UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Development of Frontend for a Podcast Hosting Platform with the use of Reactive Programming Paradigm

Author:
Roman BLAHUTA

Supervisor:
Pavlo BEREZIN

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2022

Declaration of Authorship

I, Roman BLAHUTA, declare that this thesis titled, "Development of Frontend for a Podcast Hosting Platform with the use of Reactive Programming Paradigm" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"The beautiful thing about podcasting is it's just talking. It can be funny, or it can be terrifying. It can be sweet. It can be obnoxious. It almost has no definitive form. In that sense it's one of the best ways to explore an idea."

Joe Rogan

"Podcasts free up, say, two hours a day for people to engage in educational activities that they wouldn't otherwise be able to engage in, and that's about one eighth of people's lives. So podcasts hand people one eighth of their life back to engage in high-level education."

Jordan Peterson

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Development of Frontend for a Podcast Hosting Platform with the use of Reactive Programming Paradigm

by Roman BLAHUTA

Abstract

Over the course of the last several years, both humanity's lifestyle and social trends have changed drastically. One of many phenomena that appeared as a result of both new social restrictions and new digital opportunities is the podcast genre's extreme rise to popularity.

There are two digital tools integral to creating podcasts: a streaming platform and a hosting service. While the market is flooding with many well crafted audio and video streaming services, there is a noticeable lack of podcast hosting services with well crafted UI, thought through UX and user flow, and lack of features to either affect a wider audience or to make the process of application's usage more comfortable and productive.

In order to solve these problems, a careful research should be conducted and there should be provided a new solution with both new nad reimagined old feature, a carefully designed structure and functionality. These efforts are necessary in order for this unique genre of content to thrive and serve people well, providing great entertainment and education.

Links:

- Project Repository:
  https://github.com/RomanBlahuta/Amphora-Podcast-Hosting
- Design File:
  https://www.figma.com/file/Kis5gzAQxMfTcePSfcCTRd/Podcast_Host...
- Application Architecture Diagram:
  https://drive.google.com/file/d/1PxZ6LGLTEEWHH6OWyCNiFJFr2ZxM...

Acknowledgements

First of all, I want to thank my thesis supervisor Mr. Pavlo Berezin for guiding me and providing advice about both technical moments and ideas how to organize the development and research in the most productive way. He was always available, initiative, honest and critical of my work in the best way possible.

I would also like to thank Denys Ivanenko, a member of our development team and a fellow student, whose thesis included the development of the Backend for this platform, and Mr. Joe Lindsley – a good friend from the United States of America, a professional journalist and the host of the "Speak Freely" podcast for providing important insights and allowing to personally experience a recording session.

I also want to express gratitude to the Flaticon website for providing free icons of high quality for designing and developing this podcast hosting application.

Contents

List of Figures

List of Tables

List of Abbreviations

HTML	**H**yper**t**ext **M**arkup **L**anguage
CSS	**C**ascading **S**tyle **S**heets
DOM	**D**ocument **O**bject **M**odel
SCSS	**S**assy **C**ascading **S**tyle **S**heets
SASS	**S**yntatically **A**wesome **S**tyle **S**heets
PWA	**P**rogressive **W**eb **A**pplication
MVP	**M**inimum **V**iable **P**roduct
CRUDL	**C**reate **R**ead **U**pdate **L**ist
GUI	**G**raphical **U**ser **I**nterface
UX	**U**ser E**x**perience
UI	**U**ser **I**nterface
URL	**U**niform **R**esource **L**ocator
DTO	**D**ata **T**ransfer **O**bject
RSS	**R**eally **S**imple **S**yndication
REST	**RE**presentational **S**tate **T**ransfer
API	**A**pplication **P**rogramming **I**nterface

In the loving memory of Yevhen Podoliuk and Orest Blahuta, my dear grandfathers.

Dedicated To my family: Roman Blahuta (father), Roksolyana Blahuta (mother), Roksolyana-Mariya Blahuta (sister), Mariya Blahuta (grandmother), Halyna Podoliuk (grandmother), Ihor Zhovtulia (godfather), Halyna Zhovtulia (aunt) for providing me with best possible education and opening endless opportunities in my life.

Chapter 1

Introduction

The recent years and the circumstances they brought upon the humanity have certainly changed the way of life of every single person on this planet. The lengthy and sudden COVID-19 pandemic, Ukrainian-Russian War and the global crisis they brought as a consequence made all of us adapt and come up with a new day-to-day routine, different way to handle our hobbies, work, education and social life.

These drastic changes influenced the development of new trends in all areas of our life. One of the results of our sudden change of lifestyle is the sudden growth of the podcast genre's popularity since it is a type of content that can be easily produced while following all the new restrictions and even more so the lengthy nature of this content format allows people to fill the suddenly great amount of free time with interesting and usually educational information while not preventing them from performing their daily tasks in parallel.

Consequently, any instruments and products meant to optimize the process of producing and consuming podcasts have been in especially huge demand for the last three to four years. Spotify, Google Podcasts and Apple Podcasts are only a few examples of successful podcast streaming platforms that brought a safe and comfortable experience to all podcast listeners.

Yet the market (especially in Ukraine) for digital tools meant for podcast hosting and management still has more than enough space for growth. Most of such services have an extreme lack in quality of their respective user interfaces both from the UI and UX perspectives, lack of automated features or could provide more functionality for their users' comfort.

1.1 Task Definition

It is important to outline the minimum viable functionality of the application for the purposes of podcast hosting:

1. Complete user system;
2. User dashboard page with content overview and general settings;
3. Content management:
    (a) Complete Podcast (or Show) CRUDL: Create form, edit form, delete, add episode;

   (b) Complete Episode CRUDL: Create an episode and add it to a show, remove an episode, edit episode data;

4. Streaming integration with most major listening platforms and generation of a RSS streaming feed;

The goals of this thesis can be listed as follows:

1. Conduct a thorough research of the local and global market for digital podcast hosting and distributing solutions;
2. Conduct research and interviews with the target audience for this service;
3. Provide a solution to common UI/UX problems of existing solutions;
4. Increase service's Frontend productivity and decrease resource consumption;
5. Create a UI that keeps the information up-to-date as soon as the changes arrive;
6. Implement MVP version of the application;
7. Implement features that provide a rich functionality and improve the listed downfalls of alternative services;

The application flow should be designed in a way that it does not require:

1. Unnecessary efforts from the user to navigate and use the application;
2. Lengthy investigation for the newcomers to the podcasting industry;
3. Additional efforts to see the result of their actions displayed on the GUI in real time (page reload, extra navigation etc.);

The application should provide:

1. An intuitive navigation system;
2. Comfortable user flow;
3. Always up-to-date data to display, as soon as the changes arrive;
4. Acceptable loading times;
5. Up-to-date UI/UX design;
6. Hight customizability for content;
7. Well structured data display;

1.2 Methodology

1.2.1 Research

The research necessary for defining the application's functionality was conducted in the following way:

1. Interviews with podcast hosts, investigating videos and articles about podcast hosting for further insight into the field;

2. Investigation of functionality of podcast streaming services;
3. Investigation of the existing podcast hosting services (including hands-on experience), their strengths and weaknesses, moments that can be improved and processes that can be optimized;
4. Composing general plan of the application;
5. Reviewing the initial application flow structure and functionality with podcast hosts;

1.2.2 Design

1. Investigation of free resources for GUI elements design;
2. Analysis and testing of the existing alternative services' UX/UI design, their applications' navigation tree and user flow planning;
3. First design release and review of the user flow;
4. Iterative improvement cycles;

1.2.3 Development

1. Choosing technologies for development stack;
2. Project set up, configuring linter, redux devtools and other additional instruments for development;
3. Architecture planning and setup;
4. Development of UI pages and components;
5. Utility code implementation: error handling, route guards, local storage wrapper etc.;
6. View logic implementation: adding state, callbacks, http requests etc.;
7. Final refactor and code clean-up;

1.2.4 Testing

1. Development Testing: during the course of system development, numerous integration and system testing iterations were performed by the development team.
2. Testing by associates: additional test were performed by our associates, who are not a part of the podcasting community, but were familiarized with the flow of this application and its purpose. The tests were performed to determine the quality of the UI markdown and the overall User Experience. Testers were also involved in basic bug and error detection.
3. Testing by beta-version users: after the development of a significant part of the application was done, we ran several tests with local podcast hosts in order to determine the product quality on an industrial level.

1.3 Contributions

1. To our knowledge, this will be the first podcast hosting service on the Ukrainian market;
2. Improved user flow and UI/UX for podcast hosting applications;
3. Record-and-host feature right inside the application;
4. Fast and easy integration with the major media streaming services;
5. Reimagined structure of data orgnaization for podcasts (Shows), episodes and episode subsets (series) in a podcast;
6. Simplified user flow and automated functionality;
7. Ability to integrate with most popular streaming services with little effort;

1.4 Thesis Structure Overview

1.4.1 Chapter 2: Research

In this chapter, there are present problem formulation, details of the research process and its results, currently existing solutions and the solution methods that were considered during the research phase of the thesis. These methods are compared with each other in detail.

1.4.2 Chapter 3: Proposed Solution

This chapter contains the chosen solution option for this thesis, the methodology of the solution is described. This part of the thesis also goes into the technical details of the application, its architecture and overall implementation, and the restrictions imposed on the development process.

1.4.3 Chapter 4: Perspective

This chapter describes the perspectives and future development of this project, listing ideas for improvement or additional features which did not make it into the current scope for a number of reasons which will be also stated in this part of the thesis.

1.4.4 Chapter 5: Conclusions

Chapter 5 will be the ending chapter of the thesis, containing the main points of my project and research, their current and future results.

Chapter 2

Research

2.1 Problem

In order to create podcasts, there are two essential tools that the content-makers need: a streaming platform and a hosting service. The author needs to provide the streaming platform with a RSS feed, that will pass the data of their podcast (audio, cover image, podcast details such as category, description etc.) to the streaming platform.

While the market for audio streaming platforms has been overrun with tons of carefully crafted services, the amount of well-made hosting platforms is surprisingly low. Many of these services are severely outdated, both in terms of logic and UI/UX.

2.2 Market Research

Since the topic of this thesis is developing an improved UI for such platforms, a corresponding research was conducted to outline the usual flaws in current solutions for the outlined topic.

2.2.1 Anchor by Spotify

Anchor is a mobile application developed by the Spotify team. It includes tools to both record a podcast on your mobile device and publish it to the audio streaming platforms. It also integrates ads and podcast hosting features in addition to the onse listed above. It is a very simple app for beginners with a reliable implementation. Yet still, this application has several major flaws that should be pointed out:

1. All podcasts are published on Anchor's profile instead of the author's;
2. This service lacks configurability and detailed customization of content (and UI for the corresponding purposes);
3. The application lacks integration with less known or local streaming platforms;

2.2.2 Podbean

Podbean is a web service that provides podcast hosting and integration with major streaming platforms. Podbean also offers their own monetization platform, even though it is limited by a pricing plan. The main issues with the frontend side of the applications are slightly outdated UI/UX and weird user flow. Here is an example: after signing up, the user is sent to their dashboard, but for some unknown reason,

just right after being authorized, if the user decides to create an episode, the application promts them to authorize once again and completely blocks and disrupts the user flow. The person using this application gets sent back to the signing in form and has to enter their credentials once more. Such problems have been noticed several times during the investigation of this service and could definitely be improved upon. The main problems of this platform can be listed as follows:

1. Inconsistent and non-intuitive user flow;
2. Outdated UI/UX design;
3. Limited possibilites of monetization;

2.2.3 Buzzsprout

A podcast hosting service with a formidable UI and serious statistics functionality. While definitely being a great choice for hosting your podcast there is still place to grow. The main functionality pages slightly lack in UI design (especially in comparison to their amazing landing page), the functionality is rather simplistic. While it is a big advantage for beginners, there should definitely be some advanced features for those who take their podcast content seriously. Also while providing functionality for directories, the episodes and other content can be structured and displayed in a better way: search system, advanced filters, splitting content into categories etc. The following main issues were outlined during the investigation of this service:

1. Outdated UI design;
2. Low content customizability;
3. Confusing content management: lack of proper search and filters, non-intuitive directories functionality;

2.2.4 Libsyn

Another service for podcast hosting and distribution. Overall the platform is well-made and has a wide range of offers such as integration with several popular streaming platforms, monetization and statistics. Yet some issues still came up after a deeper investigation - the loading times on this particular web application are somewhat long and disrupt the flow with noticeable pauses.

1. Lengthy loading times;
2. Performance issues;
3. Full page rebuilds;

The main conclusions of this research about problems with existing solutions can be listed as follows:

1. Lengthy loading times;
2. Constant necessity for page reload;
3. Lack of structure in the view of content, complicated directory configurations;
4. Outdated design;
5. Lack of customizability of content, lack of detailed settings;
6. Chaotic user flow;

2.3 Technical Research

After the conducted research the following technologies were chosen in order to develop "Amphora":

- Angular – a TypeScript-based free and open-source framework for single-page web applications. The other option that was considered is React.JS but there are several reasons why Angular is an objectively better choice of a framework, especially for this project:
  - Angular imposes a very well composed architecture, that is easily scalable, well organized, and improved;
  - Angular uses TypeScript by default which introduces numerous useful development features such as static typing, enumerations and interfaces etc.;
  - Angular provides the possibility to use reactive programming straight out-of-the-box. It includes the RxJS reactive programming library by default and many parts of the framework heavily rely on it;
  - Lazy loading for higher performance;
  - Angular provides a rich UI component library;
  - Angular provides the possibility to use CSS preprocessors right out-of-the-box, providing three stylesheet options on the project set up stage: CSS, SASS and SCSS. CSS Preprocessors enrich the default stylesheet syntax with a wide variety of features which minimize code duplication, unnecessary repetitions of prefixed classnames, and of course features such as reusable pieces of style such as mixins, extending one class with another etc.;
  - Angular applications' single page nature has numerous advantages, such as low band width use which allows the app to work well even with a slow internet connection, quick loading time, SPA makes it easy to add advanced features to a web application (content editing web app with real-time analysis. Doing this with a traditional web app requires a total page reload to perform content analysis). Using a SPA approach for this projectis a big advantage, because, as stated in the previous section of this chapter, some applications really suffer from lengthy loading times and constant need to reload the page;
  - Built-in form validation – Angular includes FormsModule and ReactiveFormsModule by default, while other frameworks typically rely on additional packages;
  - Easily-programmed brower animations – the framework provides an easy method of decorating components with numerous animations to provide a prettier user experience;
  - Angular integrates extremely well with another technology – Ionic Framework. Angular was the initial base framework for use with Ionic and provides numerous possibilities to expand the web application into a PWA and even a web-view mobile application.
  - While React.JS also has access to some of these features, none of them come preinstalled during set up and require a lot of additional effort to integrate into the project. React.JS's default disorderly nature and lack of

architecture imposes a lot of risks during the beginning of the development, especially when planning the project's structure and dataflow.

- Ionic Framework – a complete open-source SDK for hybrid mobile and progressive web app development. It was included for its useful component library and the perspective of expanding the project into other fields of programming without necessity to create a completely new project;
- NgRx – a state management library for Angular which uses the Redux architecture and utilizes the reactive programming paradigm. It provides simple and quick dataflow within the application and allows the user to see changes on their screen in real time, as soon as the change is made. The reactive approach and simple structure had the final say when choosing this state management method instead of any alternatives;
- NgRx/Effects - an additional library for the beforementioned Redux-inspired reactive state management solution for Angular. This library adds an additional layer of logic to the classical Redux approach consisting of actions, reducers and selectors (or alternative ways of reading a needed value). This layer is the store Effects - a sort of middleware that performs additional logic after an action is dispatched and before it enters the reducer. The usage of Effects is best described in the corresponding chapters of Farhi, 2017: *"I like to think of ngrx/effects as a layer that groups several actions as a specific chain of reactions. This leads to organize the code in such a way that the logics for side effects are placed in a dedicated directory- "effects" - while services are managed separatley in the "services" directory."*
- RxJS – a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code. It provides better performance, better modularity and better debuggable call stacks;
- CryptoJS – a JavaScript library for encryption of sensitive data such as access tokens and client secrets etc.;

Angular and its single page approach should solve the problem of unnecessary consuming of extra resources. There will be no lengthy loading times during navigation, entire page rebuilds and many other common flaws of traditional websites: *"Every web developer knows how problematic is page navigation in a web application, besides of bandwidth wasting and process time rebuilding entire pages more problems make web development painful like unwanted caching, back/forward buttons, desynchronized forms caused by the "form auto-fill" feature of some browsers and so on."* Santamaria, 2015.

Another important point of this technical research is how advantageous the reactive approach really is. Using RxJS for handling all of the asynchronous tasks of the proposed application is a great alternative to common Promise and callback based tactics and other ways of handling requests, browser events etc. Reactive Extensions library has a number of advantages that are well described in great detail in the **1.3** and **1.4** subchapters of Daniels, 2017.

In short, RxJS treats all its asynchronous tasks as data streams called Observables, thus unifying both the way of developer's thinking and the coding approach inside the program.

"Reactive Extensions for JavaScript (RxJS) is an elegant replacement for callback or Promise-based libraries, using a single programming model that treats any ubiquitous source of events—whether it be reading a file, making an HTTP call, clicking a button, or moving the mouse—in the exact same manner. For example, instead of handling each mouse event independently with a call back, with RxJS you handle all of them combined." Daniels, 2017

This library provides easily readable and testable ways of managing complex asynchronous work, by organazing it in a linear way using a wide range of functions called operators. This is an amazing alternative to the usual nested callbacks that turn the task of trying to test and understand a certain part of asynchronous code into an entire journey.

RxJS also provides great performance due to a large number of optimizations contributed to this project by a massive community of open-source developers across the world. RxJS has a well organized memory leak management, easy methods of unsubscribing from unnecessary streams to lower resource consumption and many more noticeable advantages that easily convince one to use this library that is so rich in functionality.

Chapter 3

Proposed Solution

3.1 Idea

The solution to the outlined problems is the creation of a new podcast hosting platform titled "Amphora". This application's goal is to provide the users with as many automated functions as possible. The application will feature the MVP functionality and many more features, listed in the following enumeration:

1. Email verification for extra security;
2. An intuitive password renewal process;
3. Detailed content customizability:
   (a) Create form, edit form, delete, add episode, upload show cover and add or remove Series (subcategory tag that groups a set of show's episodes by a common theme);
   (b) Create an episode and add it to a show, remove an episode, edit episode data, add an episode to a series, upload episode cover, upload audio-file or record it on-the-fly inside the app;
4. In-app recording functionality for express podcasts;
5. Structurization of data by splitting the episodes between shows and series;
6. UI with modern design and up-to-date data display with no need to reload the page;
7. Acceptable loading times for the application and data inside it;
8. Search system and display filters;
9. Intuitive and consistent user flow inside the application;
10. Select streaming options for a show with a single click and get the integration link to a RSS-feed file that needs to be inputted into the streaming service. This process will be automated with those services that support this feature;
11. User-friendly admin panels;
12. A mailing system - sending notifications, confirmation letters etc.;
13. Fully animated pop-up system;
14. Built-in media player;
15. Reactive data display and state management;

16. Formidable optimization of productivity;

The following features will significantly improve the UX while providing a rich set of features just like existing services and even more. The application will be comfortable to use for both newcomers and veterans of the podcasting community. The application and its features will be described in greater detail in the next section of this chapter.

3.2 Implementation

The implemented application provides a wide range of useful features for its users. Many of those features are ment to solve the beforementioned problems of alternative services, improve the use experience and make sure the users can manage their podcasts with ease and comfort.

3.2.1 Reimagined Data Management Design

This application features a major improvement - the episodes that the user uploads are automatically associated with more general entities: Shows and Series. This is a pretty intuitive and convenient way to approach the potentially massive amount of content that the user may produce. A Show or a Podcast obviously features episodes. Those episodes inside a show can be a part of a smaller content group called Series. This way, the user can easily filter out all unnecessary data if they have a significant amount of shows that they produce. Not only does this data organization help the user navigate their content, it also requires almost no effort to set up, unlike the complicated, unautomated and non-straightforward functionality of custom directories of alternative services.

As it was stated, the application features three main entities that organize the data and compose the main application flow: Shows, Episodes and Series. All of those entities have a complete set of operations such as creation, editing, deletion and view. The corresponding views are provided on the pages of the application with detailed data display views, creation and editing forms. All parts of this functionality possess a reactive data flow in order to provide a great responsive experience, always keeping the user up-to-date with their content information.

In order to effectively utilize the aforementioned data organization, several additional features were added to the application. While viewing their content the user can filter out unnecessary data by searching the wanted piece of content via entering its title in the search bar. The user can filter out unwanted episodes using Series, title, episode and season number.

All of the previously mentioned creation and edit forms provide detailed customizabilty to the user's content, ranging from basic properties such as title, description and image cover, to more specific data such as episode type, season and episode numbers, adding series, explicit content warning etc.

All of the content submitted by the user can be integrated with several major streaming platforms with very little effort. "Amphora" ensures that the listeners will be able to enjoy their favorite content, no matter what their preferred platform is.

3.2.2 User-Friendly UI/UX

The application features a well thought-through UI design in order to provide a great user experience while using the application. "Amphora's" navigation is intuitive and the most important options are always present in the header of the page. All main flows do not require a lot of actions and all functionality entry points are positioned and labeled in a way so the user never gets lost. The UI pays attention to detail and incorporates elements of Ancient Greek thematic elements to make the UI design pleasant and interesting, so the users are aesthetically pleased and entertained while traversing "Amphora".

An application that strives to build a big community of users definitely needs a set of administrators to manage their clients' needs. These people also need their own UI in order to perform these important tasks and further improve the UX for "Amphora's" users while not lacking in great user experience themselves.

3.2.3 In-App Audio Recording

Another unique and useful feature that this app possesses is the ability to record a podcast right in the application. When the user creates an episode, they have to input a source file for their content. If the user wants to record a bonus episode or they prefer an express format of content, the ability to record outside of their studio where all of their usual equipment is located can be extremely useful. The user can effortlessly record their content and immediately submit it and start hosting it with the help of "Amphora", without ever exiting the application.

3.2.4 Improved Performance and Optimization

Due to a careful choice of technologies and tools for development of this platform, the productivity of "Amphora" is noticeably greater. On the Frontend side, major advantages of Angular have significantly boosted the application's performance. Single page applications consume less bandwidth, load significantly quicker than traditional web applications and provide a more seamless user experience while also removing unnecessary waiting time while navigating from one route to another. All data is fetched asynchronously and delivered to the UI via Observable data streams. The Backend part of "Amphora" also largely benefits from both its choice of technologies used and architectural design. The application's backend is built with FastAPI which provides great performance while offering a wide array of functionality. All input-output operations on the Backend are performed asynchronously which noticeably improves the performance.

Another important benefit from "Amphora's" technical implementation is its always up-to-date display of information. As soon as the user performs a corresponding action in the application, the state management system implemented through the NgRx library reacts to it and applies the changes to the UI, providing an incredibly responsive experience for the user. The Store also fetches and posts data with the help of Effects and provides the user with all of their information without the need to reload the page.

3.3 Architecture

The app utilizes the default Angular architectural approach while also expanding it with several methods.

3.3.1 Architecture of Angular Applications

The architecture of Angular applications consists of the building blocks described in the Table 3.1;

TABLE 3.1: Architecture of an Angular Application

Architecture Part	Purpose	Interacts With
Module	Declares a compilation context for components, associates them with related code, services etc.	Components, Pages, Directives, Modules, Pipes
Component	Defines a class that contains application data and logic, and is associated with an HTML template that defines a view to be displayed in a target environment.	Components, Pages, Directives, Modules, Pipes, HTML templates, Stylesheets
View/Page	Sets of screen elements that Angular can choose among and modify according to your program logic and data	Components, Pages, Directives, Modules, Pipes, HTML templates, Stylesheets
Service	typically a class with a narrow, well-defined purpose. Services separate any additional logic from component's view logic for better ogranization and reusablity. Services a re provided into components, pages and other services via Dependency Injection	Pages, Compontents, Services, Modules
Directive	Classes that add additional behavior to elements in your Angular applications or change the DOM layout	Components, HTML templates, Styles
Pipe	Simple function to use in template expressions to accept an input value and return a transformed value	Data
Router	Enables navigation from one view to the next. Defines the Route object that maps a URL path to a component, and the RouterOutlet directive that you use to place a routed view in a template, as well as a complete API for configuring, querying, and controlling the router state.	Routing Modules, Components, Services, Pages

The relationship between Angular's architecture building blocks can be expressed in the following diagram, taken from the official Angular documentation by Google LLC, 2022:

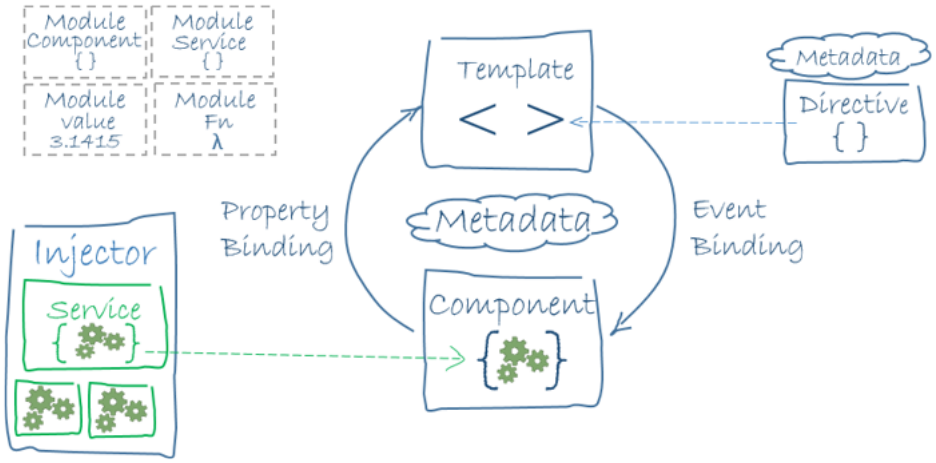

FIGURE 3.1: Angular Application Architecture

In short, module loads all declared elements, the page's view itself is a component consisting of other tags and components. Angular Injects a Service class into a component in order to provide its logic in a reusable way. A component and its logic interact with its HTML template through the following mechanisms: property binding - providing data from component class' properties to the template, and event binding - receiving user-initiated events from the template and handling them with their respective callbacks. The template can be modified with a directive, which either adds additional behaviours to the UI element or modifies DOM's structure. Each of these building blocks has their own respective metadata specified in decorators (@Injectable(), @Component() etc.) so Angular knows the specific purpose and behaviour of the class.

3.3.2 Additional Architectural Approaches

The application uses a modular architecture, meaning each page and component is contained in a seperate module, which can be optionally included in a certain part of the web application. This reduces the amount of recources that would be unnecessarily loaded, structures the application in a clear and concise way, and allows lazy-loading - one of Angular's important features. Lazy loading helps keep initial bundle sizes smaller, which in turn helps decrease load times.The project also includes many interfaces, enumerations and other utility code.

"Amphora's" architecture has several other elements that should be mentioned. A detailed description is provided in Table 3.2.

TABLE 3.2: Details of "Amphora's" architerctural approaches

Architecture Part	Purpose	Interacts With
Model	A configuration class for UI components. Provides a unified way to transport all necessary attributes to a component without passing them seperately causing the markdown to look extremely complicated. All changes can simply be applied to a class instance created in a Page Service	Components, Services
HTTP Service	Provides a wrapper class for Angular's HttpClient Service and contains all requests needed for a certain part of the application	State
State	Consists of actions, reducer, selectors and effects. Provides a single easily-manageable data source for the entire application	Services, Pages, Models
Page Service	A regular service that provides component model creation logic for a specific page of the application	Modules, Pages, State, Models, Components

3.3.3 State Management and the Reactive Approach

State management is implemented using the NgRx library. There are four main NgRx Store parts of the state management part of the application. See Table 3.3 for further details.

TABLE 3.3: State Management

State management	Purpose	Interacts With
Action	Triggers the reducer and effects in order to modify state or perform side tasks	Reducer, Effects, Used in callback and utility functions/methods
Reducer	Defines state in its initial and changed forms, changes the state according to actions and their properties	Actions
Selector	Provides an Observable stream of data notifying changes to state, a part of it or produces values based on the state and its properties	Services, Pages, Effects
Effect	Middleware between the action and the reducer, allows to execute additional code before the action enters the reducer (triggering HTTP requests, displaying popups etc.)	Actions, Selectors, Services

The NgRx Store heavily relies on RxJS and its Observables - a Producer of multiple values, "pushing" them to Observers (Consumers). Each Observable can be subscribed to and processed with a callback function. The advantage of using Observables and RxJS is that each time a value that is used in the template is produced, the UI is immediately updated. It allows the GUI to show the changes to the user in realtime and provides great way to manage those values. In addition to RxJS's stream-like Observables and Subjects (Similar to observable, but allows multiple subscriptions to a single stream), the library provides a rich variety of operators to process, combine and alter those data streams in a easily manageable way. This allows the developer to organize complex asynchronous tasks in a simple and readable way, avoiding callback hell etc. In order to use an Observable value in the UI, the only thing we have to do is use the "async" pipe on it in the component's template like this: <h1> model.title | async </h1>. Or alternatively subscribe to the Observable in the component's code and set the emitted value to a property each time. Then we use property binding and display that data on the UI.

As it was established earlier in this thesis, the NgRx Store is based on Observable data streams. Here are a couple examples of use of reactive programming in the application's state management:

- When we want to retrieve a value from state, we use a selector. The selector returns an Observable value with a generic type of whatever value you wish to select, then you can optionally alter the Observable value with the use of reactive operators such as map, filter etc. Then you can extract the value as described in paragraphs above and receive an always up-to-date value with each change straight to the UI. Here is an example of such case:

```
export const selectUserState = createFeatureSelector<fromUser.IState>(
    fromUser.userFeatureKey,
);

export namespace UserSelectors {
    export const selectEmail = createSelector(
        selectUserState,
        (state) => state.email,
    );
    // ...
```

LISTING 3.1: Selector code example

```
public userEmail: Observable<string>;
    // some component code...

    this.userEmail = this.store$.select(UserSelectors.selectEmail());
```

LISTING 3.2: Selector call example

```
<div class="dashboard__section-profile-data">
    ...
    <h3 class="dashboard__section-profile-data-email font-profile-description">{{ (userEmail | async
        ) || '' }}</h3>
    ...
</div>
```

LISTING 3.3: Selector usage in template example

This is the simplest example of selecting a value. In the examples you can also notice feature selectors and feature keys, which are also an important part of state management. A feature key defines a string that is going to be a name of the property on the big Store state management object, to which a piece of state

corresponds. The overall application state is a big object that has nested objects which correspond to a state of a part of the program. For example one of such states can be a sign in page state with a feature key "signIn". When we want to give a state base for a selector, so it knows which part of the application has the properties to select in current use case, we define a feature selector, which is provided with the feature key.

- In order to trigger some side effects to an action being dispatched, we use NgRx Effects. This is a class with Observable properties which take an Observable stream of actions, and modify the input Observable through the use of reactive operators. The usual structure of an Effect inside this application can be described in the following way:
  - Take the actions Observable and use a .pipe() function on it.
  - Pipe the ofType() operator with an action which will trigger the effect as an argument
  - If necessary, call withLatestFrom() or similar operators with a selector as an argument to get extra data from state
  - Perform necessary tasks either by using a tap() operator or using other operators such as switchMap() (On each emission the previous inner observable (the result of the function you supplied) is cancelled and the new observable is subscribed.) for triggering http requests with Observable responses etc.
  - Perform any other alterations and mappings if necessary with RxJS operators
  - Configure the effect (should the action trigerring the effect be dispatched to the reducer etc.)

3.3.4 Additional Use Cases for Reactive Programming

Most of Observable values are used inside component models as a sort of value controllers for inputs, paginations and other interactive UI elements. Additional use cases of reactive programming are such as emitting a value after an event has ocurred to alter the icon through an Observable model etc. Reactive approach allows great interactivity with a readable and relatively simple implementation.

```
1  <amphora-icon [model]="profilePictureModel"></amphora-icon>
```

LISTING 3.4: Component Model Usage Example

```
1  public profilePictureModel: AmphoraIconModel;
2      // Component code...
3      this.profilePictureModel = this.dashBoardService.createProfilePicture();
4      // ...
```

LISTING 3.5: Component model instantiation Example

```
export interface IOptional {
    inputType?: InputFieldTypesEnum;
    onInputListener?: (value: string, inputModel: AmphoraInputFieldModel) => void;
    size?: ISize;
    placeholder?: string;
}

export class AmphoraInputFieldModel {
    public value$: Observable<string>;
    public optional: IOptional;
    public valid: boolean;
    public disabled: boolean;

    constructor(value$: Observable<string>, optional?: IOptional) {
        this.value$ = value$;
        this.valid = true;
        this.disabled = false;

        this.optional = {
            onInputListener: optional?.onInputListener || undefined,
            inputType: optional?.inputType || InputFieldTypesEnum.TEXT,
            placeholder: optional?.placeholder || '',
            size: {
                width: optional?.size?.width || 480,
                widthUnit: optional?.size?.widthUnit || UnitsOfMeasurementEnum.PX,
                height: optional?.size?.height || 56,
                heightUnit: optional?.size?.heightUnit || UnitsOfMeasurementEnum.PX,
                widthDiff: optional?.size.widthDiff || 0,
                heightDiff: optional?.size.heightDiff || 0,
                widthDiffUnit: optional?.size.widthDiffUnit || UnitsOfMeasurementEnum.PX,
                heightDiffUnit: optional?.size.heightDiffUnit || UnitsOfMeasurementEnum.PX,
            },
        };
    }

    public static create(value$: Observable<string>, optional?: IOptional): AmphoraInputFieldModel {
        return new AmphoraInputFieldModel(value$, optional);
    }
}
```

LISTING 3.6: Component Model Example

3.3.5 Communication with Backend

Another use case of reactive programming inside the application is the implementation of communication with the server. "Amphora" uses a backend with a REST API implemented using FastAPI and Python.

All HTTP requests are grouped inside HTTP Services, each corresponding to the part of application logic that needs the beforementioned requests.

All requests are called via a method of a HTTP Service and the response is returned wrapped into a one-time firing Observable with a generic type of a corresponding DTO interface. DTO interfaces are present for each request payload and response for every request called within the application. The requests are initiated by dispatching an action into the NgRx Store, which in its place triggers an effect which calls one of the methods of a HTTP Service. Then the response Observable is mapped into a different action which dispatches the received data into the corresponding part of the application state. If an error occurres, an error pop-up will be shown.

After that, all necessary data is already present in the application's state and the components render correspondingly to the data which they receive through the state selectors.

3.3.6 Architecture Diagram

The diagram below is a visualization of "Amphora's" architecture, its building blocks and their relationships. The diagram portrays the architecture on an example of the flow of Sign In page. The rest of the pages and other parts of the application follow the same principles of architectural design.

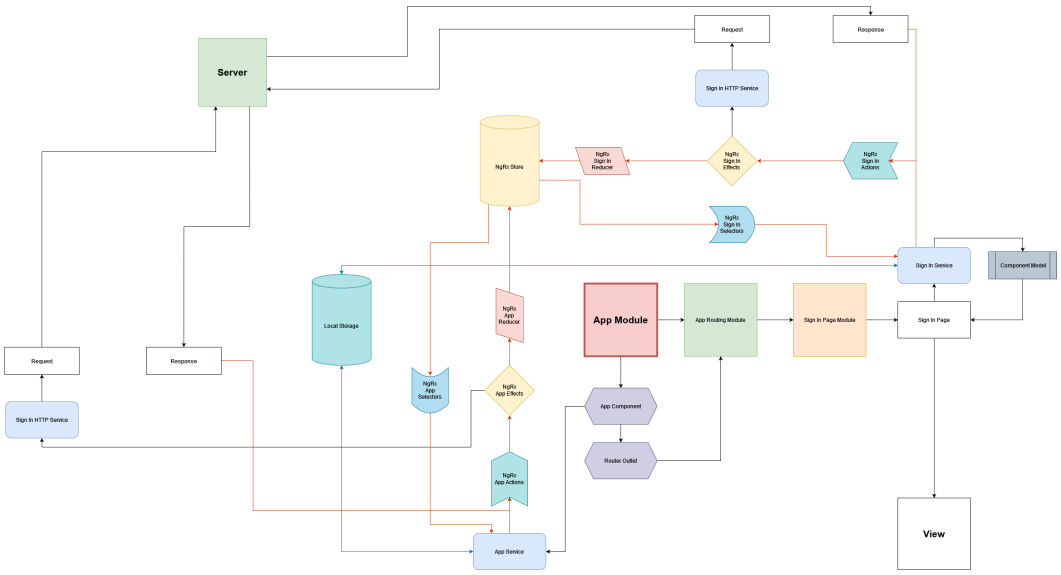

FIGURE 3.2: "Amphora" Podcast Hosting Platform Frontend Architecture

In short, the root module of this application is called App Module. The launch starts with bootstrapping the root module and rendering the root component - App Component. The root component has an Application service injected into it. This service provides necessary logic to properly initialize the application. App Component also contains a Router Outlet which provides all further navigation and displays a page corresponding to the current URL.

All routing is described in the so-called Routing Modules. App Routing Module has a defined set of routes with corresponding lazy-loaded NgModules that contain the page that will be displayed while on this route. In this case we are talking about the Sign In Module and the Sign In Page.

The page contains its markdown of components and elements in the HTML template and an injected Sign In page service that instantiates all Component Models. After instantiation they are stored in page's properties and get passed down to the corresponding Components. The page is displayed for the user to view it and to interact with it.

Another role of the beforementioned services is providing communication with the application's data storages: mainly NgRx Store and in several occasions the Local Storage. The communication with the Store is implemented via Actions to dispatch values into the Reducer that will alter the state object, and Selectors that will return

an Observable stream of data from the needed part of state. Actions also play an important role in the communication with Backend. While getting dispatched to the Reducer, that also may trigger NgRx Effects that will send a request from a HTTP Service. The response is mapped to a corresponding Action and is dispatched to the Store.

3.4 User Flow

Below there will be short descriptions of user's experience while navigating through the app to achieve a couple of all of the main goals.

User's navigation while using the application can be described in the following way:

1. User enters the application and is directed to the Landing Page
2. From here, user can choose either "Sign In" or "Sign Up" options in the header.
3. Signing Up:
   (a) User enters their data: First name, Last name, Email and Password.;
   (b) The user sees a pop-up with a prompt to check his email for a confirmation letter;
   (c) The user copies a confirmation token;
   (d) The user closes the pop-up or presses "Resend letter" button if he didn't receive one;
   (e) The user is directed to a form with an input field for the token;
   (f) If verification was successful, the user is redirected to the Sign In page, or sees an error otherwise;
4. Signing In:
   (a) The user enters their credentials: Email and Password;
   (b) If signing in was successful the user can enter the application and is redirected to the Dashboard page. If the credentials are invalid, an error pop-up is shown;
   (c) If the user does not remember their password then they can click a "Forgot Password?" link and be redirected to the Forgot Password page;
   (d) The user enters their email;
   (e) The user sees a pop-up with a prompt to check his email for a confirmation letter;
   (f) The user copies a confirmation token;
   (g) The user enters the token and a new password. If all data is correct the user will reset their password and be redirected to the Sign In Page. An error pop-up will be shown otherwise;

After the user is authorized, they can navigate the app and utilize the main functionality. The only exception is if the user skipped the verification stage during signing up. Unverified users cannot create any shows or episodes. The user can verify their email from their dashboard.

Next, we will look into the Shows functionality flow:

1. Creation
    (a) On the dashboard, the user can see paginated previews of all their shows.
    (b) Near the preview heading there is a "New" button
    (c) Clicking it will navigate the user to a show creation form
    (d) After filling all of the input fields, uploading a cover image and optionally adding Series to the Show, the user can submit their data and a show will be created.
2. View
    (a) On the dashboard, the user can see paginated previews of all their shows.
    (b) If the user hovers the cursor over a preview card, a "See More" button will be shown.
    (c) Clicking on this button will navigate the user to a show's full view page.
3. Deletion
    (a) The user can delete the show by clicking the "Delete" button on show's view page.
    (b) The user will see a prompt in a pop-up asking them if they are sure about deleting this item.
    (c) From here the user can either confirm or cancel the deletion by clicking corresponding buttons on the pop-up.
4. Edit
    (a) The user can edit the show by clicking the "Edit" button on show's view page.
    (b) The user will be redirected to a pre-filled form with show data.
    (c) After the user edits all necessary fields, they can submit the form and the show will be updated.

Now let's look into the Episodes functionality flow:

1. Creation
    (a) On the dashboard, the user can see paginated previews of all their shows.
    (b) If the user hovers the cursor over a preview card, a "See More" button will be shown.
    (c) Clicking on this button will navigate the user to a show's full view page.
    (d) By clicking the "Add Episode" button, the user can access the episode creation form.
    (e) After filling in all input fields, the user can optionally assign a series to that episode.
    (f) The user uploads a cover image and a source audio file for the episode.
    (g) Another option to provide an audio source file is to record in right in the app by clicking the corresponding option button.
    (h) After all data has been provided, the user can press "Submit" and an episode will be added to the Show.

2. View
    (a) On the dashboard, the user can see paginated previews of all their shows.
    (b) If the user hovers the cursor over a preview card, a "See More" button will be shown.
    (c) Clicking on this button will navigate the user to a show's full view page.
    (d) On this page there will be a paginated list of episodes with all necessary information and an audio player.
3. Deletion
    (a) After navigating to a show's view page, the user can hover their cursor over an episode card.
    (b) Two buttons will be showm: "Edit" and "Delete".
    (c) The user can delete the episode by clicking the "Delete" button.
    (d) The user will see a prompt in a pop-up asking them if they are sure about deleting this item.
    (e) From here the user can either confirm or cancel the deletion by clicking corresponding buttons on the pop-up.
4. Edit
    (a) After navigating to a show's view page, the user can hover their cursor over an episode card.
    (b) Two buttons will be showm: "Edit" and "Delete".
    (c) The user can edit the episode by clicking the "Edit" button.
    (d) The user will be navigated to a pre-filled form and will be able to apply any changes made by clicking "Submit".
    (e) After submitting the data, the episode will be updated.

Any additional navigation can usually be performed in one click through a button inside the header. Before authorization the buttons lead to "Sign In" and "Sign Up" options. If an authorized user goes back to Landing Page by clicking the logo etc. they can navigate back to dashboard through a corresponding header button. If an authorized user wants to log out they also will click a button in the application's header component.

Chapter 4

Perspective

The solution to the described problem requires an application, rich in functionality, complex in architecture and requires a lot of collaboration with the community, supporting platforms and needs a lot of organizational work done. While many of the planned features were implemented, this chapter will list some more features and work with community of the application that did not make it into this thesis' scope.

Some ideas and possible methods for their realization are listed in the subsections below.

4.1 Analytics

In order for a podcast host to be able to see their growth or decrease in listeners, they should be provided with a wide range of data on how, when, by who and to what degree their podcast is consumed within a requested time frame. The data shall be represented by corrseponding interactive UI elements. Possible libraries that could be used here are Chart.JS and D3.JS. A choice between these two tools will be made depending on the complexity of the visualizations. Chart.JS is great for common types of charts with high need in interactivity (tooltips, pop-ups), while D3.JS provides us with a big set of building blocks for a data visualization of almost any comlexity. Yet Chart.JS is much more organized and esier to use. Even thought D3.JS provides high customizability for the charts and other graphic elements, this library usually requires complex code structures in order to set up a necessary visualization.

4.2 Further Streaming Integration

We plan to expand the list of streaming services which we plan to add integration with. Such potential options include Deezer, Audible, Stitcher, Amazon Music and many more. Episode submission and processing will be as automatic as technically possible so the load of work on the user becomes decreased for a better experience.

4.3 Notifications

The process of submitting an episode to a streaming platform consists of several steps, one of which is validation by the streaming platforms moderating system. Sometimes the content does not pass the moderation for numerous reasons, yet the content owners pretty much never get notified of such incidents and have to figure out what happened either after seeing the absence of their episode on the streaming platform or after getting bad feedback from their audience. Including notifications

and other ways of communicating these problems to the podcast host will drastically reduce the risks in their career and will certainly improve the UX.

4.4 Advanced Promotion

Another complication for podcast content-makers lies in promoting their content on social media. For each of their streaming platforms they need to add a seperate link resulting in cumbersome posts all over their socail platforms. The solution here is to create a customizable one-page promotion sheet at our platform, a link to which the content-makers can provide in their promotion posts for a better view. The page will contain all necessary links in a comfortable view so the listeners can navigate with ease to all platform they wish to use for consuming their favourite content.

4.5 Improved Media Content

Other planned features include improved UI and logic for audio players and image loaders. Improved recording settings and other luxury features are also essential for a great user experience. Another idea that we have is a podcast cover image generator. If a user does not have their own cover design due to being a non-proffesional podcast host and doing this as a simple hobby, or due to some financial complications, the host can access the cover generator to use Amphora's own custom designs.

4.6 Improved Design

Due to budget restrictions, the current state of the application UI does not feature original icons and artwork. The current version of this application provides a simplistic yet thought-through UI/UX design which is going to be improved and enriched in the near future.

4.7 Hosting Service Migration

Another important feature to include is the service migration mechanism. If a user wants to switch from some other service to Amphora, they should have an easy way of doing so with all their data preserved and set up to work immediately. The list of migration mechanisms should include most of the popular services such as Buzzsprout, Podbean, RSS etc.

4.8 Adaptive Design for All Device Types

The usage of Ionic framework provides great potential to transform this web application into a PWA or a mobile application. Given that we orient this app to provide the possibility to record a podcast right on the platform, it provides great potential for express-podcasts that are potentially recorded out of the studio. Sometimes the user might not have a laptop on them while a mobile device is present in almost every single person's pocket. Thus an improved adaptive design for all viewports will also fix many downsides of the existing services providing a great solution to common problems.

4.9 Monetization

Another important thing to mention is podcast monetization. For this type of content to be profitable, our service will integrate itself with relevant monetization providers, one of which is Podcorn. This service centralizes monetization and consolidates independent podcasters, agencies and brands under one roof, giving more choice to brands and podcasters to find the right fit.

4.10 Improved Search

More options and filters are planned to improve the user's experience if they have lots of content hosted. Filtering the data by all parameters will help the user specify the exact set of shows or episodes the wish to see.

4.11 Tutorials

The app will feature a wide range of media and articles with thorough explanations about the details of application usage for both newcomers and veterans of the podcasting community. This will allow users to quickly immerse themselves into the flow of this service and use it in the most efficient way possible.

Chapter 5

Conclusions

After a thorough research and detailed description of the solution proposed in this thesis, the problems listed in chapter 2 were provided with solutions from both technical and design perspectives, described in great detail in chapter 3 and chapter 4.

In order for a service to be successful and comfortable in use, not just the functionality and business offers should be taken into account. Important technical details such as tools and technologies used, the software architecture, the UI/UX approach, productivity, scalability and resource consumption should be also kept in mind.

The existing alternative platforms contained the following problems and in this thesis following solutions are proposed:

1. Lengthy loading times - due to the framework of choice being Angular and its SPA nature, the consumption of band width and other network resources is decreased;
2. Constant necessity for page reload - an SPA with a data flow implemented with the use of reactive approaches ensures that the user sees their updates as soon as the changes are registered within the application;
3. Lack of structure in the content's display, complicated directory configurations - a reimagined data organization and its display allow the application to present its users with a well organized set of data, while search options allow the user to filter out unnecessary options. The idea of associating episodes with shows and series automatically structures the data without any need for user's configuration;
4. Outdated design - a more modern design approach with reusable and customizable components that receive data reactively solves this issue. The markdown is thought-through so the data is arranged in a pleasant view so the user is neither overwhelmed nor lost while traversing the UI;
5. Lack of customizability of content, lack of detailed settings - detailed forms and DTOs allow the user to enter all necessary data for a pleasant user experience for both the host and their listeners;
6. Chaotic user flow - a thorough UX planning with consistent user flows will reduce the amount of unnecessary actions needed from the user;

While "Amphora" strives to fix the flaws of existing services, it is also meant to bring a wide array of additional luxury features such as record-and-deploy functionality for specific use cases for seperate parts of the target audience, such as beginner podcast hosts, express podcast content makers etc. The marketplace still lacks modern

digital podcast solutions and could certaily use some diversity and improvement, in order for this amazing genre of content to thrive and serve the consumers well.

Bibliography

Daniels, P. (2017). *RxJS in Action*. Simon and Schuster. ISBN: 9781638351702. URL: https://books.google.com.ua/books/about/RxJS_in_Action.html?id=mjszEAAAQBAJ&source=kp_book_description&redir_esc=y.

Drifty Co. (2022). *Ionic Framework V3 Official Documentation*. URL: https://ionicframework.com/docs/v3/ (visited on 05/29/2022).

Farhi, O. (2017). *Reactive Programming with Angular and Ngrx: Learn to Harness the Power of Reactive Programming with RxJS and Ngrx Extensions*. Apress. ISBN: 9781484226209. URL: https://books.google.com.ua/books/about/Reactive_Programming_with_Angular_and_Ng.html?id=fzckDwAAQBAJ&source=kp_book_description&redir_esc=y.

Google LLC (2022). *Angular Official Documentation*. URL: https://angular.io/docs (visited on 05/29/2022).

Medium, Wes (2019). *Managing File Uploads with NgRx*. URL: https://medium.com/angular-in-depth/managing-file-uploads-with-ngrx-9fe07b084c1b (visited on 05/29/2022).

NgRx Team (2022). *NgRx Official Documentation*. URL: https://ngrx.io/docs (visited on 05/29/2022).

Rx Team (2022). *RxJS Official Documentation*. URL: https://rxjs.dev/guide/overview (visited on 05/29/2022).

Santamaria, Jose Maria Arranz (2015). *The Single Page Interface Manifesto*. URL: http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php (visited on 05/29/2022).