaucht, da nach diesem Wert der Variablenname zu ende ist und der Fileinhalt beginnt. Anschließend wird Speicher in dem Buffer bereitgestellt in der Größe des Arrays abzüglich des Variablennamens. In der nächsten for-loop wird der Array Inhalt nun in den Buffer geschrieben. Zum Schluss wird nun noch der \0 Character eingefügt, um zu signalisieren, dass der Buffer Inhalt zu Ende ist, wird aber in der Länge n nur berücksichtigt, falls der Bufferinhalt PEM codiert ist, was mit *if(strstr((const char *) *buf, „-----BEGIN“) != NULL)* abgeprüft wird.

Fazit

Abschließend lässt sich festhalten, dass alle Punkte der Aufgabenstellung erfolgreich implementiert wurden. Darüber hinaus könnten noch weitere Ressourcen angelegt und die Nutzung eines RTOS integriert werden. Außerdem könnte ein Speichermedium in den Microcontroller verbaut werden, welches in der Lage ist die TLS-Zertifikate zu speichern, wodurch die Zertifikate nicht in separaten Headerfiles gespeichert werden müssen.