Dokumentation

m0ments - Projekt C

Projektleiter: Prof. Dr. Andreas Plaß
Team:
Joel Ehlen (2288230) ,
André Voroschillin (2298891),
Sebastian Pehlke(2285904)

Einleitung

Das Ziel war es, eine Anwendung zu machen die es uns ermöglicht Bilder hochzuladen und zu empfangen. Es soll für die Endnutzer simpel zu nutzen sein und aus der programmiertechnischen Sicht einfach erweiterbar sein.

Für maximale Reichweite und optimale Nutzungsmöglichkeit haben wir uns dazu entschieden eine Anwendung für mobile Geräte auf beiden großen mobilen Plattformen Android und iOS, als auch eine Webseite zu erstellen.

Weitere Ziele waren es, die beiden Frontends (App u. Webseite) unabhängig vom Backend zu lassen und das Backend aus dem Internet erreichbar zu machen. Deswegen entschieden wir uns das Backend als eine REST API zu implementieren

Die Gruppeneinteilung wurde wie folgt entschieden. Wir haben drei große Aufgabenbereiche identifizieren können, diese waren die mobilen Applikationen, die Webseite und das Backend. Entsprechend haben wir unsere Gruppe, bestehend aus drei Mitgliedern, aufgeteilt.

Backend

Raspberry Pi Setup:

Der Raspberry Pi, auf dem das Backend läuft, ist standardmäßig aus dem Internet nicht erreichbar. Der einfachste Weg einen Server aus dem Internet erreichbar zu machen, ist es eine Portweiterleitung im Router/Firewall einzurichten. Eine Anfrage an den offenen Port wird dann intern vom Router an den Pi weitergeleitet. Der Provider vergibt bei jedem Verbindungsaufbau, und in der Regel mindestens alle 24 Stunden, eine neue IP-Adresse. Diese Adresse müsste bei Änderung für Dienste, die unseren Service nutzen, bekannt gemacht werden. Um dieses Problem zu lösen, kann man einen dynamischen DNS Dienst verwenden. Nach dem Einstellen des Dienstes wird der Router bei jeder Einwahl seine frisch

erhaltene IP an den Dienst melden. Das Heimnetzwerk ist damit immer unter dem selben Hostnamen erreichbar.
Ein Problem auf welches wir dabei gestoßen sind, ist, dass unser Internetprovider Internet-Anschlüsse mit Dual Stack Lite betreibt. Bei DS-Lite bekommt der Netzzugangsrouter des Kunden keine öffentliche IPv4-Adresse, sondern eine private IPv4-Adresse. Diese Adresse wir über IPv6 ins öffentliche IPv4-Netz getunnelt. Zwischen dem privaten Provider-Netz und dem öffentlichen IPv4-Netz vermittelt Carrier-Grade-NAT (network address translation). Der dafür zuständige NAT-Server steht beim Provider und kümmert sich um die Adressübersetzung zwischen den privaten und öffentlichen IPv4-Adressen und reicht die Pakete anschließend ins öffentliche IPv4-Netz weiter.

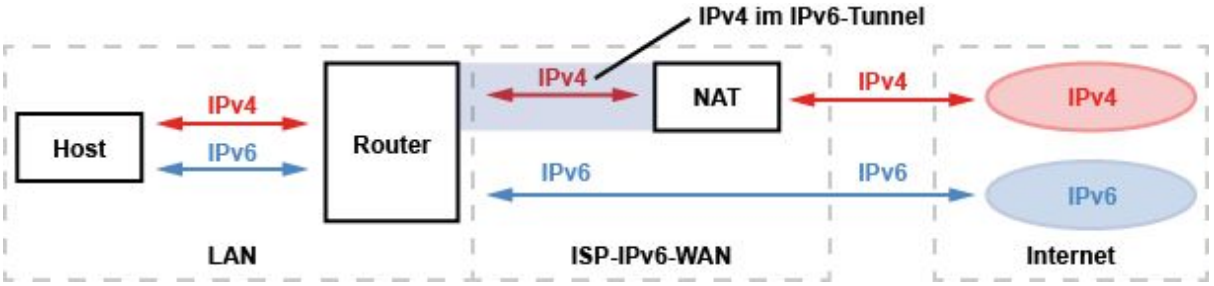

Abb 1: Quelle: https://www.elektronik-kompendium.de/sites/net/2010211.htm

Enige Netzwerk- und Internet-Dienste funktionieren bei Dual-Stack Lite nicht, wenn keine öffentliche IPv4-Adresse vergeben wurde. Beispielsweise funktioniert die von uns benötigte IPv4-Port-Weiterleitung am Router nicht mehr. Das Problem dabei ist, dass keine öffentliche IPv4-Adresse am Internet-Anschluss zur Verfügung steht. Konsequente Lösungen dafür wären den Dienst auf IPv6 umzustellen oder zu versuchen eine öffentliche IPv4-Adresse vom Provider zu beantragen.
Beide Lösungsansätze kamen für uns nicht in Frage, weshalb wir es mit einem Workaround versucht haben.

Reverse Tunnel

Ein Freund von uns ist im Besitz eines im Internet erreichbaren Servers. Der Server wird von der netcup GmbH gehostet und hat eine hohe Verfügbarkeit. Außerdem hat der Server feste Domains.
Die Idee war es von dem Raspberry Pi einen ssh Tunnel auf den aus dem Internet erreichbaren Server aufzubauen. Dazu braucht man den eingestellten ssh Port auf dem Server (660) und den neu angelegte User (tunnel).

```
ssh -p 660 tunnel@cepterhost.de
```

Dadurch besteht eine direkte, verschlüsselte Verbindung zwischen dem nicht im Internet verfügbaren Pi und dem aus dem Internet erreichbaren Server. Mit setzen des -R Flags ist es möglich neben dem ssh Tunnel einen weiteren Tunnel zu haben der Anfragen zum Host (cepterhost.de) auf einen bestimmten Port, durch den Tunnel zum Pi leitet.

```
ssh -R 4666:localhost:3000 -p 660 tunnel@cepterhost.de
```

Mit diesem Tunnel werden Anfragen zu cepterhost.de mit dem Port 4666 zum Pi zu Port 3000 weitergeleitet. Damit ist es möglich unseren Server, der auf dem Pi unter Port 3000 erreichbar ist, zu erreichen. Die Verbindung soll durchgehend bestehen, darum nutzen wir das Programm autossh, dass automatisch versucht die ssh-Verbindung aufzubauen sobald diese abgebrochen wird. Auf cepterhost.de wir mit dem Tunnel auch noch ein Skript ausgeführt, welches überprüft ob angegebene Ports noch erreichbar sind und sonst die ssh-Verbindung abbricht. Um den Tunnel direkt beim Neustarten des Pis aufzubauen haben wir den Befehl als Service unter systemd eingetragen. Der komplette Befehl sieht dann so aus:

```
/usr/bin/autossh -M 0 -o "ExitOnForwardFailure yes" -o
UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -o "ServerAliveInterval
30" -o "ServerAliveCountMax 3" -R *:4666:localhost:3000 -p 660
tunnel@cepterhost.de /opt/scripts/tunnel-endpoint-observer 4666
```

Ein auf cepterhost.de laufender nginx Server leitet anfragen auf pi.idletask.de zum Hostsystem auf Port 4666 weiter. Damit wird die Anfrage durch den Reverse Tunnel weitergeleitet und von unserem Node Backend auf Port 3000 verarbeitet. Der Node Service startet sich auch automatisch beim starten des Pis.

Installation

Zuerst muss das Repository https://github.com/idleTask/ProjektC geklont werden. Im Backend müssen alle Abhängigkeiten installiert werden (sudo yarn install). Das Node Backend wird gestartet mit ‘node server.js’.
Die Webseite muss über einen Webserver gehostet werden.
Die Flutter App muss gebaut und auf ein Smartphone geladen werden.

Datenbank:

Als Datenbank im Bankend benutzen wir MongoDB Atlas (https://www.mongodb.com/cloud/atlas). Dies ist eine MongoDB in der cloud. Wir nutzen die kostenlose Variante die 512mb Speicher bereitstellt. Die Bilder, die in der Applikation hochgeladen werden können, werden nicht in der Datenbank gespeichert. Es werden lediglich die Pfade zu den Bildern, die auf dem Pi gespeichert sind, in der Datenbank abgelegt.

API

Authentication:

Nach login bekommt man einen JSON Web Token. Dieser ist 1 Stunde gültig. Für geschützte Routen muss man im HTTP header den Parameter ‘Authorization’ auf

den Wert ‘Bearer $token‘ setzen.

Models:

User: id, name, email, password
Item: title, itemImage, description(optional), userId

Routes/Endpoints:

User

1. Signup user
   URL: /user/signup
   METHOD: post
   PARAMS: name:String(required, unique), email:String(required, unique), password:String(required)
2. Login user
   URL: /user/login
   METHOD: post
   PARAMS: email:String, password:String
   RESPONSE: {token:String}
3. Delete user
   URL: /user/:userId
   METHOD: delete
   PARAMS: -
   HEADER: Authorization:token
4. Get one user
   URL: /user
   METHOD: get
   PARAMS: -
   HEADER: Authorization:token
   RESPONSE: {name:String, email:String, id:String, items:[items]}
5. Get all users
   URL: /user/all
   METHOD: get
   PARAMS: -
   HEADER: Authorization:token
   RESPONSE: {count:Number, {users:[{email:String, name:String, id:String}]}
5.1 Update user

Items

6. Get all items
   URL: /items
   METHOD: get
   PARAMS: -

HEADER: Authorization:token
RESPONSE: {count:Number, items:[{title:String, id:String, itemImage:String}]}

7. Get one item
URL: /items/:itemId
METHOD: get
PARAMS: -
HEADER: Authorization:token
RESPONSE: {title:String, id:String, itemImage:String}

8. Create item
URL: /items
METHOD: post
PARAMS: title:String(required), description:String, itemImage:File(required)
HEADER: Authorization:token
RESPONSE: createdItem:{title:String, itemImage:String, _id:String}

9. Delete item
URL: /items/:itemId
METHOD: delete
PARAMS: -
HEADER: Authorization:token
RESPONSE:

10. Checken Update item
URL: /items/:itemId
METHOD: patch
PARAMS: [{propName:String, value:String}]
HEADER: Authorization:token

Uploads

11. Get image
URL: /uploads/{userId}/{itemImage}
METHOD: get
PARAMS: -
HEADER: Authorization:token
RESPONSE: file

Plugins

"**bcrypt**": A library to help you hash passwords.

"**express**": Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

"**jsonwebtoken**": An implementation of JSON Web Tokens.

"**mongoose**": Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box.

"**morgan**": HTTP request logger middleware for node.js

"**multer**": Multer is a node.js middleware for handling multipart/form-data, which is primarily used for uploading files.

"**nodemon**": nodemon is a tool that helps develop node.js based applications by automatically restarting the node application when file changes in the directory are detected.

Smartphone App

Konzept

Die Idee ist es eine für den Endnutzer simple, aber funktionale und übersichtliche Applikation zu gestalten. Die Plattform sollte dabei keine Rolle spielen, da wir in unserer Gruppe auf verschiedenen Plattformen arbeiten.

Zu Beginn haben wir eine Skizze gemacht, wie wir uns die App vorstellen.

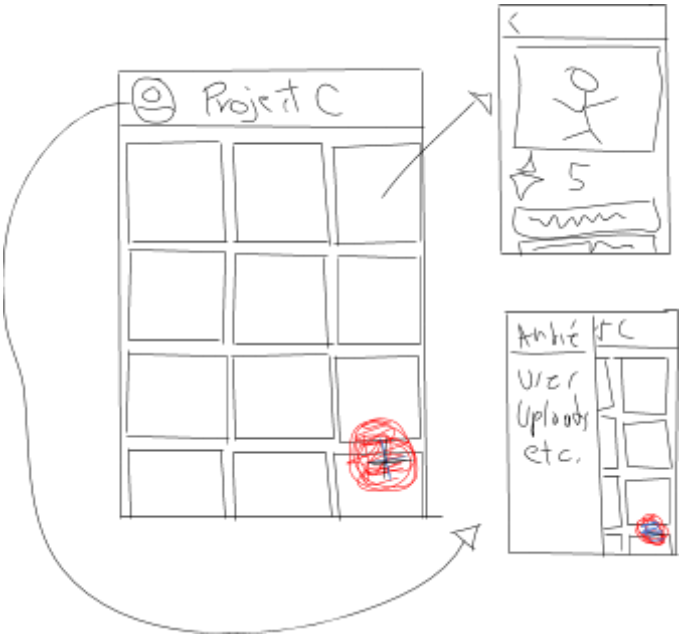

Abb 2: Erste Skizzen

Überblick und Navigation

Die Flutter App ist vom UI sehr übersichtlich gestaltet worden. Bei Start der Anwendung ist die *LoginPage* zu sehen. Hier hat man die Option sich einzuloggen wenn man die

entsprechenden Daten eingegeben hat. Falls man noch kein Konto hat gibt es über dem “Register” Knopf die Möglichkeit auf die *RegisterPage* zu wechseln und sich zu registrieren.

Bei erfolgreichen Login kommt man auf die *HomePage*. Es gibt eine *AppBar* und das Canvas für die Hauptanwendung. Die *AppBar* zeigt den Namen der App und ermöglicht es über das *Icon* auf der linken Seite einen *NavigationDrawer* zu öffnen. Dort hat man die Möglichkeit auf sein Profil zu gelangen, das Impressum anzuschauen oder sich auszuloggen.

Wenn der *HomeScreen* zum ersten Mal geladen wird, wird automatisch eine Netzwerkanfrage gestellt um die Posts zu laden. Diese werden in einem *GridView* als kleine klickbare *Cards* angezeigt. Wenn man auf eine der *Cards* klickt, kommt man auf die *DetailedPage*.

Auf der *DetailedPage* wird oben in der *AppBar* nun der Titel des Bildes angezeigt. In der Mitte des Bildschirms, auf dem Canvas, ist das Bild zu sehen und unter dem Bild ist die zum Bild entsprechende Beschreibung. Oben in der *AppBar* ist, wo der *NavigationDrawer* war, jetzt ein Zurück-Knopf.

Wahl der Entwicklungsplattform

Die Wahl der Entwicklungsplattform fiel auf Flutter. Flutter ist ein von Google entwickeltes Open Source Framework für mobile Endgeräte und wird in der Programmiersprache Dart geschrieben. Mit Flutter sind wir in der Lage native Applikationen für Android und iOS auf einer gemeinsamen Codebase entwickeln zu können. Flutter ist noch ein sehr neues Framework, jedoch bieten third-party packages alle Funktionen die unsere App benötigt. Alles in einem haben wir Flutter als geeignete Plattform gesehen.

Third party packages

Wir haben die folgenden third party packages für unserer Anwendung verwendet. Alle imports werden dem Projekt in der pubspec.yaml Datei hinzugefügt.

```
dependencies:
  dio: 2.1.16 #http client used for image upload
  http: 0.12.0+2 #Future based http library used for all other http requests
  image_picker: ^0.4.5 #plugin to easily get access to local gallery/camera images
  flutter_bloc: ^0.13.0 #bloc plugin to make it easier to work with blocs
  flutter:
    sdk: flutter
```

Abb 3: third party packages

Pattern

Als Pattern haben wir ein von Google für Dart und Flutter empfohlenes Pattern, das Bloc Pattern, verwendet. Das Bloc Pattern ist ein auf Streams basiertes Pattern welches Events in States konvertiert. Es hilft uns das Projekt besser zu organisieren und zu ändern.

Um das Bloc Pattern effizienter nutzen zu können haben wir Gebrauch des oben erwähnten Bloc Plugins gemacht.

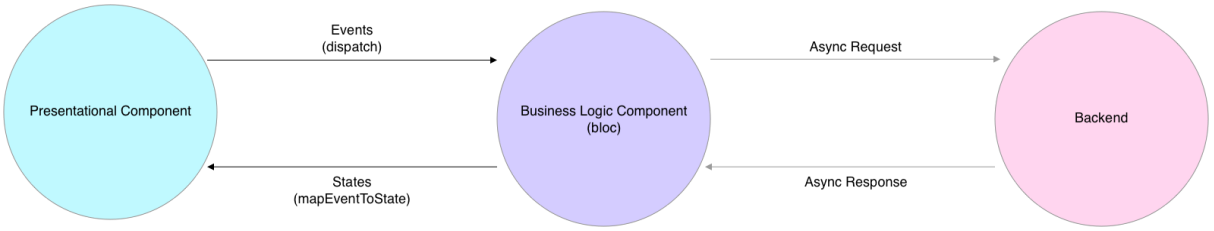

Abb 4: Bloc Beispiel, Quelle: https://pub.dev/packages/bloc

In der App werden drei Blocs für Datenmanagement benutzt.

ProfileBloc:

Im *ProfileBloc* werden die Profildaten des angemeldeten Benutzers gespeichert. Es bekommt *Profile* als State mit *ProfileEvent* die Events. Die Profildaten werden bei einem erfolgreichen Login in dem Bloc gespeichert und sind mit Hilfe des in der bloc library beinhalteten *BlocProviderTrees* von überall abrufbar. Events für den aktuellen *ProfileBloc* werden zu dieser Zeit nicht gebraucht, sind aber dank der Implementation und Natur von Blocs einfach zu integrieren.

M0mentCardBloc:

Der *M0mentCardBloc* erhält als State die *M0mentCard* und die Events aus *M0mentCardEvent*. Auch hier wird der Bloc primär für Datenspeicherung verwendet. Es werden z.B. Daten wie das Bild, ein Titel und eine Beschreibung gespeichert. Ein Bloc entspricht einem Element im Newsfeed auf der *HomePage* und erhält die Daten vom Backend.

CardListBloc:

Weil ein *M0mentCardBloc* Objekt nicht alle Elemente darstellen kann gibt es noch das *CardListBloc*. Das State ist *CardListState* und speichert eine Liste von *M0mentCardBlocs* als auch die Anzahl an List Elementen, für die Events sorgt *CardListEvent*. Der *M0mentCardBloc* ist in der Lage mehrere Events zu dispatchen um einfache Aktionen mit der Liste ausführen zu können, wie zum Beispiel das Hinzufügen einer *M0mentCard*.

Diese drei Blocs bilden den Grundbaustein der mobilen Applikation.

Netzwerkanbindung in Flutter

Die Netzwerkanbindung der Flutter App wurde hauptsächlich mit der *Flutter http*-, teilweise auch mit der *flutter dio* library realisiert. Die http Bibliothek ermöglicht uns mit Future basierte http Anfragen arbeiten zu können. Ein *Future* in Flutter ist das Ergebnis einer asynchronen Methode. Es erlaubt uns eine Operation ausführen zu können, ohne andere Operationen zu stoppen oder warten zu lassen. *Future* bietet außerdem die Möglichkeit Daten direkt in einem Objekt speichern zu lassen und es am Ende des Futures zurückzugeben, wovon wir Gebrauch gemacht haben.

Beispiel einer Netzwerkabfrage:

Es wird eine Methode erstellt, die ein *Future* wiedergibt. Dieser *Future* gibt ein Objekt in Form einer POJO Klasse wieder, welche die vom Server empfangenen Daten widerspiegelt. Es wird dem Backend eine Anfrage gestellt und das zurückkommende json In der *response* Variable gespeichert. Ist die Netzwerkanfrage fehlerfrei durchlaufen, wird der response dekodiert und die entsprechenden Werte in dem *Future* Objekt gespeichert und das Objekt returned. Falls es zu einem Fehler kommt, wird eine Exception mit einem http Fehlercode ausgeführt, bzw. ein *Alert* für den Endnutzer geworfen.

Das von der asynchronen Methode erhaltene *Future* Objekt wird ausgelesen und in dem entsprechenden *Bloc* gespeichert um lokal weiterverwendet werden zu können.

Für den Upload der Bilder war, aufgrund der Größe der Datei, ein *MultipartRequest* nötig. Bei Recherche zu diesem Thema sind wir auf die *Dio* Bibliothek gestoßen. Mit dieser war es uns möglich relativ simpel ein *MultipartRequest* über *FormData* zu machen.

Website

Die Website wurde im Entwicklungsverlauf als Testplattform benutzt um das Backend zu prüfen. Als der erste Entwurf der App fertig gestellt wurde, konnten wir so ebenfalls schnell testen, ob Bilder übertragen worden sind.
Die Intention der Website war es, sämtliche Funktionen übersichtlich und einfach erkennbar und prüfbar zu machen.

Überblick und Navigation

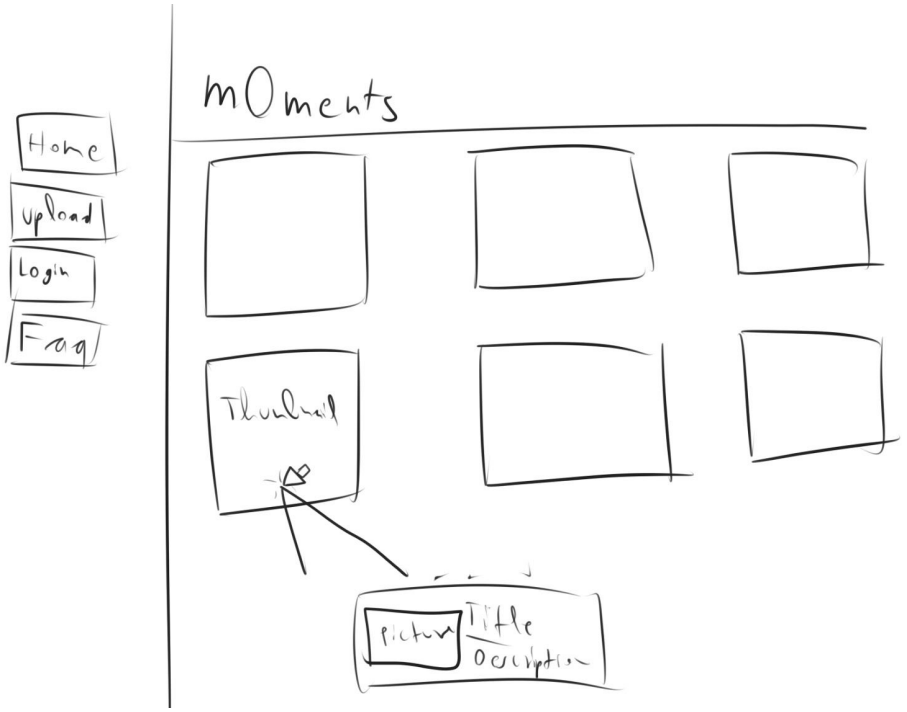

Abb. 5: Ursprüngliche Skizze des Homepage-Layouts

Der Aufbau der Website wurde hierfür nach ihren Funktionen in drei Html-Dokumente aufgeteilt: Homepage, Upload und Login/Registrierung.

Die Homepage gibt die hochgeladenen Bilder aufgelistet aus, sofern man angemeldet ist.

Die Bilder werden zunächst als kleine kastenförmige Thumbnails ausgegeben. Sobald der User auf die Thumbnails klickt, werden sie vergrößert und mit ihrem Titel und ihrer Beschreibung als Pop-Up angezeigt.
Oben links befindet sich ein Button, welcher bei Nutzung eine Sidebar öffnen und schließen kann. In der Sidebar befinden sich Hyperlinks, welche auf die anderen Html-Dokumente navigieren sowie ein Logout-Button für die Abmeldung des Users.

Die Upload-Seite besteht aus Eingabefeldern für den Titel und Beschreibung eines Bildes und einem Button, welcher ein Fenster zum Auswählen der Bilddatei öffnet.
Mit dem Drücken des Upload-Button werden die Daten ans Backend übertragen.

Das Registrierungs- und Login-Dokument besitzt statt der Bilder zwei Formulare wovon einer zum Einloggen und das andere Formular zum Registrieren dient.
Zurzeit ist die Registrierenfunktion allerdings ausgeschaltet, da es ein Sicherheitsrisiko für unseren Server darstellen würde. Das zugehörige Javascriptdokument lässt sich trotzdem in den Dateien finden.
Das Anmeldedaten für unser Beispielkonto zum Testen lauten:

admin@admin.de
G5lQJ5Sm0f5cupxe

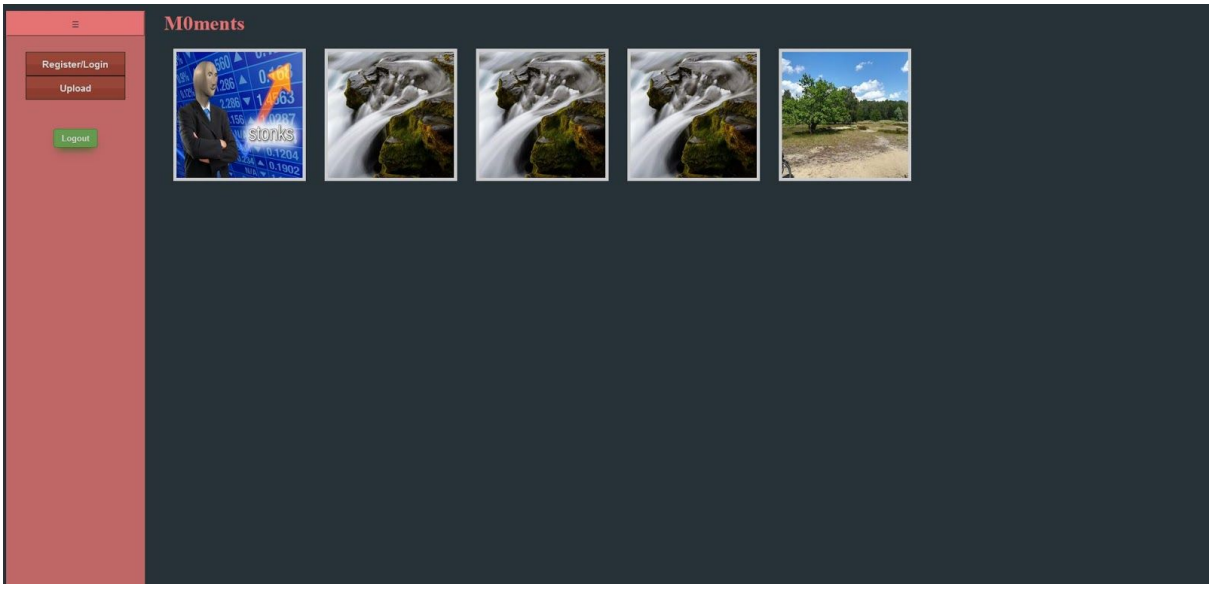

Abb. 6: Finales Layout der Homepage

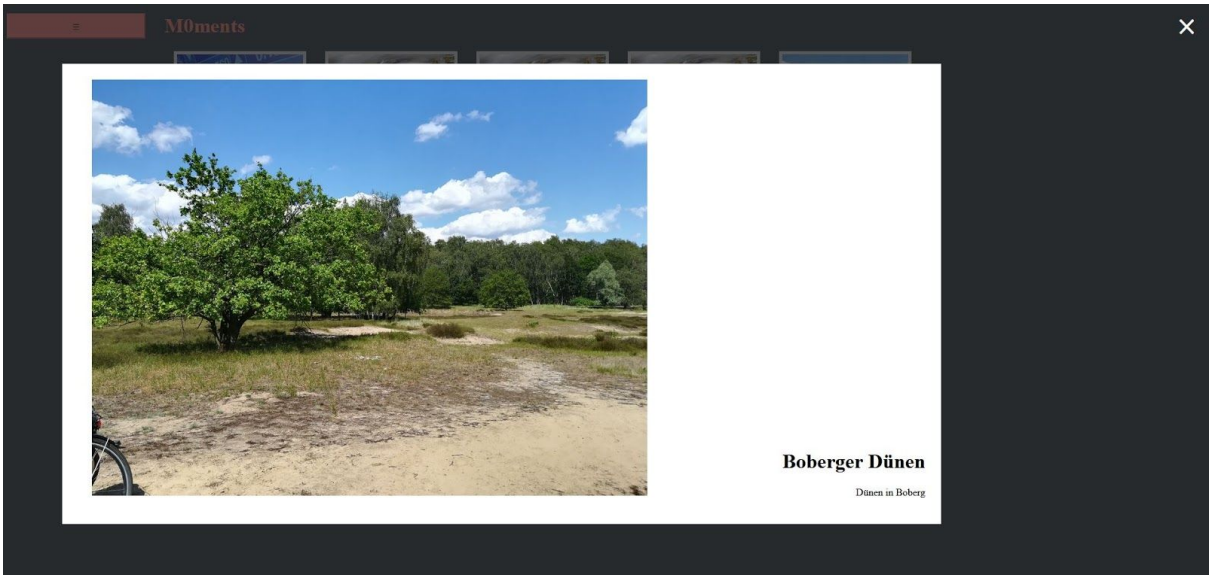

Abb. 7: Beispiel für die Wiedergabe von Bilddateien

Javascript

Die Website wurde erstellt mit HTML, CSS und Javascript und für die Anbindung zum Backend wurde mittels Javascript AJAX angewendet.

AJAX ist die Abkürzung für “Asynchronous Javascript and Xml”, welches wiederum ein Konzept für das asynchrone Laden von Website-Inhalten beschreibt. Um AJAX nutzen zu können, wurde die JQuery Bibliothek in unser Projekt eingebunden.

```
function onLogin() {
    var password, email, data;

    password = document.getElementById('passwordLogin');
    email = document.getElementById('emailLogin');

    data = {
        "password": password.value,
        "email": email.value
    }
    console.log(data);
    $.ajax({
        url: url + port + "/user/login",                    // Request URL
        type: "post",
        data: data,
        success: function (data)    //  function called when succeded
        {
            console.log('loggedIn');
            console.log(data);
            jwt = data.token;
            jwtString = "Bearer " + jwt;
            document.cookie = "jwt=" + jwtString;
            window.location.href = 'Homepage.html';
            getAllItems();
        },
        error: function (XMLHttpRequest, textStatus, errorThrown) {
            console.log('fehler');
        }
    });

}
```

Abb. 8: AJAX-Beispiel: onLogin() Funktion

Das Beispiel zeigt die Funktion, die aufgerufen wird, sobald man sich einloggt.
Nachdem die Login-Daten als “data”-Variable festgelegt wurden, werden die Daten mittels $.ajax(...) an das Backend gesendet und dort verglichen.
Wenn die Eingabedaten korrekt waren, wird Benutzer mit einem Cookie gekennzeichnet und bekommt die Authorization für die Bilddateien, welche über “getAllItems()” daraufhin abgefragt und angezeigt werden.

```
function displayItems(data) {


    var jwtString = getCookie('jwt');
    var items = data.items;
    items.forEach(function (item, index) {
        $.ajax({
            url: url + port + '/' + item.itemImage,                    // Request URL
            type: "get",
            contentType: 'image/png',
            headers: {
                "Authorization": jwtString,
            },
            mimeType: "text/plain; charset=x-user-defined",
            success: function (data)    //  function called when succeded
            {
                console.log('display success');
                var base64Image = base64Encode(data);
                $('#images').append("<div class=images><img class= 'displayImages' src='data:image;base64,"
                + base64Image + "' onclick='openModal(id)' height='200' width='200' id='" + index + "'></div>");
            },
            error: function (XMLHttpRequest, textStatus, errorThrown) {
                console.log(XMLHttpRequest);
                console.log(errorThrown);
                console.log('fehler get images');

            }
        });
    });

}
```

Abb.9: AJAX Beispiel: displayItems()

Als weiteres Beispiel habe ich hier noch die Funktion für das Anzeigen der Bilddateien eingefügt. Wenn der Cookie vorhanden ist, wird diese Funktion über getAllItems() ausgeführt.
Die Funktion fragt die Bilddaten beim Backend an und zeigt diese Bilder an nachdem sie mit der base64Encode() Funktion in den richtigen Zeichensatz/Charset überführt wurden.

Die meisten Funktionen unserer upload.js Datei funktionieren über solche Ajax Befehle.

```
function openModal(n)
{
    var image = document.getElementById("bigImage");
    document.getElementById("myModal").style.display = "block";

    image.src = document.getElementById(n).src;
    console.log(allItems.items);
    var items = allItems.items;
    $('#itemTitle').text(items[n].title);
    if (items[n].description != null) {
    $('#itemDescription').text(items[n].description);
    } else {
        $('#itemDescription').text("Description");
    }

}

function closeModal()
{
    document.getElementById("myModal").style.display = "none";
}
```

Abb. 10: Funktion zur Bildvergrößerung: openModal()

Um die Bilder auf unserer Website gleichmäßig anzeigen zu können, sind diese über CSS auf eine kleinere Größe skaliert. Wenn man die Bilder anklickt, öffnen sich die Bilder in ihrer eigentlichen Bildgröße durch die openModal()-Funktion. Hierbei wird das Anzeigeeigenschaft des Bildes verändert und das Bild zusammen mit dem Titel und Beschreibungstext vergrößert ausgegeben.

Fazit

Als Fazit können wir sagen, dass wir sehr viel Freude an der Arbeit des Projektes hatten und auch zufrieden mit dem Endergebnis sind. Es war das erste Mal, dass wir drei komplett verschiedene Teilbereiche hatten. Jedes Gruppenmitglied hatte seine eigene Aufgabe, jedoch wurde Teamarbeit und vor allem Kommunikation stark gefordert, denn alle einzelnen Projektabschnitte mussten zusammen integriert und angebunden werden.
Wegen der Natur unseres Projektes konnte sich jeder in seinem Teilgebiet verbessern, aber auch Einblicke der anderen Gebiete mitbekommen und sein Wissen erweitern.
Die Einblicke in die verschiedenen Arbeitsgebiete waren sehr interessant und es war ein gutes Gefühl nach längerer Arbeit ein funktionierendes Zusammenspiel zwischen den verschiedenen Arbeitsgebieten zu schaffen.