Universitatea Babeș-Bolyai

Facultatea de Matematică și Informatică

Specializarea Informatică în Limba Engleză

Lucrare de licență

“Social Media Manager – Instagrowth”

Student

Prisăcaru Antim

Coordonator

Lect. Dr. Mircea Ioan Gabriel

Cluj-Napoca, 2020

Contents

[1. Introduction 3](#_Toc43414059)

[2. Infrastructure 4](#_Toc43414060)

[2.1 Front-end 4](#_Toc43414061)

[NgRx State Management 5](#_Toc43414062)

[2.2 Back-end 6](#_Toc43414063)

[2.3 Dev-ops 7](#_Toc43414064)

[3. Main features 7](#_Toc43414065)

[3.1 Security, Authentication & Authorization 7](#_Toc43414066)

[3.2 User Settings 10](#_Toc43414067)

[Bibliography 11](#_Toc43414068)

# Introduction

In the context of world globalization and digitalization, more and more entrepreneurs wish to make the presence of their business known on the internet. They desire such, due to the need of intelligently attracting new clients, whether we talk about traditional on premise firm, or the 21st century notion of an “online business”.

The greatest issue with this is that the marketing has also evolved into an elaborate field, which generates immense revenue, as well as costs for those who wish to benefit from it. As such, the matter arises: how would a local company proceed to bite from this colossal corporation-absorbed trade? That said, the context entails administrators with little to no knowledge with marketing in general, let alone its subsidiary, the social media marketing, as well as no means to fund contracts with companies that specialise in such things.

The answer, in my opinion, comes as a simplified version of the downright baffling methods and tools used by a traditional marketing agency. It would also entail more usage of the social media, which is by all means the cheapest platform in which anybody can promote and generate sales.

“Instagrowth” is a client-server web application, built on several typescript-based frameworks, such as Angular, Apollo GraphQL, Express, etc. combined with micro services in order to deliver optimal performance. This application represents a user-friendly interface tool for people with all sorts of social media marketing experience, currently mainly targeting the Instagram platform.

Furthermore, this application will primarily serve as a post scheduler, with features such as performance meters, smart analytics, an image-recognition based algorithm that supports users in choosing hashtags that suit their imagery, and so on.

# Infrastructure

The following chapter will contain the description of this project’s infrastructure, structured on three parts: front-end, where the user interface will be presented; back-end, which contains the manipulation of data; and dev-ops, summing up the processes that will run behind the execution of the application.

# Front-end

**Angular** is a platform and framework for building single-page client applications using HTML and Typescript. Angular is written in Typescript. It implements core and optional functionality as a set of Typescript libraries that you import into your apps.



The architecture of an Angular application relies on certain fundamental concepts. The basic building blocks are NgModules, which provide a compilation context for components. NgModules collect related code into functional sets; an Angular app is defined by a set of NgModules. An app always has at least a root module that enables bootstrapping, and typically has many more feature modules.

* Components define views, which are sets of screen elements that Angular can choose among and modify according to your program logic and data.
* Components use services, which provide specific functionality not directly related to views. Service providers can be injected into components as dependencies, making your code modular, reusable, and efficient.

Both components and services are simply classes, with decorators that mark their type and provide metadata that tells Angular how to use them.

* The metadata for a component class associates it with a template that defines a view. A template combines ordinary HTML with Angular directives and binding markup that allow Angular to modify the HTML before rendering it for display.
* The metadata for a service class provides the information Angular needs to make it available to components through dependency injection (DI).

An app's components typically define many views, arranged hierarchically. Angular provides the Router service to help you define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.

## NgRx State Management

NgRx is a framework for building reactive applications in Angular. NgRx provides state management, isolation of side effects, entity collection management, router bindings, code generation, and developer tools that enhance developers experience when building many different types of applications.

NgRx provides state management for creating maintainable explicit applications, by storing single state and the use of actions in order to express state changes.

By normalizing state changes and passing them through observables, NgRx provides serializability and ensures state is predictably stored. This enables to save the state to an external storage, for example, localStorage. Type safety is promoted throughout the architecture with reliance on the TypeScript compiler for program correctness.

Using NgRx Effects and Store, any interaction with external resources side effects, like network requests, web socket and any business logic can be isolated from the UI. This isolation allows for more pure and simple components, and keep the single responsibility principle.

Because Store uses pure functions for changing state and selecting data from state, and the ability to isolate side effects from the UI, testing becomes very straightforward. Store is built on a single immutable data state, making change detection turn into a very easy task using an OnPush strategy. NgRx is also powered by memoized selector functions which optimize state query computations.

A good substance that might answer the question "Do I need NgRx", is the SHARI principle:

* **S**hared: state that is accessed by many components and services.
* **H**ydrated: state that is persisted and rehydrated from external storage.
* **A**vailable: state that needs to be available when re-entering routes.
* **R**etrieved: state that must be retrieved with a side-effect.
* **I**mpacted: state that is impacted by actions from other sources.

However, realizing that using NgRx comes with some tradeoffs is also crucial. It is not meant to be the shortest or quickest way to write code and encourage its users the usage of many files. It is also often require a steep learning curve, including some good understanding of RxJs and Redux.

# Back-end

**GraphQL** is an open-source data query and manipulation language for APIs, and a runtime for fulfilling queries with existing data. GraphQL was developed internally by Facebook in 2012 before being publicly released in 2015. On 7 November 2018, the GraphQL project was moved from Facebook to the newly-established GraphQL Foundation, hosted by the non-profit Linux Foundation. Since 2012, GraphQL's rise has followed the adoption timeline as set out by Lee Byron, GraphQL’s creator, with accuracy. Byron's goal is to make GraphQL omnipresent across web platforms.

It provides an approach to developing web APIs and has been compared and contrasted with REST and other web service architectures. It allows clients to define the structure of the data required, and the same structure of the data is returned from the server, therefore preventing excessively large amounts of data from being returned, but this has implications for how effective web caching of query results can be. The flexibility and richness of the query language also adds complexity that may not be worthwhile for simple APIs. It consists of a type system, query language and execution semantics, static validation, and type introspection.

**Express** is the most popular *Node* web framework, and is the underlying library for a number of other popular Node web frameworks. It provides mechanisms to:

* Write handlers for requests with different HTTP verbs at different URL paths (routes).
* Integrate with "view" rendering engines in order to generate responses by inserting data into templates.
* Set common web application settings like the port to use for connecting, and the location of templates that are used for rendering the response.
* Add additional request processing "middleware" at any point within the request handling pipeline.

While *Express* itself is fairly minimalist, developers have created compatible middleware packages to address almost any web development problem. There are libraries to work with cookies, sessions, user logins, URL parameters, POST data, security headers, and *many* more.

**Apollo Server** is a library that helps you connect a GraphQL schema to an HTTP server in Node. Apollo Server works with any GraphQL schema built with GraphQL.js, so you can build your schema with that directly or with a convenience library such as graphql-tools.

This server can be queried from any GraphQL client, since it supports all of the common semantics for sending GraphQL over HTTP. Apollo Server also supports some small extensions to the protocol, such as sending multiple GraphQL operations in one request. Read more on the sending requests page.

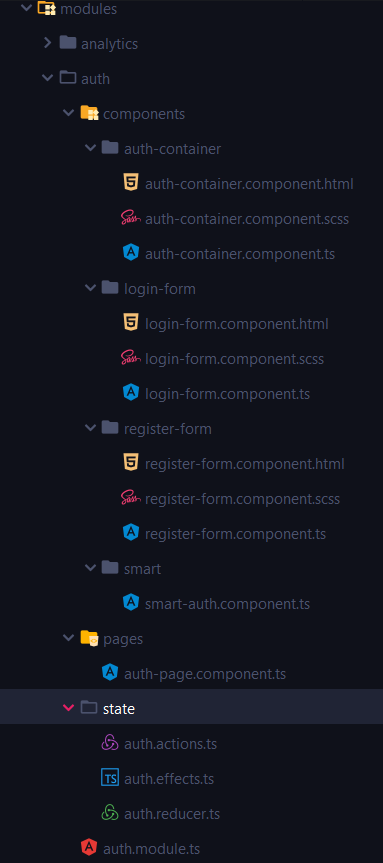
# Dev-ops

TBA.

# Main features

# Security, Authentication & Authorization

While the application will require its users to input their Instagram accounts, the access to the platform will not be made through that account. Instead, a local account will be created, which enables control over the functionalities that each user is able to access. This will later on allow for integration with a payment requirement, in order to access certain functionalities, such as the hashtag recommendations.



Thus, the authentication module bases off a user model, mainly defined by a unique ID, username, email address and password. The user can use both his username or email address in order to sign in, aside from the password that is mandatory in either case.

Architecturally speaking, the authentication module is, as the other front-end modules within the project, structured using the “smart and dumb components” design pattern. It dictates that all the logic has to be placed within a component that focuses on functionality. On the other hand, the view should be broken down in distinct, data-displaying only components that show input values, and emit events based on interaction of the user.

The flow begins with the page component, which is accessed directly by URL, which calls the smart component. Within, there is a flag that is further passed to the auth container, deciding whether the login or the registration forms will be shown.

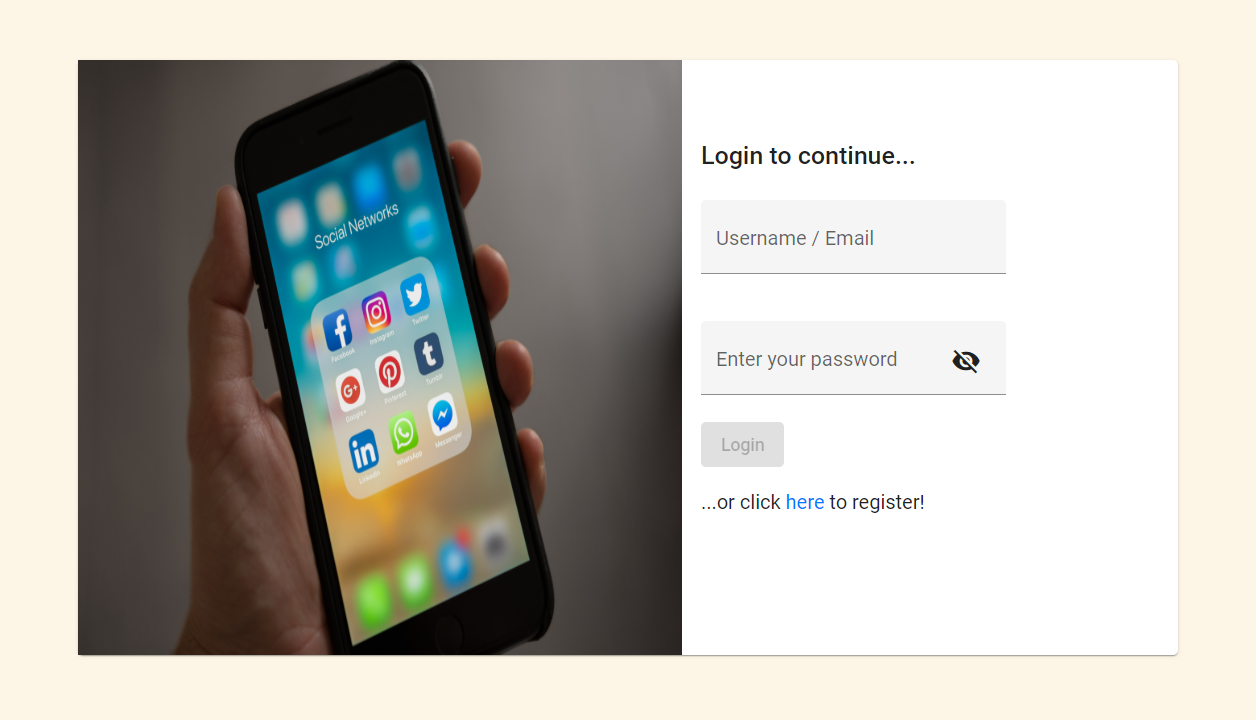
Also, a reactive forms component is initialized here, the FormGroup – a collection of fields that ought be displayed, with the possibility of adding a validator. For instance, login and password fields are mandatory, albeit there is a repeat password field, whose sole validation is that it matches the main password.



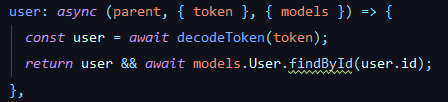
asdasdasda

State managing actions are also called within the smart component. Once the form is valid, and the login button is pressed, the “UserLogin” action will be dispatched, with its input data being the form values. Then, within its effect, the sign in function is called within the authentication service, which sends a mutation request towards the backend and expects a JSON Web Token (JWT) that has a 30 minute validity.

Thus, a promise containing the JWT is forwarded to the “UserLoginSuccess” action. In the effect, two actions take place: calling of another action that sets the token to the local storage, and another that redirects the user to the actual website.



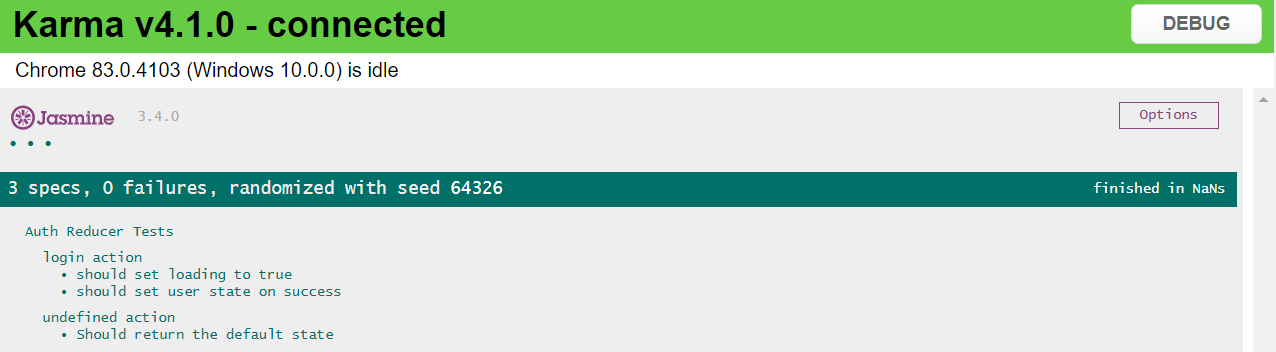
Once authenticated, the application will continuously check that the user is allowed to visit routes that belong to the core module. To do that, a guard is installed on all the respective routes that attempts to either take the user from the state, or from the API. In order to avoid sending a request for every route change that the user does, the guard will check first in the state, which only appears provided the page has not been refreshed. Otherwise, the token is sent to the server, which returns the user object.

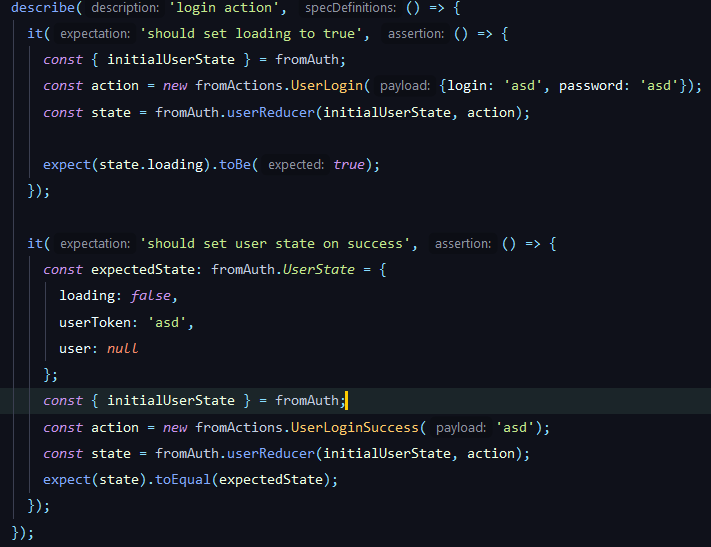


User query resolver on the back-end

Also, in order to avoid situations where the user can just return to the authentication page whilst logged in, there has been also implemented a login-guard functionality that protects those routes.

Finally, in regards to testing that above functionalities work properly, UNIT testing has been introduced for the NgRX part, which essentially does the better part of the functionality on the front-end. Reducer and effects tests are performed through Karma framework. Aspects such as the loading flag being set to true, or the user being set within the state by the reducer, are being checked in a deterministic manner.





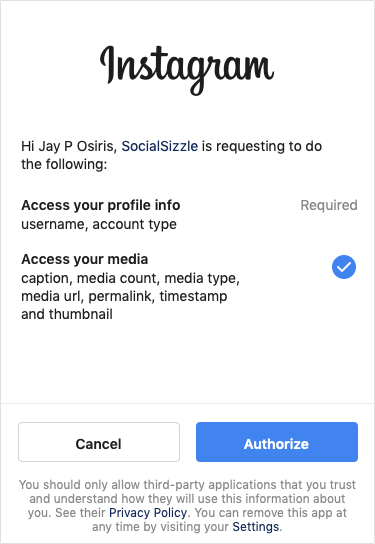
# User Settings

Another important module that directly regards the user administration is the settings module that encompasses a wide array of changes that everybody will be able to perform. Such changes include the basic fields (first name, last name, gender, etc.), or more important ones like the social media accounts, payment methods and so on.

Most of these properties of the user account will be, at this point, already in the state, meaning that there would not be any request in order to display them. Those details will be maintained in their own section, as to avoid interference with more essential features. Also, a worthwhile aspect of the implementation refers to the dual-state in which the details can be found. In order to enhance user experience, both states are shown on the same page, with the read-only fields turning into actual text boxes, dropdowns or checkboxes once the need for editing is prompted by the user.

The accounts section is a distinct feature that brings about a list of all the social media accounts that the user has linked to this platform’s account. Aside from the usernames, the user will find an “ACTIVE” or “INACTIVE” marking next to it, denoting whether the access token to that account is expired or not. This info will be provided by the backend by sending a request to Instagram for every account that has not been checked recently (defined by a constant in the application settings).

Should the user desire to reactivate an account, they would have the possibility to do so, by clicking the “INACTIVE” button, which will redirect them to the Instagram endpoint, prompting them to log in the application, and returning the refreshed access token to the GraphQL backend, where it will be stored for later use.



If the user wishes, however, to add a new account, they have the option to do so by pressing the “Add new” button above the accounts table. The same scenario will happen as described above, where the user is redirected to Instagram endpoint, prompting them to log in, with the backend registering the user ID and access token for this account.

The authorization window is provided by Instagram, and requires the user to have an active connection, as well as the agreement for data usage by the third party application (in this case - Instagrowth).

# Bibliography

<https://www.instagram.com/developer/>

<https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction>

<https://angular.io/guide/architecture>

<https://www.apollographql.com/docs/apollo-server/v1/>

<https://ngrx.io/docs>