

Vivian Lunnikivi

# WEB FRAMEWORK MODERNIZATIONS

A pilot experiment transpiling AngularJS to Angular

Master of Science Thesis  
Faculty of Information Technology and Communication Sciences  
Examiners: Professor Kari Systä  
University Lecturer Matti Rintala  
November 2025

## ABSTRACT

Vivian Lunnikivi: Web framework modernizations  
Master of Science Thesis  
Tampere University  
Master's Programme in Information Technology  
November 2025

---

Early web sites were implemented manually from scratch as static HTML documents offering little, if any, interactivity. Web 2.0 popularized dynamic, on-demand content displayed on user interactions, and introduced web technologies that support creating and displaying the new, more versatile content. One popular technique to implement modern, interactive web applications are so called single-page applications (SPA). To support creation of SPAs, several web frontend frameworks have been developed, many of these written in JavaScript. These frameworks provide ready-made implementations and tooling for common features and techniques seen in SPAs and during their development. The purpose of web frameworks is to facilitate web development and improve the quality of developed applications, when developers are not forced to solve every detail of the development process themselves, but can tap into the expertise of the framework developers.

The downside of web frameworks, as is for other technologies, is that they deprecate. Security updates or changes in the philosophy of how a framework is architected occasionally lead to newer versions of the framework being incompatible with the previous versions of the technology. Alternatively, the maintenance of a framework may cease altogether. Applications written with deprecated framework versions need to be updated to newer versions or migrated to an alternative technology altogether. These types of modernization problems are often called application migrations.

Migrating an application is often a cumbersome undertaking, especially in the case of larger code bases. Therefore, several approaches and tools have been developed to assist in different kinds of migration problems. A summary of approaches addressing web framework migrations is presented in this thesis. Special attention is given to migration of AngularJS applications, and a pilot experiment is conducted to explore the particular migration problem from AngularJS to Angular.

AngularJS is a web frontend framework originally released in 2010, used to build SPAs. In 2016, the developers of AngularJS released a complete rewrite of the framework in TypeScript, calling it Angular. The original AngularJS framework had an end-of-life in 2021, forcing any applications written in AngularJS to be upgraded, for example, to Angular. The approach the Angular team recommends for the migration is to utilize the Angular ngUpgrade module, which enables running both versions of the framework in parallel in the same application. This allows for upgrading the code base one part of the application at a time. The downside of this approach is that it does not decrease the amount of required manual work in the conversion.

An alternative approach to the migration is to perform the migration in one go, making use of helper tools that automate parts of the conversion. One such tool, `angularjs-to-angular`, has been developed for aiding the migration from AngularJS to Angular. In this thesis, the `angularjs-to-angular` tool is tested by performing a pilot experiment in which a simple test application built on a handful of the core functionalities of AngularJS, is upgraded to an Angular 6 application, utilizing the tool during the modernization.

To validate the resulting application and the migration approach used to produce it, a reference modernization is performed on the same application. The reference application is formed by following the approach laid out by the Angular migration guide, where Angular is brought alongside the AngularJS application in the original application and the application is converted piece by piece.

As a result of the experiment, both approaches produce a starting and mostly functional, pure Angular 6 application, verified by a set of manual functional tests. The applications implementations differ mostly in terms of tooling used to implement them, but some structural differences also emerge. Both approaches require preparation steps to be performed on the test application before the upgrade, but most of the steps were either identical or very similar. Most of the differences in the approaches relate to performing the actual upgrade. In one hand, the reference approach requires some additional work in terms of setting up the intercommunication between application parts implemented in different versions of the framework, but as an upside, the application remains functional throughout the conversion process. On the other hand, the script-assisted modernization approach only utilizes one framework version at the time, but there is a moment during the upgrade when most parts of the application need to be converted at once, which might prove difficult for larger applications.

Keywords: framework, modernization, transpiling, AngularJS, Angular

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

# TIIVISTELMÄ

Vivian Lunnikivi: Web framework modernizations

Diplomityö

Tampereen yliopisto

Tietotekniikan DI-ohjelma

Marraskuu 2025

---

Varhaiset verkkosovellukset toteutettiin käsin alusta loppuun staattisina HTML-dokumenteina, jotka tarjosivat vähän, jos ollenkaan vuorovaikuttavia komponentteja. Web 2.0:n myötä dynaaminen, käyttäjän vuorovaikutuksesta ladattava ja esitettävä sisältö yleistyi, ja samalla kehitettiin teknologioita, jotka mahdollistivat monipuolisemman sisällön esittämisen. Yksi suosittu tekniikka modernien vuorovaikuttavien sovellusten tuottamiseen ovat niin sanotut yhden sivun verkkosovellukset eli SPA:t (Single-Page Application). SPA:iden toteuttamiseen on kehitetty 2000-luvulla useita, erityisesti JavaScriptillä toteutettuja viitekehyskäytännöjä, jotka tarjoavat valmiita toteutuksia erilaisille operaatioille, joita SPA-sivuilla ja niiden kehityksen aikana on tapana käyttää. Viitekehysten tarkoituksena on muun muassa nopeuttaa sivujen kehittämistä ja tuottaa laadukkaampia sovelluksia, kun kehittäjien ei tarvitse ratkaista jokaista sovelluskehitykseen liittyvää yksityiskohtaa alusta asti.

Viitekehysten, kuten muidenkin teknologioiden varjopuolena on niiden vanheneminen. Tietoturvapäivitykset tai muutokset viitekehysten toteutusfilosofiassa usein johtavat tilanteisiin, jossa viitekehysistä julkaistavat uudet versiot eivät ole enää yhteensopivia edellisten versioiden kanssa. Toisinaan myös viitekehysten ylläpito voidaan lopettaa kokonaan. Vanhentuneilla viitekehysversioilla toteutetut sovellukset tulee ennen pitkää päivittää käyttämään uudempiä versioita tai viedä täysin toiselle viitekehyskäytännölle, jolloin puhutaan sovellusmigraatioista.

Kokonaisten sovellusten migraatointitehtävät ovat usein erityisesti suurempien sovellusten kohdalla suuritöisiä hankkeita, joten lukuisia lähestymistapoja ja työkaluja on kehitetty erilaisten migraatio-ongelmien ratkomisen avuksi. Tässä työssä tehdään katsaus lähestymistapoihin, joissa käsitellään verkkosovellusten migraatioita viitekehyskäytännöiltä toisille, ja erityisesti keskitytään AngularJS-sovellusten päivittämiseen.

AngularJS on alkuun vuonna 2010 julkaistu verkon asiakaspuolen SPA-teknologia, josta AngularJS:n tekijät julkaisivat vuonna 2016 alusta asti TypeScriptillä uudelleenkirjoitetun version, kutsuen sitä nimellä Angular. Alkuperäinen AngularJS vanhentui vuonna 2021, jonka jälkeen viimeistään kaikki vanhalla AngularJS:llä kirjoitetut sovellukset täytyy migrataa, esimerkiksi Angularille. Angular-tiimin suosittu tapa migraatioon on hyödyntää migraatiossa Angularin tarjoama ngUpgrade-moduulia, joka mahdollistaa sekä AngularJS:llä että Angularilla toteutettujen sovelluskoodiosoiden ajamisen rinnakkain samassa sovelluksessa. Tämä mahdollistaa koodipohjan uudistamisen yksi sovellusosa kerrallaan. Lähestymistavan kääntöpuolena on, ettei se vähennä migraatiossa vaadittavan käsin tehtävän muunnostyön määrää.

Toinen tapa on toteuttaa migraatio on kerralla, hyödyntäen apuna ohjelmakoodilla toteutettuja aputyökaluja. Eräs tällainen aputyökalu on nimeltään angularjs-to-angular, joka on toteutettu juurikin AngularJS-sovellusten migraatointiin Angularille. Tässä työssä työkalua testattiin suorittamalla koe, jossa yksinkertainen AngularJS-viitekehyskäytännön muutamia oleellisia ominaisuuksia hyödyntävä testisovellus päivitettiin Angularin versioon 6 hyödyntäen angularjs-to-angular työkalua.

Varsinaisen käännöksen ja käännöksen muodostamiseen käytetyn lähestymistavan validoi-

miseksi samasta testiohjelmasta muodostettiin vertailukäännös myös noudattaen Angular-tiimin suosittamaa migraatiotapaa, jossa Angular-viitekehys tuotiin alkuperäiseen sovellukseen AngularJS:n rinnalle, ja sovelluksen osat muunnettiin Angularille yksi kerrallaan.

Kokeen tuloksena molemmat lähestymistavat tuottivat käynnistyvän ja suurimmilta osin toiminnallisen, puhtaasti Angularin versiolla 6 toteutetun sovelluksen. Toiminnallisuus varmistettiin joutamalla manuaalisia toiminnallisia testitapauksia. Sovelluksien toteutukset eroavat toisistaan lähinnä niiden käyttämien työkalujen osalta, mutta joitakin rakenteellisiakin eroja sovelluksiin muodostui. Kumpikin lähestymistapa vaati esivalmistelua käännettävälle sovellukselle, tosin esivalmisteluaskeleissa lähestymistavoissa ei ollut juurikaan eroa. Suurimmat erot lähestymistavoissa liittyivät varsinaisen käännöksen suorittamiseen. Referenssimodernisaatio vaatii oman ylimääräisen työnsä komponenttien eri versioiden välisen vuorovaikutuksen fasilitointiin, mutta referenssimodernisaation aikana sovellus pysyy jokaisen komponentin päivittämisen välissä kääntyvänä ja toiminnallisena. Toisaalta varsinaisessa käännöksessä sovellusta ajetaan vain yhdessä viitekehyksessä kerrallaan, mutta päivityksessä muodostuu hetki, jonka aikana oleellisin osa sovelluksen osista tulee päivittää kerralla. Se voi olla hankala toteuttaa suuren sovelluksen tapauksessa.

Avainsanat: viitekehys, modernisaatio, sovellusmigraatio, SPA, yhden sivun verkkosovellus, transpilointi, AngularJS, Angular

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

## USE OF ARTIFICIAL INTELLIGENCE IN THIS WORK

Artificial intelligence (AI) tools have been used in this work:

Yes

No

I hereby declare, that the AI-based applications used in generating this work are as follows:

| Application | Version |
|-------------|---------|
| elicit.com  | 2024    |
| OpenAI GPT  | 4.1     |

### Purpose of the use of AI

During the background research phase of the thesis work, Elicit was utilized to identify some additional references. Supplementary meaning that researching was mainly executed manually via traditional searches in research databases and reference databases. The first prompt for Elicit considered approaches to modernizing web applications, focusing on papers that include transpilation as a methodology, or where the source or target technology is a single-page application. The second prompt was for finding AI-assisted approaches developed for modernizing or transpiling web applications. The third for searching sources about LLM technologies and the last about Model-Driven Engineering. Some of the studies recommended by the Elicit tool were read to develop understanding over the topics.

### Parts of this work, where AI was used

All text in the document is hand-written without the use of AI, except for chapter 5, where the experiment setting was explained to GPT model 4.1 and then the LLM was asked to present an example structure for the given information following the best practices of a thesis work. The suggested structure was used as a basis for the structure seen in chapter 5 on title level, but the structure ended up modified heavily during the otherwise manual writing process.

## **Acknowledgement of risks**

I hereby acknowledge, that as the author of this work, I am fully responsible for the contents presented in this thesis. This includes the parts that were generated by an AI, in part or in their entirety. I therefore also acknowledge my responsibility in the case, where use of AI has resulted in ethical guidelines being breached.

## PREFACE

If there ever was a long project, it was my master's thesis. In a seemingly straight-forward project, there was a surprising amount of desperation, illness, changing jobs and other life upside-down turning moments. Never in my life have I been more certain I would fail than with this project. Seldom have I been more pleased to have been proven wrong.

Topmost in my head there is the immense gratitude to the multitude of the capable, empathetic, and supportive people that helped me along this journey.

First, a thank you to my colleagues at Patria whom the idea for this thesis sparked with. As a part of a lunch conversation. As one does. Especially, thank you to my supervisor Jarmo, our brilliant architect Kimmo and a close team member Antti for the support and interest towards my work, career and well-being.

I would also like to thank my current employer Futurice for the concrete support in the form of a master thesis bootcamp, and the capable instructor Jesse Haapaniemi there, for most useful writing tips. A special thank you to my supervisor Miika for all the sparring and support with practicalities. For attitude sparring, I'd also like to thank my close colleague Lumi.

My apologies and thanks to Teekkarikvartti, Teekkarikuoro, Aukea Ensemble and Ruoste-vaurio for allowing me to take part in the small capacity that I have been able to alongside working. For me, music and creating with you wonderful and warm-hearted people has been the predominant opportunity for respite, self-expression and enjoying life for the past decade. I would be at loss without you.

Thank you also for my brother Henri for taking the time to answer the many practical questions I have had. Alongside Henri, the list of rubber ducks to thank is too long to include here. Thank you all regardless.

My utmost appreciation and gratitude to Janne, who taught me what "support" means.

Lastly, thank you to my thesis supervisor Kari, who so patiently and promptly has been giving feedback on my work during a project that ultimately took way longer than I dare to admit.

At Tampere, 26th November 2025

Vivian Lunnikivi

## CONTENTS

|       |  |    |
|-------|--|----|
| 1.    | Introduction . . . . .   | 1  |
| 2.    | Modern web development with SPA frameworks . . . . .   | 4  |
| 2.1   | Application development with frameworks . . . . .  | 4  |
| 2.2   | Framework deprecation . . . . .  | 6  |
| 3.    | Web application modernization approaches . . . . .   | 8  |
| 3.1   | Model-driven re-engineering . . . . .  | 8  |
| 3.1.1 | Model-driven Architecture and Model-Driven Engineering . . . . .                                 | 9  |
| 3.1.2 | Model-driven Reverse Engineering and model-driven approaches<br>to modernizations . . . . .      | 10 |
| 3.2   | Applications of artificial intelligence in modernizations . . . . .                              | 15 |
| 3.3   | Dedicated parser for migrating AngularJS to Angular. . . . .                                     | 18 |
| 3.3.1 | The official migration approach from AngularJS to Angular 2.0+ . . . . .                         | 19 |
| 3.3.2 | angularjs-to-angular by Grubhub . . . . .  | 19 |
| 4.    | Transpiling AngularJS to Angular with a dedicated parser. . . . .                                | 20 |
| 4.1   | Modernization approach overview . . . . .  | 20 |
| 4.2   | Selecting a test application for the modernization test . . . . .                                | 21 |
| 4.2.1 | The source application: angular-phonecat . . . . .   | 21 |
| 4.2.2 | A reference modernization for the angular-phonecat application . . . . .                         | 26 |
| 4.3   | The modernization tool: angularjs-to-angular . . . . .   | 30 |
| 5.    | Transpilation process and results . . . . .  | 32 |
| 5.1   | Application preparation before conversion . . . . .  | 32 |
| 5.2   | Starter application . . . . .  | 34 |
| 5.3   | The upgrade . . . . .  | 34 |
| 5.4   | A working modernization of the angular-phonecat . . . . .  | 35 |
| 6.    | The results and validation: Comparing the upgrade with the reference modern-<br>ization. . . . . | 39 |
| 6.1   | Functional tests . . . . .   | 39 |
| 6.2   | Structural validity . . . . .  | 40 |
| 6.3   | The validity of the conversion approach . . . . .  | 42 |
| 7.    | Discussion . . . . .   | 45 |
| 8.    | Conclusions . . . . .  | 47 |
|       | References . . . . .   | 48 |

## ABBREVIATIONS AND DEFINITIONS

|                     |  |
|---------------------|--|
| Angular             | A web frontend framework, and successor to AngularJS   |
| AngularJS           | A deprecated web frontend framework with end-of-life in the end of 2021.   |
| AST                 | Abstract syntax tree. A tree representation of a program.  |
| CDO                 | Component Definition Object. A way to define an AngularJS component configuration as an object, passed to a component registration call.   |
| CLI                 | Command-line interface, a tool that provides a text-based interface from command-line.   |
| ECMAScript          | The programming language specification for JavaScript  |
| Forward engineering | The part of the re-engineering process where a new version or view of the system is developed, usually involves, for example, generating code. See chapter 3.1   |
| MDD                 | Model-driven development, see Model-Driven Engineering   |
| MDE                 | Model-driven engineering, is a software engineering methodology that utilizes conceptual models as the base artifact to represent domain knowledge.  |
| MDRE                | Model-driven reverse-engineering, a field that applies MDE techniques in reverse engineering, see chapter 3.1  |
| MDWE                | Model-Driven Web Engineering refers to applying the practices of Model-Driven Software Engineering in the context of web applications, that is, advocating models as the key development artifact in all phases of the development of a web application. The approach drives separation of presenting information of the system from its technical implementation [68] |
| Migration           | The act of moving an software application implementation from one implementation technology to another.  |

|                     |  |
|---------------------|--|
| MVC                 | Model-View-Controller, an architectural pattern that organizes application functionality around the concepts of a model, a view and a controller. The model refers to the definition of application data, view presents the data and the controller represents the business logic orchestrating application functionality. |
| Reverse-engineering | The part of the re-engineering process that covers techniques to learning about and documenting the behavior and structure of an existing system   |
| SOA                 | Service-Oriented Architecture, the architecture resulting from viewing business activities from a service-oriented point-of-view, that is, prioritizing a set of activities that help an organization create business value.   |
| Source application  | The AngularJS application that is modernized during the modernization pilot in this thesis   |
| Starter application | Angular 6 application serving as a base application for building the modernization result  |
| Target application  | The conversion result application acquired by modernizing a source application.  |
| Transcompiler       | Source-to-source code translator [56], synonymous to "transpiler".   |
| Transpilation       | Machine-based source-to-source code translation from a programming language or framework to another.   |

## 1. INTRODUCTION

In modern web development, web frameworks are used to make implementing web applications faster and easier, as frameworks provide ready-made and tested implementations for common web engineering problems [92]. However, the life span of any given framework is not tied to the life cycle of the applications developed with the framework. As frameworks deprecate, applications relying on the framework need to be modernized to avoid a variety of risks arising from, for instance, the end of security updates received for the deprecated framework. Software application implementations often rely heavily on the implementations provided by the utilized software frameworks, which makes rewriting the application with modern technology or migrating the application to a new framework often a cumbersome task, featuring manual code conversion or re-implementation work involving all parts of the application that rely on the framework.

One such deprecated framework is AngularJS, which is a JavaScript-based web frontend framework first published in 2010 [22] and with end-of-life in the end of year 2021. The AngularJS development team recommends migrating existing AngularJS applications to Angular, which is a TypeScript-based rewrite of the original framework. [36, 93] To support the transition, the Angular team offers a built-in helper library called ngUpgrade to run AngularJS and Angular in parallel in the same application. [93] According to the Angular team, running the source and target frameworks in parallel allows seamless incremental modernization of the application code to Angular, by rewriting the application in Angular in pieces. The helper library itself does not perform any code conversions. Therefore, the conversion work is left to be done by other means. [93, 40]

The research from the early 2000's on the modernization of Web 1.0-styled legacy web sites to Rich Internet Applications (RIA)<sup>1</sup> concerned enriching the user experience through the introduction of rich features of the user interface and modernizing the architecture

---

<sup>1</sup>Rich Internet Application is a term coined in the early 2000's to refer to the user-centric websites of the new millennium. RIAs equip user interactions with websites and other internet users by incorporating varying technologies such as bidirectional asynchronous communication over internet connections, initiated by the client or server, and executing business logic or storing data on the client end as well. [42] These rich interface features expand the web user role from passive content consuming to content creation, opposed to the thin clients of traditional web sites, which limit user interactions to link navigation and form submissions. Unlike traditional web sites that respond to user interactions by full-page reloads, a typical RIA follows the Single-page Application (SPA) architecture [65], which construct web sites that refresh only the necessary parts of the page content [49, 42, 65]. The term Web 2.0 and the retronym Web 1.0 have also been used to emphasize this distinction between the web sites of the past and the current millennium. [59]

of existing legacy web software constrained by features of pure HTML. Systematic approaches for modernizing traditional web pages to RIAs were presented. For example, the model-driven approach developed by Rossi et al. [77] and the reverse engineering approach of Mesbah and Deursen [65]. Some of these works even include code generation based on models representing the original application. [60, 59] While the late 2000s were dominated by AJAX<sup>2</sup> powered development of rich single-page applications, the turn of 2010 marked the beginning of framework-empowered web development by the emergence of rapidly proliferating frameworks, many of which also adopted the SPA architecture and AJAX frameworks under the hood – AngularJS [4], React [86] and Vue.js [48] to mention a few of the popular ones.

Modernizing SPAs from deprecated SPA frameworks to up-to-date frameworks has been addressed in fewer studies. Kaushalya and Perera [53] developed a conversion framework and helper tool for modernizing AngularJS applications to React. Similarly, Verhaeghe et al. [97] developed an approach for modernizing SPAs written in a Java-based framework, Google Web Toolkit (GWT), to React. Outside of the scientific context, the conversion of AngularJS to Angular has been addressed by developers at GrubHub, who published a transpiler<sup>3</sup> tool [23] and a series of blog posts that discuss the conversion. [47, 18, 29, 88]

Similar to the presented work related about modernizing SPAs, the goal of this thesis is to outline a method for modernizing SPAs implemented with AngularJS. The target framework is Angular, the modern counterpart of AngularJS. Modernization enables further usage and development of applications written originally in AngularJS. Furthermore, this thesis focuses on a transpilation solution that automates parts of the conversion and aims to minimize manual work during the migration. This thesis describes the required steps to transpile AngularJS applications to Angular, based on a transpilation tool originally developed by the development team of Grubhub company.

We also discuss the required application preparation steps and limitations that apply when using the approach for modernizing AngularJS applications. As a proof-of-concept, we test the approach by transpiling a fairly simple AngularJS test application built on a set of main features from the AngularJS framework. The conversion result is built on a minimal started project obtained by using `angular-cli`, and the resulting application and the

---

<sup>2</sup>AJAX, shorthand for Asynchronous JavaScript and XML, describes a set of web techniques developed in the 1990s to early 2000s to enable developing interactive web applications that utilize a browser-builtin JavaScript interpreter to enable asynchronous communication over HTTP to web servers without blocking user's interaction in the browser view [75]. Browsers interact with the server via the XMLHttpRequest object [105], which in practice, is how JavaScript web client frameworks implement the AJAX functionality under the hood.

<sup>3</sup>By transpilation, this thesis refers to the process of machine-based source-to-source code translation from a programming language or framework to another. A tool that performs such translations is called a transpiler. Unlike compiling, the target code produced by transpilation is approximately at the same level of abstraction as the origin code. This is the definition often adopted in work related to code translations, e.g. [61, 97, 64].

modernization process are validated by a set of functional tests, and by comparing them to a reference modernization. The reference modernization is produced by following the manual migration guide [93] from AngularJS to Angular 6 given by Angular team.

Ultimately, the pilot experiment demonstrates that the `angularjs-to-angular` upgrade tool can be successfully utilized to perform a test application migration from AngularJS version 1.5 to Angular version 6. The required modernization steps and limitations are documented.

The structure of the thesis goes as follows. Chapter 2 familiarizes modern web development with frameworks. Chapter 3 summarizes previously presented work on approaches to web application modernizations. Chapter 4 lays out a transpilation experiment on a dedicated parser tool, which is applied to an example application in chapter 5. The results of the pilot experiment are summarized and validated in chapter 6. Chapter 7 discusses how the work relates to a broader context of modernization approaches and presents questions for further research. Lastly, chapter 8 concludes the work.

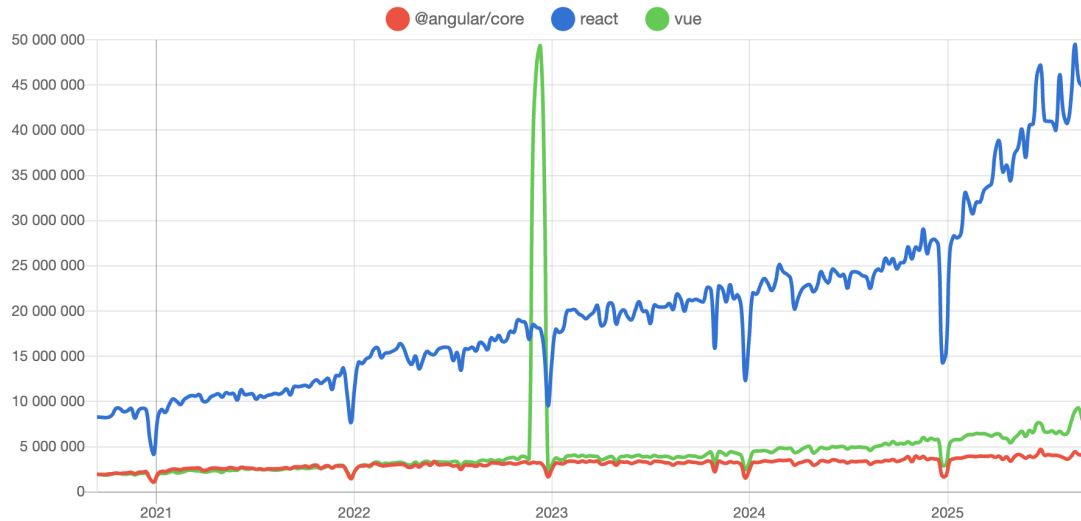
## 2. MODERN WEB DEVELOPMENT WITH SPA FRAMEWORKS

### 2.1 Application development with frameworks

Modern JavaScript frameworks make web application development, testing, and maintenance faster and more cost-efficient, opposed to writing application code manually from scratch. Frameworks aim to enable more robust, more secure, and higher-quality application code by providing tested, opinionated, but customizable, implementations for recurring engineering problems in the scope of the framework. [92] The scope of any given framework varies from solving an isolated problem area such as how to implement HTTP requests between a web client and a server, as axios [45] does, to providing a full-fledged architecture templates for an application, as Next.js [69] does. A JavaScript framework may also target either the client side, the server side, or both. For example, Vue.js targets the web frontend [48], the Express framework targets the web backend [38] and Next.js allows building both the server- and client-side functionality of an application [69]. Additional benefits of using frameworks include bug and security fixes, and the introduction of new backward compatible features in the framework. However, an organization or individual using a framework often has little control over how the framework is developed moving forward, especially in the context of open-source software (OSS). [62]

There often exist several opinions on how to best implement software architecture. Therefore, it is understandable that there are several competing frameworks that provide their own opinionated style of implementation for similar problem areas. More frameworks also continuously emerge from the software engineering community. For instance, the figure 2.1 demonstrates the popularity of a couple of popular, competing OSS frontend frameworks in September 2025: Angular, React, and Vue.js.

Each of the three technologies represents or provides support for building web sites as SPAs. [4, 86, 48]. In an SPA, instead of the user navigating between multiple web documents as a result of user interactions on a web site, only a single web document is initially downloaded from a server, after which the content of the same document is dynamically updated in response to the interactions. Browsers use JavaScript APIs to manipulate the



**Figure 2.1.** The download counts of the SPA core packages of the frameworks Angular, React, and Vue.js, over the time period of 5 years from Sep 13th, 2020 to Sep 7th, 2025. Figure is drawn with the NPM trends [71] OSS tool that visualizes the downloads of selected NPM packages over time. The download frequency can be used as an indicator for predicting the continuation of developer community's support for a given NPM package moving forward. [90]

DOM<sup>1</sup>, to refresh only the parts of the page's body that require changes. Supplementary resources are downloaded from the page via web APIs as needed, in hopes of reducing the download and rendering time of the initial document. In comparison to traditional web sites constructed from a series of static HTML documents downloaded one at a time and rendered as full page reloads, the SPA style aims to provide highly dynamic, user-friendly applications that reduce web traffic and latencies in the user interface, and shift processing load from server to client-end, all of which explain the popularity of SPAs. [49, 42] However, SPAs also have drawbacks, such as more effort required to implement Search Engine Optimization (SEO) [41, 49].

Despite the several benefits of using frameworks, there are also significant downsides. One notable factor is the life cycle of the frameworks in use. Especially in the context of OSS frameworks, the development of the component is often driven by the motivations of the contributors [62]. Occasionally, OSS projects may get the backing of large software companies, such as Vue.js [101], which may prolong the lifetime of the component. An indicator used in the industry to predict the lifespan of the project is the presence of an active community developing the framework [62, 72, 92]. When developers start shifting their way from the framework – either from using it or as decreased release activity [97], it can be interpreted as a sign of the framework's nearing end-of-life, motivating migration

<sup>1</sup>Document Object Model (DOM) is the in-memory representation of a web document's structure that details the rendering of, and allows the programmatic manipulation of the content of a web document via a standardized API [103, 37]

away from the framework [97].

## 2.2 Framework deprecation

In the case of AngularJS, the official end-of-life of the framework dates in the end of year 2021 [36], after which no new bug fixes, features, or maintenance for the framework has been or will be offered, even if security vulnerabilities are revealed in the framework. This forces any projects that are still using AngularJS to detach from the deprecated framework.

Several options exist for moving forward. An existing AngularJS project can be abandoned altogether or a complete rewrite of the project can be implemented on another framework that has support in the future. However, abandoning an application might not be possible and rewriting it manually is seldom a feasible option either, as applications may feature thousands or even hundreds of thousands of lines of code, making rewrites costly undertakings, as Somogyi and Kövesdán explain [84]. Indeed, re-implementing already delivered software solely for maintenance purposes is reluctantly funded work and might introduce new problems in the system [84].

The improved user experience may explain the motivation behind the migration movement from traditional multi-page web sites to AJAX SPAs in the early 2000s [59], but migration between web frameworks or framework versions may not produce much visible improvement for the end user [84]. Furthermore, the common sense indication is that any migration work ties development resources down, keeping them from developing other new value, which further discourages allocating resources for manual system rewrites.

Consequently, numerous techniques and tools have been developed to aid the conversion work. [84] For example, the Angular team provides a library, ngUpgrade, that enables the AngularJS framework to run alongside Angular in the same project, allowing for an incremental migration from the deprecated framework to the new one. [36] Incremental migration means performing the migration a small portion, such as a single component at a time, slowly getting rid of the deprecated code over time while still allowing new features being developed with the modernized technology alongside the modernization work. Furthermore, this is the migration approach the developers of AngularJS recommend [36][40].

Then again, in the case of AngularJS, a paid extended support period was also made available on a contract basis [39], allowing contract organizations to avoid imminent migration for the contract period. However, once the long-term support (LTS) period is over, the need to migrate is imperative. Running deprecated code may expose the application, for example, to security vulnerabilities that cannot be fixed without breaking the application code. For example, a critical new security issue fix might be available only in newer

versions of a package referenced by the framework that the application builds on. To receive the fix in the affected package, the framework code must be updated. This might require updating the framework's dependency over a major version<sup>2</sup>, since the framework itself would break, if the references were not updated in the framework code. Thus, if the framework itself isn't maintained anymore, using a deprecated framework would block fixing the new security issues in the application as well. This is just one example of a maintenance problem created by deprecated software. Therefore, the demand for automated migration tools is evident: a tool that could automatically convert the existing code base to another language or framework could eliminate migration costs altogether and allow development to continue smoothly. Even partial automation of the transformation process could prove beneficial compared to fully manual conversion.

---

<sup>2</sup>The publicly distributed JavaScript packages, like the OSS JavaScript frameworks and their dependencies discussed in this thesis, often follow a widely accepted practice called semantic versioning. For example, the npm software registry recommends following the semantic versioning practice when distributing software via npm's registry [2]. Semantic versioning communicates how the changes introduced in a new package release affect the software systems that utilize the package in its source code, aka., the dependent software systems. Any major releases in a semantically versioned package introduce changes in the package's public API that are not backward compatible, meaning that any references to updated parts of the public API will break. [80]

## **3. WEB APPLICATION MODERNIZATION APPROACHES**

As illustrated in chapter 2, there is a need for tools aiding the modernization of web applications implemented with deprecated web frameworks. The topic of systematized transformations has sparked a significant amount of research over the past few decades, and several approaches have been developed to perform these conversions. This chapter first summarizes existing approaches and clears out the terminology related to web application modernizations and categorizes the approaches on a general level into model-driven approaches, AI-assisted transformations, and dedicated parsers. The chapter then summarizes the approach the AngularJS team recommends for making the transition to Angular and gives a short introduction to a dedicated parser tool made for this purpose.

### **3.1 Model-driven re-engineering**

Re-engineering is a term used in software development and maintenance to describe the process of gaining understanding of, and systematically evolving existing software systems [34], making application modernizations a type of re-engineering problem. Re-engineering may be motivated by a variety of reasons, such as quality improvements or extending the lifespan of a given system, as discussed in chapter 2. According to the definition given by Chikofsky and Cross already in 1990 [34], the re-engineering process itself consists of three separate stages: reverse engineering, restructuring, and forward engineering. Reverse engineering covers techniques that examine an existing system to learn about it and document the specifications of its behavior and structure. During reverse engineering, system components and their interrelationships are identified and analyzed to create abstract representations of the system. In the restructuring process, desired transformations, such as refactors or changes in functionality, are performed on the presentations. Restructuring creates new transformed system representations that are then used in the forward engineering step to add implementation details to create an updated concrete implementation of the system. [34] In other words, to re-engineer a software system, one needs to understand the existing system first, for which purpose several reverse engineering approaches exist, such as the techniques for Model-Driven Reverse Engineering (MDRE).

### 3.1.1 Model-driven Architecture and Model-Driven Engineering

This subsection lays out a foundation for understanding MDRE as a modeling-based approach to reverse engineering [82]. MDRE is an application of Model Driven Engineering (MDE) [82], which in turn has been derived from Model-Driven Architecture (MDA) [54].

Model-Driven Architecture is an approach to specifying large and complex systems separate from their implementation, first defined by the Object Management Group (OMG) standards consortium in 2001 [73], revised in 2014 [66]. The notion behind MDA is that the use of well-defined standardized models captures the information in the said systems in a manageable format. When expressed as models, complex systems can be managed by deriving necessary operations and artifacts from the models with semi-automated or automated tools. The Object Management Group believes that the approach improves the communication and agility of system life cycle processes such as planning, designing, and implementing, which helps organizations manage different systems within the organization. The approach can be applied to both software and organizational systems and processes. [66]

As stated above, in the MDA approach, system development is supported by deriving value from models and architecture. [66] Value in this context means, for example, documentation, system requirements, or technical artifacts, such as source code or database schemas. The value derivation requires that the structure, semantics, and notation of the models are expressed in a well-defined and widely understood language. That is, MDA models are defined in standardized languages such as the Unified Modeling Language (UML)<sup>1</sup> or the Knowledge Discovery Metamodel (KDM)<sup>2</sup>. MDA standards include models and model mapping descriptions in several modeling languages, and how to transform, execute, or exchange the models to allow realizing artifacts and implementations. [66] For example, there could be a partially or fully automated model transformation path from a Platform Independent Model (PIM) like a class diagram to a Platform Specific Model (PSM) like an XML schema<sup>3</sup>.

However, in 2002, Kent argued that MDA ignores several important aspects of the model-driven approach and proposed Model-Driven Engineering (MDE) as an extension of MDA [54] as a point of reference to what aspects still need to be considered to harness the full potential of the model-driven approach in system management and development. Kent discusses how the process of generating system models and model mappings is

---

<sup>1</sup>Unified Modeling Language (UML) is an open language standard maintained by the Object Management Group for modeling and documenting software systems [91] as specific-purpose diagrams, such as the class diagram [82]

<sup>2</sup>Knowledge Discovery Metamodel (KDM) is another open standard maintained by the OMG for defining meta-model representations of existing software systems [55]

<sup>3</sup>Extensible Markup Language (XML) is a markup language with a standardized syntax that allows defining documents of data and its structure in a format that is both human and machine readable [104]

connected to the models and mappings themselves, and points out missing modeling dimensions along the abstract-to-concrete axis of PIM to PSM. For example, Kent argues that the MDA approach does not account for the stakeholders or the subject area of the model, and thus does not provide the necessary conceptual framework to identify and create necessary models in the modeling space. [54] Therefore, in the eyes of this thesis, despite MDE having its roots in MDA, MDE is seen as a roof term for model-driven techniques to system re-engineering. The development and application of MDE techniques has received a significant amount of research attention, for which reverse engineering is a popular application field. [82]

### **3.1.2 Model-driven Reverse Engineering and model-driven approaches to modernizations**

Model-driven reverse engineering (MDRE) is a field that applies MDE techniques in reverse engineering, often as part of re-engineering efforts. [82] Although these approaches are aimed at reverse engineering, many MDE techniques can also be applied to other parts of the re-engineering process, like forward engineering, and chained to perform a complete re-engineering transformation. [82] MDRE has been studied extensively, and the state of research has also been mapped and laid out in 2024 by Siala, Lano, and Alfraihi [82]. The systematic literature review of Siala et al. covers 538 papers, of which they identify 64 key approaches to MDRE. They categorize the approaches according to the objective of the approach; The 14 identified distinct objectives include 16 approaches in modernization, 11 in program migration, and 8 in program transpilation, as listed in table 3.1. Table 3.1 highlights the approaches that use web applications as the source system. Five of the approaches have the objective of modernization, program migration, or program translation / transpilation, namely the approaches A1, A4, A10, A19, and A21. Moreover, approaches A25 and A35 use web applications as the source system, despite the objectives of those approaches being software maintenance and reuse for A25, and software testing for A35. Therefore, the categorizations made by Siala et al. identify seven MDE-based approaches potentially relevant to the topic of this thesis. These approaches are summarized in the following paragraphs.

The approach A1 presents an MDRE-based open source framework called MoDisco for reverse engineering legacy systems to support system modernizations [31]. The MoDisco framework presented as a generic, technology-independent approach that builds the support for specific technologies on a plug-in principle. This is possible because the approach decouples the system representation and the reverse engineering scenarios from each other; the legacy system to re-engineer is first transformed into initial models that represent the system on a low level, resembling closely the legacy system itself. This phase is called "model discovery", and the technology-specific software components needed

| The objective                       | The source is not 'Web Applications'                         | The Source is 'Web Applications' |
|-------------------------------------|--|----------------------------------|
| Modernization                       | A5, A7, A9, A11, A20, A29, A32, A43, A45, A48, A50, A53, A55 | A10, A19, A21                    |
| Program migration                   | A7, A11, A14, A15, A20, A38, A39, A48, A49, A50              | A4                               |
| Program translation / transpilation | A27, A44, A45, A46, A49, A54, A55                            | A1                               |
| Other                               | <i>Several approaches...</i>                                 | A25, A35                         |

**Table 3.1.** In their work, Siala et al. categorize MDRE approaches by the source system of the approach, and by the objective of the approach [82]. Based on the categorizations made by Siala et al., this table collects the approaches that target either software modernization, migrations, or translations / transpilation, and divides the approaches into web and non-web approaches depending on whether the source system for the approach is a web application or other kind of a system.

to automatically generate the unprocessed raw model representations are called model discoverers. The next step of the process is called "model understanding". In model understanding, reusable model manipulation operations are chained on the initial models to implement desired transformations and obtain derived models and views of the legacy system.

To use the approach to re-engineer a legacy system, the appropriate technology-specific model discoverers need to be built. According to Brunelière et al., model discoverers have been developed for Java core language and Java Enterprise Edition, including JavaServer Pages, and those have been used in reverse engineering scenarios. [31] For example, the authors summarize a use case where the framework is applied successfully to implement automatic refactors in a Java codebase of over a million lines of code. A MoDisco Java metamodel and a corresponding model discoverer are used to reverse engineer the input Java application, model transformations are applied on the initial models obtained by the model discoverers to restructure the system, and then a MoDisco Java generator is used to generate refactored code from the model.

Approaches A4 by Trias et al. [89] from 2014, and A19 by Moutaouakkil and Mbarki [67] from 2020 both present MDA inspired approaches that represent a so-called Architecture-Driven Modernization (ADM) style that aims to standardize and automate the re-engineering processes for modernizing legacy web applications. Both approaches target systems built with PHP and use automatic tools to build KDM metamodels to represent the legacy code as Abstract Syntax Trees (AST)<sup>4</sup>, allowing model-based manipulation on the rep-

<sup>4</sup>Abstract Syntax Tree (AST), a data structure representing the structure of a program.

resentation of the system and generating new representations of the system from the models. In the A4 approach by Trias et al., the applications are migrated to a Content Management System (CMS)<sup>5</sup>, and the presented toolkit contains a PHP modeling language, model-to-model transformation rules to generate the KDM models, and support for forward-engineering the models to text, and an implementation of the approach to migrating a Drupal widget to Wordpress. The later study, approach A19 by Moutaouakkil and Mbarki, replaces the PHP language definition step of the A4 approach by glayzzle<sup>6</sup> PHP source code parser tool to save effort in the AST extraction step during the model discovery phase. The A19 approach also takes the approach taken with the MoDisco framework a step further by creating generic ASTM models from the specific ASTM models presented in the MoDisco approach. This is achieved via defining an additional model-to-model generation step that uses the QVT-operational language<sup>7</sup> to generify the specific ASTM to a generic one in the model understanding phase.

The A10 approach by Rodríguez-Echeverría et al. targets modernizing legacy web applications that have been developed in an ad hoc manner, meaning that model-driven engineering approaches have not been utilized in the development phase of the legacy system. [76] The approach targets a generic framework for building modernized RIA-styled web clients from the combination of legacy web application presentation, server-side business logic, and the service layer connecting the two. The approach is a specialized scenario for the generic MoDisco framework, and as such, the first step of the approach is to create language-dependent initial models representing the entirety of the legacy system, by using the existing model discoverers. The initial models focus on extracting exact information of the UI layout, web page and data relationships, as well as the navigational and operational maps of the application. These models are used to derive a technology-independent conceptual specification of the legacy system, namely KDM, to be refined into pattern expressions that are used to manually restructure the KDM. Rodríguez-Echeverría et al. use QVT for pattern recognition, but report low precision for the tool, which is improved in a later study [35]. They propose an additional step of projecting the derived models on RIA-extended Model-Driven Web Engineering methodologies<sup>8</sup> models before code generation to facilitate more convenient and accurate generation of final views, which is named as the last step in the modernization process. The group has also adapted the framework to modernize MVC<sup>9</sup>-based legacy web applications to

---

<sup>5</sup>Content Management Systems (CMS), such as Wordpress, are software systems used to create, modify and otherwise manage digital content, in relation to, for example, a given organization, usually in a collaborative manner.

<sup>6</sup><https://php-parser.glayzzle.com/>

<sup>7</sup>QVT by OMG, <http://www.omg.org/spec/QVT>

<sup>8</sup>Model-Driven Web Engineering (MDWE) refers to applying the practices of Model-Driven Software Engineering in the context of web applications, that is, advocating models as the key development artifact in all phases of the development of a web application. The approach drives separation of presenting information of the system from its technical implementation [68]

<sup>9</sup>The Model-View-Controller (MVC) architectural pattern organizes application functionality around the

Service-Oriented Architecture<sup>10</sup>-based web services in a later study. [85]

In contrast, in the approach A21, Garzón et al. [44] present a framework called Umple for Model-Driven Development (MDD)<sup>11</sup> and MDRE to help developers adopt the model-driven approach. The Umple framework represents a model-is-the-code approach where instead of discovering models from existing legacy systems, the approach primarily encourages defining the software system as an integration of models and algorithmic code, and generating the system from the models, although extracting Umple models from existing systems and modernizing the systems are also supported use-cases. Umple tools allow generating base language code from the Umple models, so the approach indeed covers the complete re-engineering process. For reverse-engineering, the framework provides a parser, model extractor and transformer tools that incrementally convert an existing system to Umple models. The reverse-engineering component parses a source system to an AST, and uses a base-language metamodel to extract a base-language model of the system. The base-language model is mapped to an Umple model, to be validated for Umple code generation. Any Umple modeled system can then be forward-engineered to base code such as Java or C++ with the included parser, analyzer and code-generator components. The Umple code is tokenized by the parser, processed to an Umple metamodel representation by the analyzer, and then translated to source code or diagrams by generator components.

In approach A25, Katsimpa et al. present the tools and techniques required to extract conceptual model representations of existing web applications. [52] Their approach targets web applications implemented with the ASP.NET OSS framework, and the extracted models are presented as WebML[33] conceptual schemas, to facilitate code reuse and the maintenance of the application. The approach focuses on the reverse engineering phase of the re-engineering process and does not particularly comment on how the models can then be used during forward engineering. Similarly, the approach A35 from Sacramento and Paiva focuses on reverse engineering. [79] They automatically extract testable UI models from existing web applications, but do not produce models of the system with the goal of facilitating transformations or modernizations on the system itself. The constructed models are meant to allow Pattern-Based Graphical User Interface Testing<sup>12</sup> of the software system. Therefore, while both approaches demonstrate useful use cases for

---

concepts of models, views and controllers. The model refers to the definition of application data, view presents the data and the controller represents the business logic orchestrating application functionality.

<sup>10</sup>The service oriented manifesto defines Service-Oriented Architecture (SOA) as the "type of architecture that results from applying service orientation", viewing business activities from a point of view that prioritizes a set of activities that help an organization create business value [83]

<sup>11</sup>In Model-Driven Development (MDD) is a software development methodology aimed at developing techniques to represent and develop software systems as models. [44]

<sup>12</sup>Pattern-based GUI Testing (PBGT) is an approach to testing the GUI of a software system where recurring behavioral patterns of the UI are recognized by building and analyzing models of the behaviors of the UI, in order to construct testing strategies for common UI behaviours, despite the underlying implementations varying. [79]

model-driven reverse engineering of web applications, neither approach present a solution for modernizing the applications.

In addition to the works recognized by Siala et al. as web-specific approaches, Garcés et al. present an MDE based approach for generic legacy application modernizations. [43] In their approach, they aim to maintain the quality attributes of the modernized system by configuring the architecture of the system on the model level, before generating a modernized code version of the system. The approach is generic and the produced architectural UML or OCL<sup>13</sup> models are independent of the implementation technology of the legacy system. They use AgileUML to translate the system to the target language, for example, Java, C#, or Python. Despite the approach extracting UI components, it does not include extracting the layout of the legacy system UI, but rather, generates the modernized target application with a default layout. Similar to the MoDisco approach, this approach follows a common technique in MDE where the first step of the modernization produces Technology Specific Models (TSM) of the system that allow the system to be manipulated in the modeling realm. Then, the TSMs are transformed into Technology Agnostic Metamodels (TAM) during the transformation step of the re-engineering process, and then combine an architecture configuration with the metamodel to facilitate the model-to-text transformation of generating code. The approach is demonstrated with a case study on Oracle Forms applications, but the authors believe their approach to be adaptable for other technologies as well that support the client-server architecture. They anticipate this could happen by implementing a new text-to-model parser for creating the TSM from the legacy system, and a new model-to-model parser to translate the TSM to a TAM, while maintaining the forward-engineering steps the same.

In 2024, Lano and Siala also published their own MDRE approach to program migrations from one language to another. [57] In the approach, they focus on the reverse-engineering step and extract semantic models in UML and OCL from the source application. These models can then be used to forward engineer the system in another language using existing MDE forward engineering tools for UML and OCL models, like Garcés et al. [43] do with AgileUML. In the approach by Lano and Siala, an AST is parsed first from the source system with ANTLR v4<sup>14</sup>, and CSTL scripting language is then used on the AST to abstract the origin program constructs as UML/OCL representations. The abstraction process aims to maintain semantic correctness of the source system by defining exact abstraction rules in the CSTL language to capture the semantics of the source system in the model representation of the program. However, UI elements of a software system and

---

<sup>13</sup>Object Constraint Language (OCL) is a language used to express precise rules that apply to models defined in UML [57]

<sup>14</sup>ANTLR (ANother Tool for Language Recognition) is a tool used to generate programming language parsers for constructing ASTs. [27] A language parser is generated by using a separately defined programming language grammars, collected in the grammars-v4 GitHub repository. [28] The collection contains contributions for over 200 language grammars, Java and JavaScript among them.

asynchronous operation invocations are not currently modeled. Abstraction mappings are provided from Java, Python, and JavaScript to mention a few. The semantic models are then fed to AgileUML code generator to generate target language code. Lano and Siala verified their approach via translating code snippets from multiple languages to a set of target languages, and compared runtime equivalence and computational accuracy to numbers given by other approaches, and received promising results in terms of translation accuracy. They also measured the completeness of the source language coverage and how adaptable the approach is for alternative CSTL-based translation policies, and concluded that the approach seems productive in comparison with manual translation.

In summary, several MDRE approaches to different re-engineering problems have been developed during the past few decades. The approaches may focus on different phases of the re-engineering process. Modeling based approaches are often more or less generic, although Java is a commonly supported source and target language for the approaches. Many of the approaches involve automatically parsing the source system into an AST, and performing varying model-to-model transformations before using tools to generate code.

### **3.2 Applications of artificial intelligence in modernizations**

As recognized by e.g., Lano and Siala, Artificial intelligence and its applications have also been applied in re-engineering in recent years [57] and thus, its applications in modernizations deserve a discussion in this thesis as well. This section presents some of the existing approaches to applying artificial intelligence in modernizations, and the characteristics of these approaches.

In 2021, Somogoi and Kovesdan [84] argued that the two most important properties of software modernization tools are a) maximum level of automation in the conversion, and b) conversion precision. Moreover, they assert that with approaches based on static analysis of the origin code base, usually only one of the two properties can be achieved at the same time: An increase in conversion automation usually results in lower-quality code, but on the other hand, increasing human intervention during conversion usually improves the quality of the resulting code but decreases the level of automation.

To address this issue, Somogoi and Kovesdan propose a modernization method that uses Machine Learning (ML)<sup>15</sup> techniques to achieve an automated, but natural translation. To illustrate the conversion precision benefits of ML-devised automatic conversion, they perform a case study on transforming C pointers to Java variables, which represents a problem in which the origin structure has more than one syntactically correct translation in the target language. In their approach, they teach a deep learning (DL) algorithm to make classification decisions based on a relatively small sample of manually collected

---

<sup>15</sup>Machine Learning (ML) is a field that studies applying statistical algorithms to recognize patterns and learn from data. [107]

training data. The data determine training vectors that signify whether an attribute should be converted to an array or a variable and is used to train and validate a neural network. They report an increased conversion precision for their ML approach compared to an approach based on static analysis of the source code.

While the classification decision making itself is an automated process, Somogoi and Kovesdan themselves point out, that the approach requires a significant amount of work in data collection and training the ML algorithm. Thus, applying their solution to every conversion decision during a modernization process is not feasible.

Despite it is not explicitly mentioned in their work, Somogoi and Kovesdan represent a field of research centering around modeling source code by the means of ML and DL in particular, called Big Code, as defined by Triet et al. [58]. The existence of this research area is based on the suitability of ML and DL approaches to analyze and process natural languages [58], as these techniques capture statistical patterns hidden in large data sets [63]. In principle, source code is its own kind of specially structured natural language, which makes DL approaches applicable in analyzing and transforming it. [58] Therefore, if suitable training set is available, ML-based tools can learn the statistical alignments between two high-level programming languages, which can be used to create translation tools. [63] What sets DL approaches apart, is that DL considers both syntactic and semantic information, which allows capturing long-term dependencies from source code [58]. An example of a long-term dependency could be making sure a database connection opened in the beginning of a function, is always closed, even if failures occur during the function execution. According to Somogoi and Kovesdan, capturing long-term dependencies is considered a difficult task to achieve with traditional means of program analysis, without at least human intervention. [84] However, in their approach, Somogoi and Kovesdan demonstrate a common feature of neural program models, which also Triet et al. point out, that most source code models are constrained to very specific tasks and perform well merely on problems similar to its training data [58]. They explain that designing a pure AI-approach to synthesize complete programs is difficult, even though AI approaches can certainly be utilized in solving subproblems of source code modeling.

In their work, Triet et al. categorize Big Code tasks according to the format of the task input and output, cover the state-of-the-art of DL models and how the DL models can be applied to code modeling and generation tasks. This gives an overview of DL models and how they apply to different tasks. In their taxonomy, Big Code tasks are categorized into source code analysis or source code generation tasks. Code migration is considered a generation problem. More specifically, Triet et al. categorize program migrations as type of a program transduction task. Program transductions are approaches that aim to convert source code from one form to another, for example, porting projects from a language to another, or upgrading the language or framework version of a code base. As demonstration, Triet et al. mention the works of Aggarwal et al. [3] and Gu et al.

[46]. Aggarwal et al. modernize Python projects from Python version 2 to Python 3 using statistical machine translation, and Gu et al. migrate a Java API to C# using sequence to sequence learning, namely a seq2seq model called DEEPAM<sup>16</sup>.

Previously discussed approaches represent ML techniques called supervised learning. Supervised learning refers to techniques that utilize a pattern recognition machine that requires a set of labeled training data and a separate training phase for the learning algorithm, during which an algorithm learns by example to identify the correct answer for a given classification problem. [106] When trained with a proper data set, supervised learning produces accurate translations, but its main constraint is acquiring large enough correctly labeled data sets. [63] The training data is usually acquired via laborious manual classification, but in the program translation context, some approaches have been presented to improve translation accuracy via back-translation. [56] In contrast to supervised learning, unsupervised learning refers to decision machines that estimate classifications and the number of classifications in data sets, where no data points have been classified beforehand. [106] In other words, unsupervised learning algorithms identify previously unknown patterns from unlabeled data. When applied in the context of high-level programming language translations, unsupervised learning can produce more versatile and scalable translations than supervised learning or rule-based solutions, although it lacks translation accuracy. [56]

However, there is ongoing research into improving the translation accuracy. Malyala et al. [63] study unsupervised machine learning based program translations from legacy applications to modern languages to dig deeper into the patterns where unsupervised program translators fail. They identify both semantic and syntactic translation errors, such as the model's tendency to add extraneous logical constraints in the translation. To address these, they propose a hybrid transpiler tool that combines ML-based program translator with a program mutation engine. This approach adds rule-based pre and post processing steps to the translation pipeline to address the syntactic and semantic errors, respectively. The approach shows promising initial results for enhancing the fast, yet error-prone statistical conversion with program analysis based domain-knowledge. Lachaux et al. also discuss improving the translation accuracy in unsupervised learning using TransCoder machine translator model, and conclude that the approach produces viable unsupervised machine translation that is generalizable to any programming language, although the approach is not mistake-free. [56]. Common to the works of Lachaux et al., and Malyala et al., is that the transpilation scope is still limited to individual functions.

In 2024, Siala [81] attempted to integrate ML to MDRE-based software modernizations

---

<sup>16</sup>DEEPAM (Deep API Migration) uses a large scale corpus of code projects to mine for API mappings, or, in other words, apply non-manual methods to find equivalent API functions between different programming languages or APIs. In the case of DEEPAM, the approach learns the semantic sequences from both the source and target APIs to identify related sequences and thus, reveal corresponding API functionality. [46]

using Large Language Models (LLM)<sup>17</sup>. The approach focuses on making the model extraction phase in MDRE more efficient with an automated program specification extraction method and a related tool to support the extraction. Like in Siala's work with Lano [57], the target representation is UML with OCR constraints, but instead of traditional methods of source code analysis, this approach uses LLMs as the method of extracting models from legacy software. Similar to the conclusion made by Malyala et al. [63], Siala predicts the LLM approach to outperform the state-of-art MDRE approaches, although the research is still work-in-progress. Cámara et al. also investigate applying LLMs to software modeling tasks. [32] In their approach, Cámara et al. utilize ChatGPT in the role of an modeling assistant and investigate how to best prompt ChatGPT and evaluate the correctness of the UML models ChatGPT produces, and the modeling concept coverage of the produced models. They also evaluate the expressiveness and "cross-modeling language translation capabilities" and ChatGPT's sensitivity to context. Cámara et al. conclude that ChatGPT is prone to produce errors, except domains with large code bases, performs better for code generation tasks than modeling, can handle only small models and is unable to correctly handle some basic concepts of modeling, like multiple inheritance. Notably, ChatGPT performed well on OCL constraints, which the authors contribute to OCL resembling closely SLQ, for which there is extensive code base used to train the LLM.

In this chapter, we summarized some of the several approaches for AI-assisted program translations. Some approaches focus on converting code snippets such as individual functions to provide natural translations. Others mine mappings between APIs, and some applications have been adopted also for MDRE. Techniques cover sequence-to-sequence learning, unsupervised and supervised learning, or even LLMs. An occurring theme seems to be unsatisfactory translation accuracy or scalability of the approaches, although especially the MML-based approaches prove versatile, although inconsistent and difficult to reproduce.

### 3.3 Dedicated parser for migrating AngularJS to Angular

In addition to the generic, technology agnostic approaches discussed in subsections 3.1 and 3.2, one approach to program migration is developing specialized tools that perform conversions from a specific platform or language to another. This can be a useful approach, for example, when semantic and syntactic similarities between the source and target languages allow for implementing the conversion via simple text-replacement. Such approaches may utilize a technique this thesis calls Automatic Syntax Replacement (ASR). An ASR approach was adopted, for example, for optimizing the performance of Python programs by automatically converting programs to almost equivalent Rust [61]

---

<sup>17</sup>Large Language Models (LLM) are ML-models trained to catch statistical patterns and structures in a vast amount of textual training data, which offers a way to understand and generate human-like output. [78]

by visiting the nodes of a Python program AST and replacing Python syntax with Rust syntax.

### **3.3.1 The official migration approach from AngularJS to Angular 2.0+**

Unlike many of the approaches presented in subsections 3.1 and 3.2, the official approach to migrating AngularJS applications to its successor, Angular, aka. Angular version 2.0 or later, is not supported by automation. The intended migration process is described in a migration guide [93], provided in the documentation of the Angular framework. This approach is described in detail in chapter 4.2.2, but in a nutshell, the Angular way of migrating is performed in two manual steps: preparation and migration. During preparation, the implementation of the AngularJS application is manually aligned with a generic Angular application structure. The actual migration is then performed in a subsequent migration step, by manually upgrading the application code one feature at a time. Incremental upgrade is made possible by using the Angular library called `ngUpgrade`, which runs both versions of the framework in parallel and bridges the communication between components implemented in different framework versions. [93].

### **3.3.2 angularjs-to-angular by Grubhub**

However, once the movement to migrate existing AngularJS applications had begun, the community of developers who had built applications with AngularJS developed their own approaches to performing the migration. One approach resulted in a dedicated conversion tool, `angularjs-to-angular`[23], presented by Eric Tsai and Sung Choi, while working on the engineering team of the Grubhub food delivery company. The approach is summarized in a series of four posts published on the GrubHub Bytes platform [47, 18, 29, 88], and the related tooling was made publicly available on GitHub [23]. The approach is discussed in detail in chapter 4, but in summary, the same application preparation step as in the official migration path is adopted, but instead of full manual incremental conversion in the migration step, the `angularjs-to-angular` tool assists in the conversion. The tool builds an AST of the prepared application, implements a series of normalizations and AST manipulations, and then complements the manipulation by pure regex-based syntax replacement and other manipulations, before writing the converted code to disk. However, manual conversion work is still required, as the approach only accounts for converting components, services, and template files, but does not handle routing, modules, application bootstrapping or unit tests completely, as discussed later, in chapter 5.

## 4. TRANSPILING ANGULARJS TO ANGULAR WITH A DEDICATED PARSER

The chapter 3 briefly discussed dedicated parser tools as a means of modernizing applications, and mentioned `angularjs-to-angular` in particular as an example of such a tool. In this chapter, we describe a methodology for verifying whether the tool can be successfully used to modernize an existing AngularJS application and what the modernization steps are when implementing the approach. To this end, the `angularjs-to-angular` modernization approach is implemented for converting a test application to Angular, and the result of the conversion is compared to another modernization of the same application, produced by manual conversion. The experiment is implemented to gain insight into the approach and its viability. This chapter describes the experiment, and the results of the experiment are discussed in chapter 5.

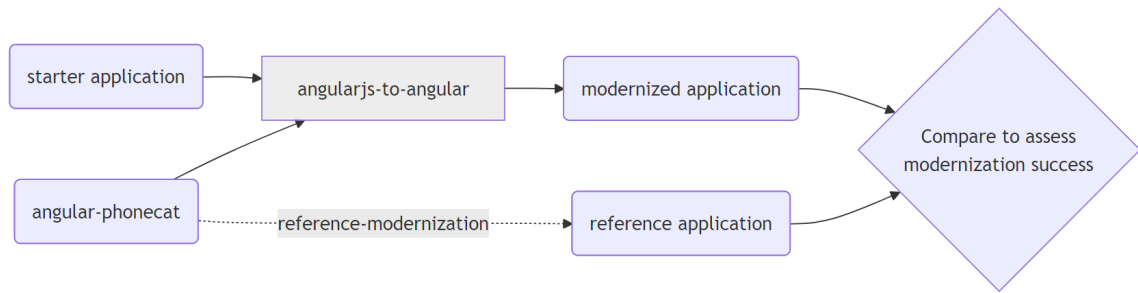
### 4.1 Modernization approach overview

An overview of the modernization pilot experiment is presented as a process chart in figure 4.1. The starting point for the modernization consists of three main parts: a source application to upgrade, the transpilation tool, and a minimal starter application, which serves as the foundation for the modernized application. A reference modernization of the same source application is produced manually following the AngularJS to Angular migration guide [93] to validate the modernization approach and the modernized result. More on reference modernization in chapter 4.2.2.

The details of the modernization experiment are discussed in the following sections as follows: the source application to-be-modernized is described in section 4.2.1 to give the reader an understanding of the application that is converted to Angular. Then, section 4.3 gives an overview of the `angularjs-to-angular` tool that is used in the experiment to produce a modernization of the test application. The actual modernization process utilizing the `angularjs-to-angular` tool is described in chapter 5, including a description of the role of a starter application<sup>1</sup>, in section 5.2. The reference modernization used to validate the conversion result is summarized in section 4.2.2.

---

<sup>1</sup>Using a starter or seed application is the officially recommended way to start building a new Angular 6 application [8], so this approach is adopted for the modernization approach in this thesis as well



**Figure 4.1.** In the modernization experiment, *angular-phonecat* source application is modernized via two methods to produce a modernized application and a reference for it. The resulted applications are compared to assess whether the selected modernization approach produces a valid modernization. The chart is created with Mermaid Live Editor.

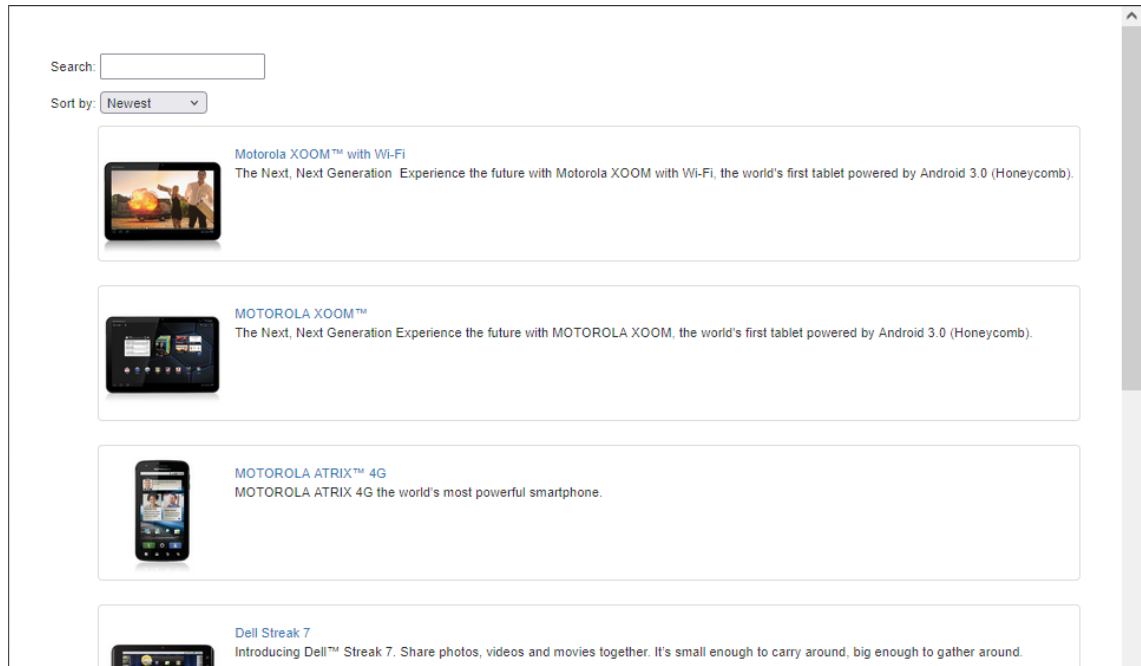
## 4.2 Selecting a test application for the modernization test

The application selected to test the approach is the *angular-phonecat* application [20] by the AngularJS team. The application is a tutorial application, built piece by piece in the AngularJS documentation [26] to demonstrate the features of the AngularJS framework. The application is a perfect candidate for a conversion test because it is a minimal example of an AngularJS application that incorporates the main features of the framework and follows the common code writing conventions typical to the technology. The *angular-phonecat* application has a permissive license [19], and its codebase is publicly available in the GitHub archive [20]. The application is also already familiar to many, who have worked with the framework before. As an additional bonus, the Angular team also provides their own modernization approach that uses the *angular-phonecat* application as an example application to modernize to Angular. This provides a valid reference modernization that can be used to assess the *angularjs-to-angular* approach and the modernization result. The *angular-phonecat* application is introduced in more detail in subsection 4.2.1 and the reference modernization in subsection 4.2.2.

The modernization pilot experiment is implemented with version 1.5-snapshot, commit 0970e05 of the *angular-phonecat* application [21]. This matches the application version in step 14 of the AngularJS tutorials [26], which is also the last step of the tutorials. This version of the *angular-phonecat* application is written in version 1.5 of the AngularJS framework, and the source files for *angular-phonecat* at 1.5-snapshot are available in the GitHub public archive [21].

### 4.2.1 The source application: *angular-phonecat*

To familiarize the source-application-to-modernize, this section briefly describes it from three points of view. The first point of view is the User Interface (UI); How the application presents itself to the user and what the user can do with the application. Then, we de-



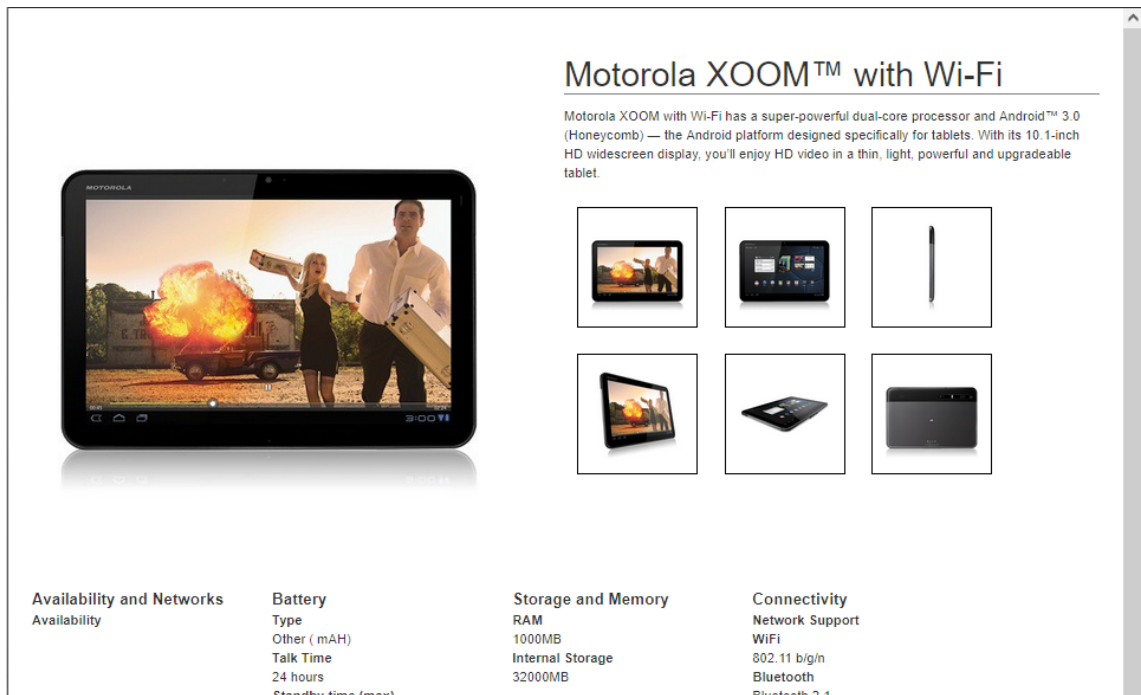
**Figure 4.2.** A screen capture of the landing view of the angular-phonecat application version 1.5-snapshot, displaying a listing of the items cataloged in the application.

scribe how the application is built with AngularJS features and gain an understanding of how the UI views and functionality translate to AngularJS concepts. Lastly, the incorporated AngularJS concepts are mapped to the file structure of the application source code to better demonstrate exactly what source code is translated during modernization. For convenience, the files to translate are summarized in a table in the end of this chapter.

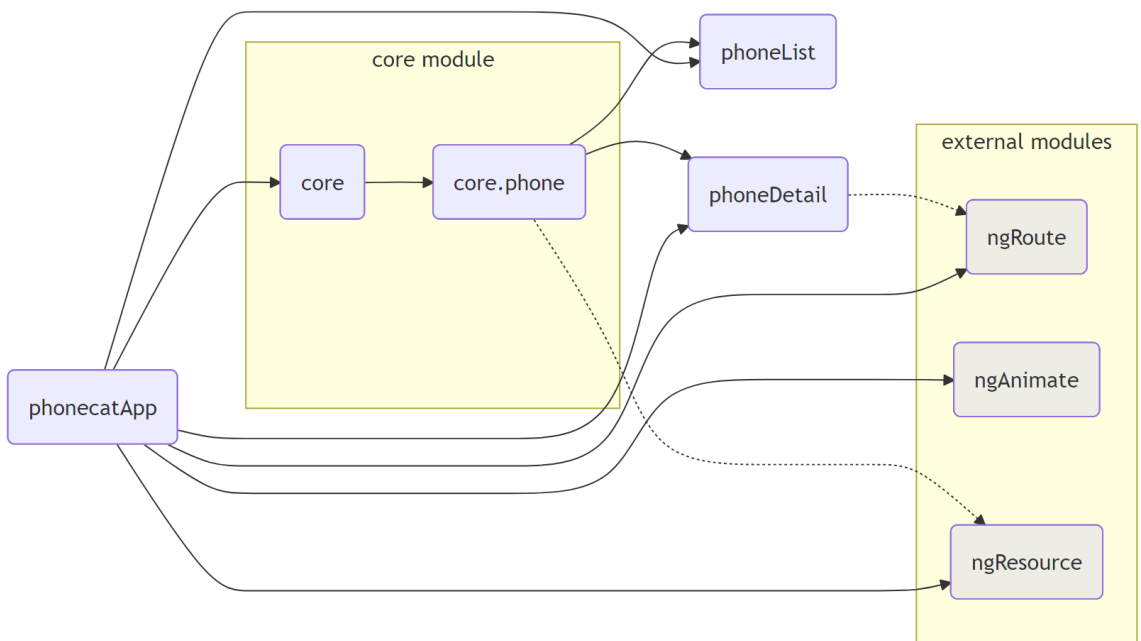
The angular-phonecat tutorial application is a simple example web application used to display a catalog of mobile devices. From the UI perspective, it is constructed from two views: a landing view and a detail view. The landing view, shown in figure 4.2, displays a listing of the cataloged items. The landing view is also called the phone list view, as the landing view contains a list of phones. The detail view, shown in figure 4.3, displays the details of an item selected from the list in the landing view. From a functionality point of view, the landing view also includes a text input element that allows the user to filter the listed items by custom search words, and a drop-down menu that allows the user to change the order of the phone list.

The application code structure is typical to an AngularJS application: it consists of a main application module, which employs a set of feature modules. The feature modules encapsulate component definitions and services to package the needed code for displaying a feature and provide the necessary business logic to provide the functionality specific to the given feature.

More specifically, the main module of the angular-phonecat application is `phonecatApp`. The `phonecatApp` module employs seven other modules that are used to construct the



**Figure 4.3.** A screen capture of the detail view of a selected item, as displayed in the angular-phonecat application.



**Figure 4.4.** The module structure of the angular-phonecat application. The redundant submodule dependencies from core.phone to ngResource and from phoneDetail to ngRoute are drawn with dotted line.

application: `ngAnimate`, `ngRoute`, `ngResource`, `core`, `phoneDetail`, `phoneList`, and `core.phone`. See figure 4.4 for a visual presentation of the module hierarchy. The modules prefixed with 'ng', namely `ngRoute`, `ngResource`, and `ngAnimate`, are AngularJS-provided modules distributed separately from the built-in AngularJS core module `ng`. The `ngRoute` module allows for configuring deep links<sup>2</sup> between the phone list and phone details views, whereas the `ngResource` is used to build a CRUD client for the application [25]. The `ngAnimate` module is used to define animations by hooking into DOM events with CSS. The `core`, `core.phone`, `phoneDetail`, and `phoneList` modules are custom modules defined in the source code of the `angular-phonecat` application. The `phoneList` module is responsible for rendering the landing view seen in figure 4.2, and the `phoneDetail` displays the phone detail view shown in figure 4.3. The `core` module defines the functionality and concepts that are common to different parts of the application. In this case, the common functionality contains a `core.phone` module and a checkmark filter, and these are shared with other modules of the application via the dependency injection (DI) [7] system of AngularJS. In this case, the feature modules `phoneDetail` and `phoneList` are interested in the `core.phone` module, so the module is injected to those feature modules as a dependency, which allows the model of a phone to be shared and updated in the `phoneDetail` and `phoneList` modules and therefore displayed and updated in both the landing and the phone detail views.

The modules discussed above construct the application. These are also the parts of the application code that are modernized during the pilot experiment. In terms of source code, these modules are defined in the `app` directory of the `angular-phonecat` code repository. Figure 4.5 lists the contents of the `app` directory as a file tree to demonstrate the original file structure of the source application to transpile. The directory contains 120 files, out of which the application definition is contained in 20 files of JavaScript, HTML, and CSS, and the other 100 files define application data. Namely, there is 79 product images as JPG files and 21 product data files in JSON. The `.spec.js` files found in the directory, are for unit tests defined with the Jasmine [50] test framework and Karma test runner [51]. In addition to unit tests, the `angular-phonecat` repository also contains a set of end-to-end tests written for Protractor[74] defined in `angular-phonecat/e2e-tests/scenarios.js`. The handling of unit tests and end-to-end tests is excluded from the modernization experiment to limit the scope of the experiment. Despite tests being a useful and closely related part of application development, they are meaningless for the application runtime and thus are not required to be converted for the application to start and run normally. Lastly, there are external dependencies to Bootstrap and jQuery as seen by style references in the `phone-detail` and `phone-list` template files. These external libraries should be preserved through the modernization process.

---

<sup>2</sup>Deep linking refers to using links to refer a specific, searchable parts of the application, the phone detail and landing views in the case of the `angular-phonecat` application [24]

```

.../angular-phonecat/app
├─ app.animations.css
├─ app.animations.js
├─ app.config.js
├─ app.css
├─ app.module.js ----- definition for the phonecatApp module
├─ core ----- common components & services
│  └─ checkmark
│     └─ checkmark.filter.js
│        └─ checkmark.filter.spec.js
│  └─ core.module.js
│     └─ phone
│        └─ phone.module.js
│           └─ phone.service.js
│              └─ phone.service.spec.js
├─ img
│  └─ phones ----- application image data: 79 product images
│     └─ dell-streak-7.0.jpg
│        └─ ...
├─ index.html ----- application entry point
├─ phone-detail ----- phone-detail feature definition
│  └─ phone-detail.component.js
│  └─ phone-detail.component.spec.js
│  └─ phone-detail.module.js
│  └─ phone-detail.template.html
├─ phone-list ----- phone-list feature definition
│  └─ phone-list.component.js
│  └─ phone-list.component.spec.js
│  └─ phone-list.module.js
│  └─ phone-list.template.html
└─ phones ----- application json data: 21 phone detail files
   └─ dell-streak-7.json
      └─ ...

```

**Figure 4.5.** An annotated file tree representation of the angular-phonecat application structure. The file tree representation is a 3-level printout from the tree-cli tool with compressed data folders and added explanations for the different parts of the application. <https://www.npmjs.com/package/tree-cli>

In summary, the files to modernize are listed by AngularJS feature in table 4.1. There are five custom modules, two component files, one service, two templates, a filter, and four test files, as well as five uncategorized files. The four uncategorized files are used to define animations (app.animations.js and app.animations.css), configure the service provider for the router service from the ngRoute module (app.config.js), and define CSS styles for the application (app.js). While custom animations written in JS and service configurations are modernized during the conversion, pure CSS is left untouched, since the DOM styling technology is independent of how the application is defined. The JPG and JSON files that define the application data are omitted from the table because the image data will remain intact during the transformation, and the JSON files are there to emulate server responses to data requests. Therefore, these files are not modified

| AngularJS feature | File extension | Definition file   |
|-------------------|----------------|---|
| module            | .module.js     | app, core, core.phone, phone-detail, phone-list                               |
| component         | .component.js  | phone-detail, phone-list  |
| service           | .service.js    | phone   |
| template          | .template.html | phone-detail, phone-list, index.html  |
| filter            | .filter.js     | checkmark   |
| unit tests        | .spec.js       | checkmark.filter, phone.service, phone-detail.component, phone-list.component |
| other             | varying        | app.animations.css, app.animations.js, app.config.js, app.css                 |

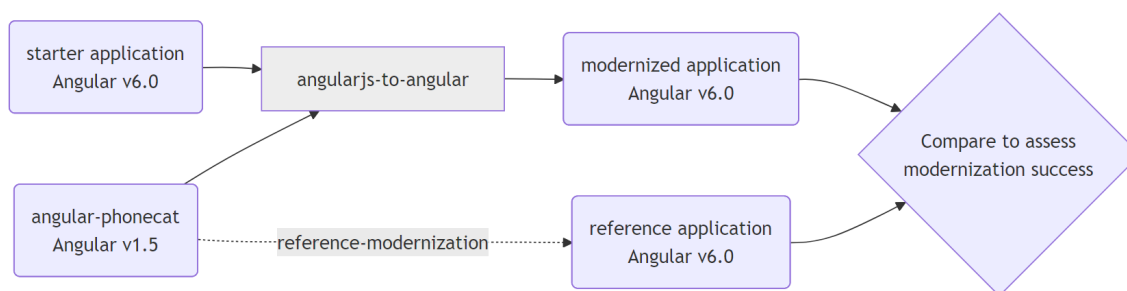
**Table 4.1.** *The angular-phonecat application features to modernize and the files that contain the definitions for the features, categorized by AngularJS feature type. Application data files are omitted for clarity.*

during the translation; how the JSON data is served over HTTP is independent of how the phonecat application is modernized, as AngularJS is a pure frontend framework. Neither the ng, ngRoute, ngResource, and ngAnimate modules are listed, since these are not defined as part of the application custom code, and thus the translation of the framework's own modules depends on there being a modernized version available of the modules in the target version of the framework.

#### 4.2.2 A reference modernization for the angular-phonecat application

As mentioned in chapter 4.2, one argument for choosing angular-phonecat as the source application to modernize during the pilot is that there are instructions available for how to manually produce a reference modernization of the application. This chapter describes how the reference modernization was produced, so that it can be later utilized in assessing the semi-automated modernization approach described in chapter 5.

The reference modernization was constructed by following the guidelines given in an Angular upgrade guide [94], provided in the Angular documentation. The upgrade guide describes the modernization approach that the Angular team recommends to adopt when upgrading AngularJS applications to Angular. To demonstrate the approach, the guide



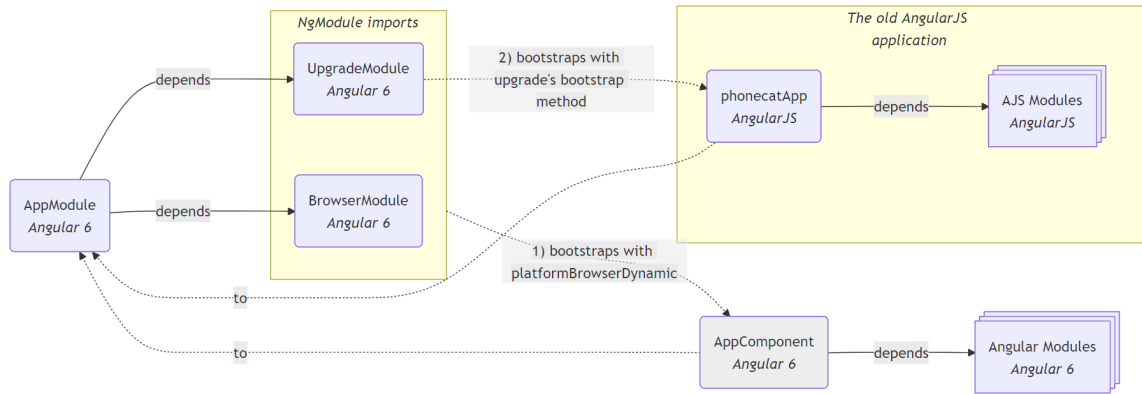
**Figure 4.6.** During the modernization experiment, the framework version of the angular-phonecat source application is modernized from Angular version 1.5 to version 6 with two different methods. This section describes the reference modernization.

uses the angular-phonecat application, version 1.5-snapshot, as an example and walks through the steps required to modernize the application from AngularJS version 1.5 to modern Angular versions. An upgrade path is described for all Angular versions since the release of Angular 2.0 and in the context of this pilot experiment, the target version is fixed to Angular 6, as this is the latest version the angularjs-to-angular tool supports, as is discussed in section 4.3. The resulting application can then be further migrated to more recent Angular versions by following the upgrade instructions provided in the Angular documentation, but that is out of the scope of this experiment. Figure 4.6 demonstrates how the Angular Framework versions are upgraded throughout the pilot experiment.

As a limitation, the reference application source code is indeed not provided as a complete application in the upgrade guide. Instead of a ready made repository containing a modernized application, the modernization steps are provided as instructions and code snippets meant to be applied on top of the original angular-phonecat application. For this reason, this chapter briefly summarizes how the suggested modernization steps were applied on the angular-phonecat version 1.5-snapshot to produce the reference modernization in Angular 6.0, and the source files for the produced reference implementation are shared in a GitHub repository [99]. It is also worth noting that modernization instructions are not provided for animations in the guide, so animations are excluded from the reference modernization.

The reference application is built on top of the existing angular-phonecat application, so to produce the reference modernization, the original angular-phonecat repository is forked and a new git branch is created, using the branch 1.5-snapshot as the branch basis. Then, the PhoneCat Upgrade Tutorial [93] is followed. An assumption for the process is that the 1.5-snapshot version of the source application indeed conforms to the Angular style guide [17], so the application style preparation step [95] is skipped, as the application structure is already aligned to Angular practices<sup>3</sup>. However, SystemJS version

<sup>3</sup>Notably, the angular-phonecat application structure follows the rule of 1, the folders-by-feature rule and the modularity rules; Each component, controller, service and filter are contained in their own files, as seen in table 4.1, the files are arranged in a folder structure where each feature is its own directory,



**Figure 4.7.** The Angular UpgradeModule allows for creating a hybrid application that bootstraps both versions of the framework simultaneously, and enables the intercommunication of AngularJS and Angular components, modules and services within the same application.

0.19.40 is introduced as a module loader[96] for the reference modernization to align the reference application with the capabilities of the angularjs-to-angular tool, as discussed in 4.3. TypeScript and related typing libraries are installed for AngularJS, and the TypeScript compiler is configured according to the default setup recommended in the setup documentation [15]. AngularJS related package versions and configuration options are selected to reflect the most recent versions available at the time of the last update to the Angular 6 upgrade guide in October 2018. For Angular related libraries and configurations, version 6.0.0 releases are selected to match the version 6 framework release. Necessary compile and start scripts are saved in package.json, as intended for a project using NPM as the package manager. Notably, the angular-phonecat application manages some packages also with Bower. For a modernized application, it makes sense to eventually remove the redundant package manager. However, instructions for removing Bower are not provided in the upgrade guide, and so, Bower is left in the reference modernization. Similarly, the ahead-of-time compilation capabilities are deemed out-of-scope and left out of the reference modernization.

The figure 4.7 demonstrates the key mechanism to this modernization approach, where Angular 6 is brought alongside the AngularJS framework version, constituting a hybrid application. In a hybrid application, both versions of the framework function seamlessly together, allowing the application to be gradually converted from AngularJS to Angular 6, one module, service and component at a time. This is achieved via the ngUpgrade library, which allows for bootstrapping both versions of the framework in the same application, and adapting together the interfaces of features written in different versions of the framework, so the features may interoperate. Once the features are brought over to

as demonstrated in figure 4.5, and the application is composed of small modules, including a thin root module, phonecatApp, that pulls together the application modules, as demonstrated in figure 4.4. All the rules mentioned here are defined in the Angular 1 style guide [17]



tool. The automated conversion is compared with the reference conversion in chapter 6. This allows for validation of the modernization approach, as described in more detail in chapter 6.

### 4.3 The modernization tool: angularjs-to-angular

The transpilation tool, `angularjs-to-angular`, is a Node.js [70] based command-line interface (CLI) to a JavaScript library of processing utilities for different AngularJS file types. The basic workflow with the tool is to specify a set of files to convert, and a processing utility to use on the files. This is done by running the tool from command-line with CLI arguments to specify a file pattern and a conversion utility to use on the files. The tool then reads, converts and writes the converted files into a specified target folder. The most convenient style is to set the target folder to point at a ready made, empty starter project, because the tool preserves the file hierarchy of the source files when writing the converted files to disk. For conversion, one may link the transpiler to the source AngularJS project to convert via npm links, run the tool from the root of the source application, and specify the files-to-convert as relative paths to the directory where the tool is run from.

Table 4.2 summarizes the different processing utilities available in the tool. The `angularjs-to-angular` utilities do not provide a comprehensive toolkit for modernizing every aspect of applications written in AngularJS, but offer a collection of useful utilities that automate parts of the conversion. For instance, the tool offers a utility for converting AngularJS component and controller classes to corresponding Angular 6 components, but the tool does not provide utilities for upgrading the application bootstrapping. In other words, the developer using the tool must provide a working application body that bootstraps the Angular 6 version of the framework, to which the upgraded source files are then integrated. Additionally, the original `angularjs-to-angular` tool makes several assumptions about the format of the definition of the AngularJS source files. For example, it makes the same assumption as the Angular upgrade guide that the implementation of the AngularJS source application to upgrade is already aligned with Angular practices. The rest of the assumptions are discussed in chapter 5.1, which discusses the preparation steps for the source application to convert.

The basic principle of conversion is the same for all of the existing processing utilities; a utility first parses the code file into an abstract syntax tree, which it then uses to identify different parts of the AngularJS entities defined in a given file. The identified parts are then used to recreate the corresponding Angular 6 entities. The entities are normalized<sup>4</sup> and parsed into a string representation, after which regular expression-based string

---

<sup>4</sup>Normalization in this context means that the order of the different parts of the entity definition is the same across normalized files. For example, a converted component file would first define imports, variables, and interface definitions, then enums and decorators, and lastly, the class definitions.

| Processing utility       | CLI argument, alias | Explanation  |
|--------------------------|---------------------|--|
| process-components.js    | –components, -c     | Convert AngularJS component files to Angular 6 component class files with correct decorators. Assume the component and its controller are declared as classes in the file-to-convert, so that the AST parser can extract component properties. |
| process-templates.js     | –templates, -t      | Convert AngularJS HTML template files to Angular 6 format. This utility requires specifying both the template file and the component using the template to correctly convert the template file.  |
| process-services.js      | –services, -s       | For converting AngularJS service files to corresponding Angular 6 service files.   |
| process-service-specs.js | –serviceSpecs       | Create spec file shells for unit tests. This utility is not used in the conversion pilot.  |
| process-files.js         | –copy, -y           | For copying files from one place to another. This utility is not used in the conversion pilot.   |

**Table 4.2.** Processing utilities in the original *angularjs-to-angular* tool.

replacements and modifications are performed on the string representation to convert AngularJS specific syntax to the corresponding Angular syntax. The conversion principles are discussed in more detail in the Grubhub blog series, part three, Automating the Angular conversion [29], which also identifies places where the script can be edited to adapt to the specific project structure and patterns of anyone wishing to use the script for their project modernization.

## 5. TRANSPILATION PROCESS AND RESULTS

This section describes the conversion steps when using the `angularjs-to-angular` tool to produce a modernization of the `angular-phonecat` application. As the `angularjs-to-angular` tool makes several assumptions on the format of the source application, section 5.1 first describes the different manual refactors<sup>1</sup> and other preparation steps that were implemented in the `angular-phonecat` source application required to produce a functional modernization of the application with the tool. Then section 5.2 describes the minimal starter application that is used as a basis for the upgraded version of the `angular-phonecat` application. Section 5.3 presents an overview of the conversion steps and, lastly, section 5.4 describes how the steps were executed.

### 5.1 Application preparation before conversion

To make it clear what steps exactly were performed to prepare the `angular-phonecat` for the upgrade, a separate Git repository was created [100]. This repository is called `angular-phonecat-to-upgrade`. This repository is a fork from the original `angular-phonecat` at 1.5-snapshot, so the starting point for this modernization is exactly the same as in the reference modernization. This section describes the preparation steps, which can also be followed via the commit history in the `angular-phonecat-to-upgrade` repository.

The first assumption that the `angularjs-to-angular` tool makes derives from the tool being designed to convert AngularJS applications that align with the Angular version 6 practices. These practices are detailed in the AngularJS style guide [17]. For example, the tool only knows how to parse AngularJS components defined in component files, one per file, even though in AngularJS it would be possible to define the whole application in one file. The `angularjs-to-angular` tool just would not be able to correctly identify the different parts of the application then. Of course, the `angular-phonecat` application already conforms to the AngularJS style guide [93]. Therefore, for the transpilation pilot experiment, the preparation steps in terms of code style can be skipped.

---

<sup>1</sup>Refactoring refers to implementing modifications in existing code that do not change the functionality of the code, but improve its quality. In the context of the pilot experiment described in this thesis, the `angularjs-to-angular` tool is able to extract component definition from a given source code file only if the component is defined as a class. Therefore, component definitions in the `angular-phonecat` application need to be first refactored as classes before running the conversion script on the component files.

The second assumption is that the source application should use TypeScript. The angular-phonecat version 1.5-snapshot does not yet use TypeScript. Therefore, it is a necessary preparation step to first install the TypeScript compiler and the corresponding typing libraries in the angular-phonecat application first, as well as to fix the possible compile errors that TypeScript now reveals in the application. The details of how these are done, can be found in the angular-phonecat-to-upgrade repository repository at commits e63f80b and cd687617 [100] These are the exact same steps taken in the reference modernization 4.2.2, which takes care of importing and exporting the different JavaScript modules in the application during executing. This way, all the different JavaScript files the application loads during execution, do not need to be included manually in the index.html file where they are immediately downloaded as the application site is opened. With a proper configuration, the module bundler takes care of importing and exporting the script modules as needed. An optional preparation step is to install and configure Webpack module bundler [102] in the project, this is done in commit 3b50873 in the angular-phonecat-to-upgrade repo. It becomes a mandatory step only if the source application needs to remain functional during the required refactoring steps, because some of the needed refactors rely on a module system that is not configured for the original angular-phonecat project. Of course, if the source application does not need to compile or if it is acceptable to run into runtime errors during application execution, then a module loader is not required in the source application during the refactoring phase.

The required refactors on the angular-phonecat are dictated by the assumptions that the angularjs-to-angular tool makes on the format of the source application definition. Most notably, components, component controllers, and services need to be refactored as TypeScript classes, and the templateUrl definitions need to be switched to template imports. The definition order of the different application parts in the definition files need to also be normalized. For example, the component definition files should first define imports and interfaces, then the component class, and the component controller as the last definition. In the case of angular-phonecat the \$resource service also needs to be replaced with the \$http service<sup>2</sup>. This is due to the implementation of the angularjs-to-angular tool, which supports replacing \$http service syntax with the corresponding Angular http service syntax, but does not recognize the \$resource service.

Another way to go about the problem of angularjs-to-angular tool not recognizing the \$resource service syntax would be to extend the implementation of the tool to account for the conversion of \$resource service syntax to the corresponding Angular. This way the manual refactoring from \$resource to \$http service would not be a necessary preparation step. Other syntax conversion support extensions are possible to implement as well to

---

<sup>2</sup>The \$resource and the \$http service are both AngularJS-compatible utility libraries for defining RESTful clients and they differ mostly in syntax. However, the \$resource service does not have a corresponding service in the Angular 6 version of the framework, unlike the \$http service, so the \$resource service is replaced before the conversion

avoid having to implement manual refactors. However, in the context of this thesis work, the use of the tool is studied without editing it, so the conversion assumptions are fulfilled by refactoring the source application rather than editing the conversion tool.

## 5.2 Starter application

The transpilation result is built on top of a starter application. A starter application is a minimal implementation for an Angular application that contains only the necessary parts for starting and bootstrapping the application. It does not contain any actual application logic yet but serves as a skeleton to which the upgraded application parts are attached during transpilation.

The recommended way to produce a minimal starter application is to use the `angular-cli` tool. The tool is purposed for creating, developing, testing, and maintaining Angular applications. [8, 14] Like the framework itself, the tool and starter applications generated with it come with a permissive MIT license.

The starter application used in this thesis is generated with the `angular-cli` version 6.0.0, which is the version of the tool published together with Angular version 6. The recommended way to create a new Angular application is to use the new command of the `angular-cli`, which generates a directory containing an Angular workspace [8]. An Angular workspace is a collection of logically joint applications and libraries that may share common project configurations [9, 10]. Most importantly, the workspace contains the source code files for an empty Angular version 6.0.0 skeleton application, meaning definitions for a root module, a root component, and a root template that bootstrap the framework in the application once it runs in the browser. The skeleton application also comes with default configurations for a TypeScript compiler and the workspace configuration also provides Webpack as a module loader, saving the developer the effort of setting up their own module loading configuration for each project contained in the workspace. The only content in the application is a placeholder app component template, which can be removed. The starter application used in the modernization pilot is in the first commit of the modernization application repository, available on GitHub [98].

## 5.3 The upgrade

Once the `angular-phonecat` application and the starter application have been prepared for conversion, it is time to perform the actual conversion. The first step is to configure the tool to read the input files from the `app` folder of the source application and write the upgraded files to the correct target folder in the starter application project. The source folder path is defined in the `package.json` file of the `angularjs-to-angular` project by editing the `npm` scripts there to give the correct relative path as an execution parameter

to the conversion script. Then, to write the upgraded files in the application source file folder of the starter project, a relative path is edited in the `outputRoot` variable to point to the `src/app/` folder of the starter project. The `outputRoot` configuration variable defines the root folder where the upgraded files are written to. The configuration variable is found in the `src/config.js` file of the `angularjs-to-angular` tool repository. As the tool is originally built to write the upgraded files in the tool repository, a small tweak to the `get-output-file-path.js` file help getting the upgraded files written directly to the application source definition folder in the target project: replacing the `process.cwd()` call in the `outputFileName` expression with a new `sourceDir` variable that contains the absolute path of the source folder.

The next step is to run the upgrade scripts, starting from service script, then the template script and lastly the component script. Once the source and target paths are defined as described above, one may execute the service script in the `angularjs-to-angular` tool repository root with `npm run services`. This processes the service files matching to given configuration, upgrade the files and write the upgraded files in the starter application repository, under the `src/` folder.

The services script creates a new `core/phone/phone.service.ts` file in the target application. The template script converts the template files and the related component files. This makes for additional four converted files: `phone-detail.component.ts`, `phone-detail.template.html` as well as `phone-list.component.ts`, and last of all, `phone-list.template.html`. However, the resulting application still fails to compile as the converted files still contain some syntactical errors, and some relevant parts of the application are missing.

The `angularjs-to-angular` could not convert all files from the original `angular-phonecat` application. The upgrade is now missing the checklist filter, and animations, the application routing is not configured, and data files are missing. Data-wise, the JSON and JPG files still need to be copied over to the upgraded project. To obtain a functional project, these need to be fixed.

## 5.4 A working modernization of the angular-phonecat

As stated in the previous section, the modernized version of the application requires some manual fixes before becoming a functional application. The first set of fixes is to correct the compilation errors in the upgraded files. When compiled with `ng serve`, the integrated TypeScript compiler now gives three compilation errors for the upgraded phone service file, another three errors for the upgraded phone-detail component file, and yet another three errors for the upgraded phone-list component file. All compilation errors are related to the `angularjs-to-angular` tool not recognizing syntax in the original source files, and thus copying the AngularJS syntax to the upgraded files as-is.

The first category of compilation errors was caused by the upgrade tool not recognizing imports from the angular module. The host module of used AngularJS features such as `import { IComponentController } from 'angular';` had to be specified with in-place dot syntax. For example:

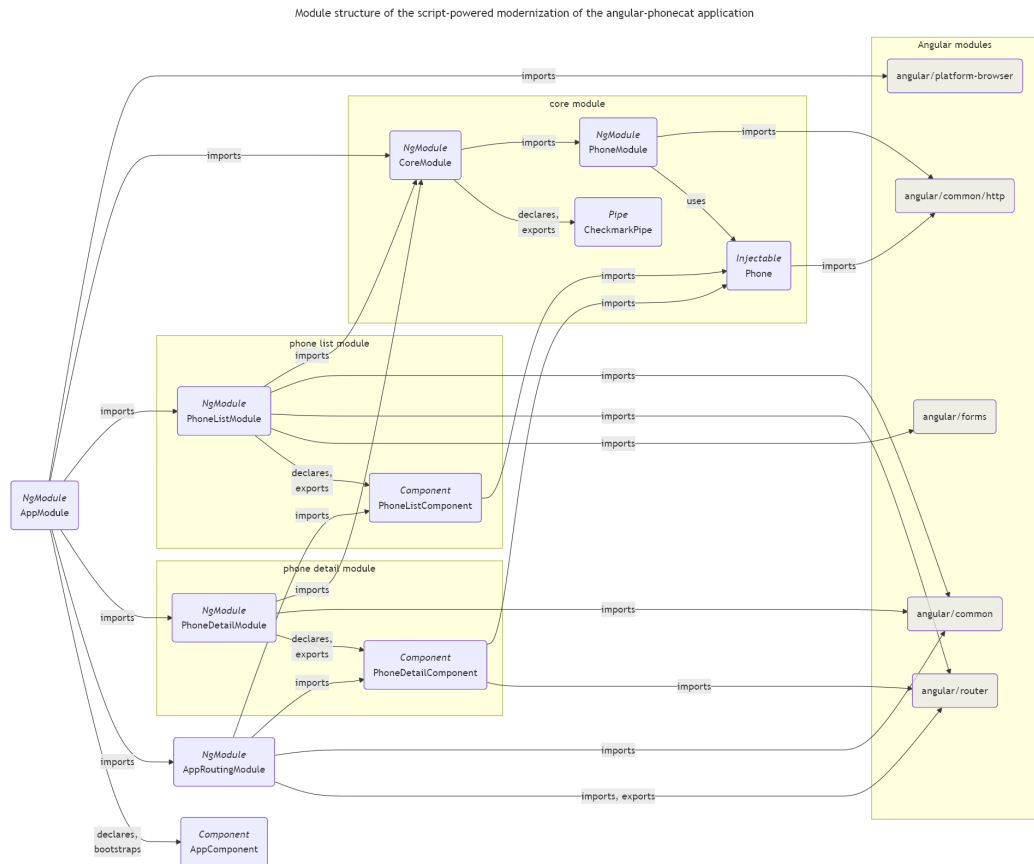
```
class PhoneListController implements
    angular.IComponentController {
    ...
}
```

The second class of compilation errors originated from the upgrade tool not recognizing static inject syntax, so the static injects needed to be removed by hand. The last compilation issue was caused by the upgrade tool not knowing how to replace the `routeParams` service, so obtaining the `phoneId` route parameter needed to be manually implemented via the `ActivatedRoute[5]` interface.

After these compilation errors are fixed, the application compiles again, but is not bootstrapping the angular-phonecat application, but rather the `AppComponent` of the starter application. This is fixed by editing the `AppModule` to import the angular-phonecat modules. However, the upgrade scripts do not create modules on their own, so the module definition files need to be created manually. Modules are created for application routing, application core definition, phone detail, phone list and phones, and the module structure follows the same structure as the reference modernization, displayed in figure 5.1. Compared with the original angular-phonecat module structure, displayed in figure 4.4, the only major difference in terms of module structure is that the modernized version now defines an additional module `AppRoutingModule` for configuring the correct components to display, depending on the URL user navigates to in the browser.

In addition to creating the missing module definitions, the `AppComponent` template needs to be updated to use the `RouterOutlet` directive instead of the placeholder template defined by the starter application. The `RouterOutlet` directive is used by Angular to dynamically display the configured component of the currently selected route in the router state. [13] The available routes are configured for the Angular router in the `AppRoutingModule`. An additional tweak in the `AppRoutingModule` allows to use an `APP_BASE_HREF` injection token [6] and a hash location strategy [11] to set the URL paths to always start with a constant path component, `#!/`, which is a AngularJS practice utilized in the original angular-phonecat application as well. With this configuration, the modernized application preserves the same URL addresses as the original.

Once the new modules are in place and the application routes are defined, the template files of the phone list and phone detail components require some fixes. In the phone list template, the upgrade scripts copied an expression that should fetch and order a phone list from the component. The expression contains references to a `query` and `orderBy`



**Figure 5.1.** The modernized application structure for the angular-phonecat. The `@angular/core` dependencies are omitted for clarity, since the core module is used by all modules and components.

filters that are not defined in Angular. This causes runtime errors. The proper way to fix this issue would be to add an additional preparation step to the migration to implement custom filters and migration script extension that converts the filters to Angular pipes. As AngularJS filters and Angular pipes resemble each other very closely, the script extension should be rather straight-forward to implement. Using this approach would, for instance, allow the pipes to utilize caching the list order. However, in this pilot experiment the goal is to produce a migration with minimal structural changes, so the filters are replaced manually by implementing custom functions in the phone list component that fetch a list of phones and order them according to selected order prop, which is the same approach as adopted in the reference modernization. The functions are then called each time the template is evaluated, which could be less performant than caching the result [12].

Another required fix in the template is to swap the HTML `a` element attribute `href` to Angular `routerLink` directives, so that the framework can control displaying the correct phone images depending on the selected phone. Lastly, the HTML `img` tag attribute specification uses a mixture of Angular `src-directive` syntax and dynamic binding syntax, which conflict with each other. This is likely a bug in the upgrade script. It can be fixed manually by either removing the double-curl braces indicating an Angular binding, or

switching the `src` directive `[src]` to a simple HTML `src` attribute. The same fix is required in the phone detail template as well, and the last required template fix is to wrap the phone detail template in an HTML `div` element with an `ngIf` directive checking the phone data variable is defined before rendering the template. This is to avoid runtime errors from referring nonexistent variables during the template rendering.

Finally, the phone data and images are copied over to the upgraded application. However, when utilizing the selected starter application, the target folder for the application data is under the `assets` directory, not the `app` directory. Therefore, the path operations `get(...)` and `query()` in the phone service require an update in the search path URLs to include the `assets/` path component. The same update needs to be made on all paths defined in the phone data JSON files. The last missing file is the checkmark filter, which is modernized exactly the same way as in the reference modernization.

With these fixes, the application now compiles and is mostly functional, as discussed in the next chapter. To get the application styles working, the `app.css` and the bootstrap style sheets need to be configured in the `angular.json` configuration file in the root of the repository. This concludes the upgrade steps.

## **6. THE RESULTS AND VALIDATION: COMPARING THE UPGRADE WITH THE REFERENCE MODERNIZATION**

In this pilot experiment, a modernized version of the phonecat-application in Angular 6 was produced in two different ways: by utilizing the angularjs-to-angular upgrade tool, and by following the instructions for manual conversion given by the Angular team. To assess the success of the upgrade approaches, the functionality of the application versions is tested by running a set of functional test cases, as discussed in section 6.1. The reference modernization approach and the reference application are then used as a reference point to validate the modernization approach and the modernized application produced using the angularjs-to-angular upgrade tool. The validation is done by comparing first the produced applications and then the modernization approaches. The comparison of the two upgraded application versions is presented in section 6.2, and the steps of the two modernization approaches are summarized and compared in section 6.3.

### **6.1 Functional tests**

To give a meaning for what a functional application means, a set of functional test cases is defined based on how the original angular-phonecat application works. Table 6.1 lists 15 functional test cases for the landing view, which cover the meaningful functional aspects of the original angular-phonecat application. Similarly, table 6.2 lists 10 test cases that cover the meaningful functionality of the phone detail view. An upgrade is considered functional if the application compiles, starts up, passes these test cases and does not produce errors in the console during test execution.

To follow best practices of software development, one would make sure these test cases were implemented as end-to-end tests before the upgrade, and then run the tests again after migration as regression tests. However, as upgrading the existing or any new unit and end-to-end tests in the original angular-phonecat application is excluded from the pilot experiment, these test cases were performed only manually for the original application, the upgraded version, and the reference application. As the test cases are drawn up based on the original application, all cases naturally pass for the 1.5-snapshot version of

the angular-phonecat, but the test results for the upgraded versions are summarized in tables 6.1 and 6.2.

As seen in tables 6.1 and 6.2, both the angularjs-to-angular-powered upgrade and the reference modernization pass all cases but three. Animations were excluded in both modernizations, so the transitions on the landing view are instant instead of smooth transitions, and on the detail view, clicking the small phone icons on the phone detail pages does not change the selected main image, or display a smooth fade-in and fade-out animation, as it does in the original angular-phonecat application.

In summary of the functional test results, it seems that the transpilation tool can be utilized to produce an application that functions similar to application produced by the official migration path, at least in the case of a small test application.

## 6.2 Structural validity

However, functionality is not the only relevant measure for a working application, but code quality also counts. In this case, the application produced by the reference modernization is considered a baseline for a modernization with a sane application structure. If the structure of the application produced by the angularjs-to-angular tool ended up resembling it, that structure can also be considered sane.

Figures 4.8 and 5.1 present the module structures of the upgraded applications produced via the two modernization methods. The original angular-phonecat application structure is presented in figure 4.4. The figures demonstrate that in the reference modernization the modules defined in the original angular-phonecat application disappear. In contrast, the angularjs-to-angular approach preserves the modules. This is not really due to the tool itself, as it currently does not implement conversion functionality for modules. The reason the module files were modernized and brought to the upgraded version as well was to follow the structure of the original application, even though it added the manual work of upgrading the files. There is no reason why the same modules could not be manually added to the reference modernization as well.

The most major aspect that separates the two applications is the tooling utilized in scaffolding the applications. In the original application, Bower [30] is used to manage package dependencies alongside npm [1], and during the reference modernization, SystemJS [87] module loader is brought along to manage module loading. However, with the angularjs-to-angular approach, one may choose the starter application used as the skeleton for the upgraded application. In the pilot experiment, the starter application generated by the angular-cli tool comes pre-configured with Webpack [102] module bundler, and the workspace created by the angular-cli tool takes care of configuring several other aspects of the project scaffolding as well, all easily edited in one configuration file[8]. It

| Test case, the landing view  | Upgraded application | Reference modernization |
|--|----------------------|-------------------------|
| Displays a list of 20 phones.  | Yes                  | Yes                     |
| Each item in the list is styled by box border.   | Yes                  | Yes                     |
| Each item in the list displays a title row presenting the phone name.  | Yes                  | Yes                     |
| Each title is a functional link to phone details.  | Yes                  | Yes                     |
| Each item in the list displays a thumbnail image displaying an image of the phone.   | Yes                  | Yes                     |
| Each item in the list includes a short description of the phone.   | Yes                  | Yes                     |
| In the top left corner of the view, there is a search box.   | Yes                  | Yes                     |
| The search box is labeled Search: and next to it, there is an empty input box.   | Yes                  | Yes                     |
| Inputting text in the input box filters the phone list on each key stroke, displaying only the phones, of which name contains the inputted text. | Yes                  | Yes                     |
| Under the search box, there is a select element labeled Sort by.   | Yes                  | Yes                     |
| The search element displays 2 options: Alphabetical and Newest.  | Yes                  | Yes                     |
| The option Newest is selected by default and the phones are ordered according to release date.   | Yes                  | Yes                     |
| Selecting option Alphabetical reorders the phone list according to phone title.  | Yes                  | Yes                     |
| Selecting Newest returns the ordinal phone order.  | Yes                  | Yes                     |
| Transition animations between phone list reorders are smooth.  | No                   | No                      |

**Table 6.1.** An implementation of the *phonecat* application is considered functional if it passes functional test cases. This table lists functional tests for the landing view.

| Test case, the detail view  | Upgraded application | Reference modernization |
|---|----------------------|-------------------------|
| Displays a div HTML element for a large phone image in the top left corner.   | Yes                  | Yes                     |
| The phone title is displayed in a h1 element in the top edge of the page and the title is underlined.   | Yes                  | Yes                     |
| Under the title, there is a description of the phone.   | Yes                  | Yes                     |
| Under the description, there is a set of thumbnail images of the phone.   | Yes                  | Yes                     |
| Clicking on the thumbnail images toggles the large image.   | No                   | No                      |
| The toggle is animated and the previous image fades out as the new image fades in.  | No                   | No                      |
| Under the image selectors, there is a list of 10 different categories of specifications displayed for a phone.  | Yes                  | Yes                     |
| The title of each category is a span element.   | Yes                  | Yes                     |
| Details of the specification category are displayed as a table (using dl, dt and dd elements).  | Yes                  | Yes                     |
| Informing the user on whether a phone has a touch screen is informed by a checkmark icon or a cross icon, indicating that the checkmark functionality is working. | Yes                  | Yes                     |

**Table 6.2.** Functional test cases for the detail view.

is worth noting, that the angular-cli comes with several useful built-in commands such as `ng update`, which automates simple upgrades of the core framework version.[16] In theory, once an AngularJS application is upgraded to Angular 6, the `ng update` command could provide a migration path all the way to the latest version of the framework, but testing this theory is left for future work.

Other than the module structure and the scaffolding, there is little difference in the two versions of the application. Therefore, it would be fair to claim that structure-wise the angularjs-to-angular tool produced a similar modernization as the reference approach.

### 6.3 The validity of the conversion approach

In the approach utilizing the angularjs-to-angular tool, the summarized steps for the upgrade read as follows:

1. First, the source application was prepared for conversion by:
  - Installing TypeScript and the typing libraries.
  - Installing and configuring a module bundler.
  - Refactoring application parts in a normalized format for the tool to read.
2. Choosing a starter project as the skeleton for the upgraded project.
3. Configuring the source and target directories for the tool.
4. Running the conversion scripts for services, templates and components.
5. Fixing the conversion results:
  - Fixing the compilation errors.
  - Manually upgrading and filling in the non-converted parts.
  - Copying the application data over to upgraded application.
  - Bootstrapping the application.
6. Validating the result application by running functional tests.

In this version of the conversion, only one version of the framework is running in the application at a time. The application remains functional during the preparation steps, but after step 4, the first time the application compiles again is in step 5, after fixing any compilation errors produced by the upgrade tool. If the application starts, it is unlikely that it is completely functional before completing the rest of the fix list presented in step 5. If the application to modernize is meant to be used during the modernization, steps 4 to 6 need to be performed at once before the application is usable.

In comparison, in the reference modernization, the steps to modernize are:

1. Preparing the source application for conversion:
  - Installing TypeScript and the typing libraries.
  - Refactoring component controllers as classes.
  - Bringing in a module loader.
2. Preparing a hybrid application:
  - Installing Angular 6 in the project.
  - Installing the Angular ngUpgrade library.
  - Importing the ngUpgrade module.
  - Bootstrapping a hybrid application.
3. Upgrading the application one part at a time.
4. Adding ngRouter and Bootstrap.

5. Removing AngularJS from the project.
6. Validating the result application by running functional tests.

The two processes have some identical preparation steps where a TypeScript compiler is set up in the project and where the component controllers are refactored to classes. Instead, the approaches differ in the application used as a base for the upgrade: the semi-automated process is built on top of a completely separate starter application, which can be chosen somewhat freely, as long as the framework version matches the syntax produced by the angularjs-to-angular tool. For example Angular 6 is supported. In contrast, the reference modernization modifies the source application directly, so any upgrades to tooling need to be modified in the context of the source application code base. For example, the original angular-phonecat uses Bower to manage some dependencies of the application, which is overhead in the upgraded version of the application, as the same functionality can be achieved via npm. Therefore, to reach a clean upgrade, Bower should be removed, which constitutes an additional, albeit optional cleanup task.

Process-wise, the most significant difference between the two approaches is that in the reference modernization, the application can be upgraded gradually, one application part at a time. As a drawback, the reference version of the conversion is fully manual and incorporates the extra steps in terms of setting up the hybrid application. Performance-wise, it is also worth looking into if the two versions of the framework actually do play together seamlessly, albeit that is a question for a different investigation. The validation of the resulting projects can be done via identical test cases.

## 7. DISCUSSION

This thesis investigated how to perform a specific modernization task from AngularJS to its more modern counterpart, Angular 6, in a semi-automated way. The steps were laid out for how to perform a modernization utilizing the `angularjs-to-angular` conversion tool by: first preparing the application and choosing a starter application, then configuring and running the upgrade tool, fixing any errors produced in the conversion and completing the modernization with parts not automatically converted. Lastly, the conversion result is validated by testing.

In addition to sorting out the required upgrade steps, the conversion results were validated by functional testing and comparison with a reference modernization. The limitations and assumptions of the approach were also identified and documented. Most importantly, anyone hoping to utilize this approach should choose the starter project to support the future needs of their project and consider whether it is feasible in their migration project to perform the upgrade and fixes in the whole application at once after completing the preparation steps.

This thesis set out to find out whether a semi-automated tooling could be utilized in modernizing applications written in AngularJS, and if so, what are the steps to perform the migration. As a result, the `angularjs-to-angular` tool was successfully applied to produce a functioning application in Angular 6, and the modernization steps were documented. As a restriction, the tool was only able to read a limited set of AngularJS features, defined in a specific format, so a significant amount of manual preparation work was required to refactor the existing application in a format that the tool could read. Upgrading several application aspects were not supported at all by the tool. For example, upgrading animations, modules, or filters needed to be manually upgraded to complete the application upgrade. Multiple manual fixes were also required in the parts of the application that were converted by the tool.

As a recommendation for future work, it should be investigated how to migrate the unit tests and the end-to-end tests written in conjunction with the original application. Tests are an integral part of any code migration, development, or refactoring project to prevent regression during the migration and to give confidence the migrated application remains functional. Furthermore, the `angularjs-to-angular` tool could be developed further to ac-

count for a more complete set of AngularJS features to convert. For example, implementing automatic module upgrade should not be that different from the component migrations the tool already implements. Additively, some of the normalization done in preparation of the conversion, could be automated by implementing logic for recognizing component, controller, or service class definitions. Furthermore, upgrading AngularJS filters to Angular 6 pipes could be potentially straight-forward automation, since the features resemble each other very closely. This way, the tool could account for a greater part of the upgrade work.

Yet another interesting topic for further investigation would be looking into the ng update command provided by the angular-cli tool. In conjunction with the presented migration path from AngularJS to Angular 6, the ng update could potentially provide a migration path from Angular 6 to all the way to modern Angular version 21.

Lastly, even though the angularjs-to-angular tool was validated to be applicable to modernizing a small test application, it is not evident how applicable the tool would be for modernizing more extensive systems, at least in its current form. In one hand, the significant preparation work required to refactor the application into a format angularjs-to-angular tool currently supports, might overlap with the actual conversion. On the other hand, in a extensive code base, a process that normalizes the code base and automates even some parts of the conversion might prove to be a valuable aid. In conclusion, the results received in this experiment give some support to the idea of utilizing tooling in the AngularJS to Angular 6 migration, whenever a gradual migration is not a hard requirement.

## 8. CONCLUSIONS

In this thesis, a transpilation tool for migrating AngularJS applications to Angular version 6 was validated by conducting a pilot experiment. In the experiment, a simple test application was modernized by performing a set of preparation steps, utilizing the tool to upgrade the component, template and service files on top of an empty starter application, and implementing manual fixes to complete the upgrade. A reference modernization was also formed of the same source application by following a migration guide laid out by the Angular development team. The required steps to perform the two migrations were summarized, and a set of functional tests were performed on the upgraded applications to verify their functionality. The resulting application from the upgrade with the upgrade tool was compared with the reference modernization to assess the quality of the resulting application.

As a result of the experiment, it was concluded that the angularjs-to-angular upgrade tool can be utilized to produce a valid modernization of at least a simple AngularJS application. However, several restrictions were identified and it was left unclear whether the use of the tool reduces the amount of manual work required to perform a successful migration. Anyone considering using the tool should also consider whether it is feasible to perform the upgrade in one go once the preparation steps are implemented on the application, since the approach does not allow for incremental migration.

## REFERENCES

- [1] *About npm | npm Docs*. Oct. 23, 2023. URL: <https://docs.npmjs.com/about-npm> (visited on 11/26/2025).
- [2] *About semantic versioning*. Sept. 23, 2025. URL: <https://docs.npmjs.com/about-semantic-versioning> (visited on 09/23/2025).
- [3] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. "Using machine translation for converting Python 2 to Python 3 code". eng. In: *PeerJ preprints* (2015). ISSN: 2167-9843.
- [4] *Angular. Introduction to Angular docs*. Aug. 27, 2023. URL: <https://angular.io/docs> (visited on 08/27/2023).
- [5] *Angular - ActivatedRoute*. Oct. 10, 2018. URL: <https://v6.angular.io/api/router/ActivatedRoute> (visited on 11/26/2025).
- [6] *Angular - APP\_BASE\_HREF*. Oct. 10, 2018. URL: [https://v6.angular.io/api/common/APP\\_BASE\\_HREF](https://v6.angular.io/api/common/APP_BASE_HREF) (visited on 11/26/2025).
- [7] *Angular - Dependency Injection in Angular. Dependency Injection in Angular*. Oct. 10, 2018. URL: <https://v6.angular.io/guide/dependency-injection> (visited on 11/26/2025).
- [8] *Angular - Getting Started*. Oct. 10, 2018. URL: <https://v6.angular.io/guide/quickstart> (visited on 11/16/2025).
- [9] *Angular - Glossary. workspace*. URL: <https://angular.io/guide/glossary#workspace> (visited on 09/29/2023).
- [10] *Angular - Glossary. workspace configuration*. URL: <https://angular.io/guide/glossary#workspace-configuration> (visited on 09/29/2023).
- [11] *Angular - HashLocationStrategy*. Oct. 10, 2018. URL: <https://v6.angular.io/api/common/HashLocationStrategy> (visited on 11/26/2025).
- [12] *Angular - Pipes. Pure and impure pipes*. Oct. 10, 2018. URL: <https://v6.angular.io/guide/pipes#pure-and-impure-pipes> (visited on 11/26/2025).
- [13] *Angular - RouterOutlet*. Oct. 10, 2018. URL: <https://v6.angular.io/api/router/RouterOutlet> (visited on 11/22/2025).
- [14] *Angular - Setup for local development*. Oct. 10, 2018. URL: <https://v6.angular.io/guide/setup> (visited on 11/25/2025).
- [15] *Angular - TypeScript Configuration*. Oct. 10, 2018. URL: <https://v6.angular.io/guide/typescript-configuration> (visited on 11/26/2025).
- [16] *Angular - Updating your Angular projects*. Oct. 10, 2018. URL: <https://v6.angular.io/guide/updating> (visited on 11/24/2025).

- [17] *Angular 1 Style Guide*. Dec. 3, 2017. URL: <https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md#angular-1-style-guide> (visited on 06/26/2023).
- [18] *Angular is coming: Preparing the upgrade*. Feb. 26, 2018. URL: <https://bytes.grubhub.com/angular-is-coming-preparing-the-upgrade-be9ec8954ea2> (visited on 06/18/2023).
- [19] *angular-phonecat/LICENSE at master · angular/angular-phonecat*. Oct. 10, 2013. URL: <https://github.com/angular/angular-phonecat/blob/master/LICENSE> (visited on 11/26/2025).
- [20] *angular/angular-phonecat. Tutorial on building an angular*. Dec. 10, 2016. URL: <https://github.com/angular/angular-phonecat> (visited on 01/10/2021).
- [21] *angular/angular-phonecat at 1.5-snapshot. angular-phonecat at GitHub public archive*. Dec. 21, 2016. URL: <https://github.com/angular/angular-phonecat/tree/1.5-snapshot> (visited on 07/24/2023).
- [22] *angular/angular.js v0.9.0 GitHub. Releases*. Oct. 21, 2018. URL: <https://github.com/angular/angular.js/releases/tag/v0.9.0> (visited on 08/27/2023).
- [23] *angularjs-to-angular*. June 18, 2023. URL: <https://github.com/erictsai6/angularjs-to-angular> (visited on 11/23/2025).
- [24] *AngularJS: Tutorial: 12 - REST and Custom Services*. Apr. 7, 2022. URL: <https://docs.angularjs.org/api/ngRoute> (visited on 11/26/2025).
- [25] *AngularJS: Tutorial: 12 - REST and Custom Services*. Apr. 7, 2022. URL: [https://docs.angularjs.org/tutorial/step\\_13](https://docs.angularjs.org/tutorial/step_13) (visited on 11/26/2025).
- [26] *AngularJS: Tutorial: Tutorial. PhoneCat Tutorial App*. Apr. 12, 2022. URL: <https://docs.angularjs.org/tutorial> (visited on 10/23/2025).
- [27] *antlr/antlr4: ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files*. Sept. 3, 2025. URL: <https://github.com/antlr/antlr4> (visited on 10/14/2025).
- [28] *antlr/grammars-v4: Grammars written for ANTLR v4; expectation that the grammars are free of actions*. Oct. 17, 2022. URL: <https://github.com/antlr/grammars-v4> (visited on 10/14/2025).
- [29] *Automating the Angular Conversion*. Aug. 2, 2018. URL: <https://bytes.grubhub.com/automating-the-angular-conversion-31e73f6c6f70> (visited on 06/18/2023).
- [30] *Bower a package manager for the web*. Nov. 26, 2025. URL: <https://bower.io/> (visited on 11/26/2025).
- [31] Hugo Brunelière et al. “MoDisco: A model driven reverse engineering framework”. eng. In: *Information and software technology* 56.8 (2014), pp. 1012–1032. ISSN: 0950-5849.
- [32] Javier Cámara et al. “On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML”. eng. In: *Software and systems modeling* 22.3 (2023), pp. 781–793. ISSN: 1619-1366.

- [33] Stefano Ceri, Piero Fraternali, and Aldo Bongio. “Web Modeling Language (WebML): a modeling language for designing Web sites”. eng. In: *Computer networks (Amsterdam, Netherlands : 1999)* 33.1-6 (2000), pp. 137–157. ISSN: 1389-1286.
- [34] E.J. Chikofsky and J.H. Cross. “Reverse engineering and design recovery: a taxonomy”. eng. In: *IEEE software* 7.1 (1990), pp. 13–17. ISSN: 0740-7459.
- [35] José M. Conejero et al. “Re-engineering legacy Web applications into RIAs by aligning modernization requirements, patterns and RIA features”. eng. In: *The Journal of systems and software* 86.12 (2013), pp. 2981–2994. ISSN: 0164-1212.
- [36] *Discontinued Long Term Support for AngularJS*. Sept. 8, 2022. URL: <https://blog.angular.io/discontinued-long-term-support-for-angularjs-cc066b82e65a> (visited on 09/29/2022).
- [37] *Document Object Model (DOM) - Web APIs | MDN. Document Object Model (DOM)*. Sept. 18, 2025. URL: [https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model) (visited on 09/18/2025).
- [38] *Express - Node.js web application framework*. Sept. 18, 2025. URL: <https://expressjs.com/> (visited on 09/18/2025).
- [39] *Extended Long-term support for AngularJS*. Sept. 29, 2022. URL: <https://xlts.dev/angularjs> (visited on 09/29/2022).
- [40] *Finding a Path Forward with AngularJS*. Feb. 21, 2022. URL: <https://blog.angular.io/finding-a-path-forward-with-angularjs-7e186fdd4429> (visited on 09/29/2022).
- [41] Gil Fink and Ido Flatow. “Search Engine Optimization for SPAs”. eng. In: *Pro Single Page Application Development*. Berkeley, CA: Apress, 2014, pp. 267–276. ISBN: 9781430266730.
- [42] Piero Fraternali, Gustavo Rossi, and Fernando Sanchez-Figueroa. “Rich Internet Applications Introduction”. eng. In: *IEEE internet computing* 14.3 (2010), pp. 9–12. ISSN: 1089-7801.
- [43] Kelly Garcés et al. “White-box modernization of legacy applications: The oracle forms case study”. eng. In: *Computer standards and interfaces* 57 (2018), pp. 110–122. ISSN: 0920-5489.
- [44] Miguel A. Garzon, Hamoud Aljamaan, and Timothy C. Lethbridge. “Umple: A framework for Model Driven Development of Object-Oriented Systems”. eng. In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*. IEEE, 2015, pp. 494–498. ISBN: 9781479984695.
- [45] *Getting Started | Axios Docs. Learn web development | MDN*. Sept. 20, 2025. URL: <https://axios-http.com/docs/intro> (visited on 09/20/2025).
- [46] Xiaodong Gu et al. “DeepAM: Migrate APIs with multi-modal sequence to sequence learning”. eng. In: *IJCAI (United States)*. 2017, pp. 3675–3681. ISBN: 9780999241103.

- [47] *How I Learned to Stop AngularJS and Love the Angular*. Feb. 9, 2018. URL: <https://bytes.grubhub.com/how-i-learned-to-stop-angularjs-and-love-the-angular-e84b0aabb49d> (visited on 06/18/2023).
- [48] *Introduction | Vue.js*. June 12, 2023. URL: <https://vuejs.org/guide/introduction.html#the-progressive-framework> (visited on 10/27/2023).
- [49] *Introduction to client-side frameworks. Learn web development | MDN*. July 3, 2023. URL: [https://developer.mozilla.org/en-US/docs/Learn/Tools\\_and\\_testing/Client-side\\_JavaScript\\_frameworks/Introduction](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction) (visited on 10/27/2023).
- [50] *Jasmine Documentation*. Nov. 26, 2025. URL: <https://jasmine.github.io/> (visited on 11/26/2025).
- [51] *Karma - Spectacular Test Runner for Javascript*. Nov. 26, 2025. URL: <https://karma-runner.github.io/6.4/index.html> (visited on 11/26/2025).
- [52] T. Katsimpa et al. "Application modeling using reverse engineering techniques". eng. In: *Proceedings of the 2006 ACM Symposium on Applied Computing*. Vol. 2. New York, NY, USA: ACM, 2006, pp. 1250–1255. ISBN: 9781595931085.
- [53] Thilanka Kaushalya and Indika Perera. "Framework to Migrate AngularJS Based Legacy Web Application to React Component Architecture". In: *2021 Moratuwa Engineering Research Conference (MERCon)*. 2021, pp. 693–698. DOI: 10.1109/MERCon52712.2021.9525659.
- [54] Stuart Kent et al. "Model Driven Engineering". eng. In: *Integrated Formal Methods*. Vol. 2335. Lecture Notes in Computer Science. Germany: Springer Berlin / Heidelberg, 2002, pp. 286–298. ISBN: 9783540437031.
- [55] *Knowledge Discovery Metamodel v1.4*. Specification. OMG document number: formal/16-09-01. The Object Management Group, Dec. 2016.
- [56] Marie-Anne Lachaux et al. "Unsupervised Translation of Programming Languages". eng. In: *arXiv.org* (2020). ISSN: 2331-8422.
- [57] Kevin Lano and Hanan Siala. "Using model-driven engineering to automate software language translation". eng. In: *Automated software engineering* 31.1 (2024), pp. 20–. ISSN: 0928-8910.
- [58] Triet H.M. Le, Hao Chen, and Muhammad Ali Babar. "Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges". eng. In: *ACM computing surveys* 53.3 (2020), pp. 1–38. ISSN: 0360-0300.
- [59] Marino Linaje, Juan Carlos Preciado, and Fernando Sanchez-Figueroa. "Engineering Rich Internet Application User Interfaces over Legacy Web Models". eng. In: *IEEE internet computing* 11.6 (2007), pp. 53–59. ISSN: 1089-7801.
- [60] Marino Linaje et al. "Automatic Generation of RIAs Using RUX-Tool and Webratio". In: *Web Engineering*. Ed. by Martin Gaedke, Michael Grossniklaus, and Oscar Díaz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 501–504. ISBN: 978-3-642-02818-2.

- [61] Henri Lunnikivi, Kai Jylkkä, and Timo Hämäläinen. “Transpiling Python to Rust for Optimized Performance”. In: *International Conference on Embedded Computer Systems* (2020), pp. 127–138. DOI: doi:10.1007/978-3-030-60939-9\_9.
- [62] T. R. Madanmohan and Rahul De’. “Open source reuse in commercial firms”. eng. In: *IEEE software* 21.6 (2004), pp. 62–69. ISSN: 0740-7459.
- [63] Aniketh Malyala et al. “On ML-Based Program Translation: Perils and Promises”. eng. In: *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. Piscataway, NJ, USA: IEEE Press, 2023, pp. 60–65. ISBN: 9798350300390.
- [64] Benjamin Mariano et al. “Automated transpilation of imperative to functional code using neural-guided program synthesis”. eng. In: *Proceedings of ACM on programming languages* 6.OOPSLA1 (2022), pp. 1–27. ISSN: 2475-1421.
- [65] Ali Mesbah and Arie van Deursen. “Migrating Multi-page Web Applications to Single-page AJAX Interfaces”. In: *11th European Conference on Software Maintenance and Reengineering (CSMR’07)*. 2007, pp. 181–190. DOI: 10.1109/CSMR.2007.33.
- [66] *Model Driven Architecture (MDA) MDA Guide rev. 2.0*. White Paper. Final Draft. The Object Management Group, June 2014.
- [67] Amine Moutaouakkil and Samir Mbarki. “PHP modernization approach generating KDM models from PHP legacy code”. eng. In: *Bulletin of Electrical Engineering and Informatics* 9.1 (2020), pp. 247–255. ISSN: 2089-3191.
- [68] Antonio Vallecillo Nathalie Moreno José Raúl Romero. “An Overview of Model-Driven Web Engineering and the MDA”. In: *Web Engineering Modelling - and Implementing Web Applications*. Ed. by Luis Olsina Gustavo Rossi Oscar Pastor Daniel Schwabe. Human-Computer Interaction Series. London: Springer, 2008, pp. 353–382. ISBN: 978-1-84628-922-4.
- [69] *Next.js by Vercel - The React Framework*. Sept. 17, 2025. URL: <https://nextjs.org/> (visited on 09/17/2025).
- [70] *Node.js – About Node.js*. Nov. 11, 2025. URL: <https://nodejs.org/en/about> (visited on 11/26/2025).
- [71] *npm trends. Compare NPM package downloads over time*. Sept. 8, 2022. URL: <https://npmtrends.com/> (visited on 09/08/2022).
- [72] Z. Obrenovic and D. Gasevic. “Open Source Software: All You Do Is Put It Together”. eng. In: *IEEE software* 24.5 (2007), pp. 86–95. ISSN: 0740-7459.
- [73] OMG Architecture Board ORMSC. *Model Driven Architecture (MDA)*. White Paper. OMG document number: ormsc/2001-07-01. The Object Management Group, July 2001.
- [74] *Protractor - end-to-end testing for AngularJS*. Jan. 31, 2020. URL: <https://www.protractortest.org/> (visited on 11/26/2025).

- [75] Arno Puder, A Montresor, and F Eliassen. "A code migration framework for AJAX applications". eng. In: *DISTRIBUTED APPLICATIONS AND INTEROPERABLE SYSTEMS, PROCEEDINGS*. Vol. 4025. BERLIN: Springer Nature, 2006, pp. 138–151. ISBN: 9783540351269.
- [76] Roberto Rodríguez-Echeverría et al. "Modernization of Legacy Web Applications into Rich Internet Applications". eng. In: *Current Trends in Web Engineering*. Vol. 7059. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 236–250. ISBN: 3642279961.
- [77] Gustavo Rossi et al. "Refactoring to Rich Internet Applications. A Model-Driven Approach". In: *2008 Eighth International Conference on Web Engineering*. 2008, pp. 1–12. DOI: 10.1109/ICWE.2008.41.
- [78] Sudhir K. Routray et al. "Large Language Models (LLMs): Hypes and Realities". eng. In: *2023 International Conference on Computer Science and Emerging Technologies (CSET)*. IEEE, 2023, pp. 1–6. ISBN: 9798350341737.
- [79] Clara Sacramento and Ana C. R. Paiva. "Web Application Model Generation through Reverse Engineering and UI Pattern Inferring". eng. In: *2014 9th International Conference on the Quality of Information and Communications Technology*. IEEE, 2014, pp. 105–115. ISBN: 1479961337.
- [80] *Semantic Versioning 2.0.0 | Semantic Versioning*. Sept. 23, 2025. URL: <https://semver.org/> (visited on 09/23/2025).
- [81] Hanan Abdulwahab Siala. "Enhancing Model-Driven Reverse Engineering Using Machine Learning". eng. In: *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. New York, NY, USA: ACM, 2024, pp. 173–175. ISBN: 9798400705021.
- [82] Hanan Abdulwahab Siala, Kevin Lano, and Hessa Alfraihi. "Model-Driven Approaches for Reverse Engineering-A Systematic Literature Review". eng. In: *IEEE access* 12 (2024), pp. 62558–62580. ISSN: 2169-3536.
- [83] *SOA Manifesto*. 2009. URL: <https://soa-manifesto.org/default.html> (visited on 10/07/2025).
- [84] Norbert Somogyi and Gabor Kovesdan. "Software Modernization Using Machine Learning Techniques". eng. In: *2021 IEEE 19th World Symposium on Applied Machine Intelligence and Informatics (SAMII)*. IEEE, 2021, pp. 000361–000365. ISBN: 1728180538.
- [85] Encarna Sosa et al. "A model-driven process to modernize legacy web applications based on service oriented architectures". eng. In: *2013 15th IEEE International Symposium on Web Systems Evolution (WSE)*. IEEE, 2013, pp. 61–70. ISBN: 9781479916085.
- [86] *Start a New Project - React*. July 3, 2023. URL: <https://react.dev/learn/start-a-new-react-project> (visited on 10/27/2023).

- [87] *systemjs/systemjs: Dynamic ES module loader*. Dec. 24, 2022. URL: <https://github.com/systemjs/systemjs> (visited on 11/26/2025).
- [88] *The Angular conversion part 4: What we couldnt automate*. Feb. 19, 2019. URL: <https://bytes.grubhub.com/the-angular-conversion-part-4-what-we-couldnt-automate-aa37061cb328> (visited on 06/18/2023).
- [89] Feliu Trias et al. "A toolkit for ADM-based migration: Moving from PHP code to KDM model in the context of CMS-based web applications". eng. In: *Information Systems Development: Transforming Organisations and Society Through Information Systems - Proceedings of the 23rd International Conference on Information Systems Development, ISD 2014*. 2014, pp. 210–217. ISBN: 9789536071432.
- [90] *uidotdev/npm trends. NPM package comparison*. Sept. 8, 2022. URL: <https://github.com/uidotdev/npm-trends> (visited on 09/08/2022).
- [91] *UML history FAQ | Object Management Group*. Sept. 24, 2025. URL: <https://www.omg.org/uml/uml-history-faq.htm> (visited on 09/24/2025).
- [92] *Understanding client-side JavaScript frameworks*. Feb. 24, 2023. URL: [https://developer.mozilla.org/en-US/docs/Learn/Tools\\_and\\_testing/Client-side\\_JavaScript\\_frameworks](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks) (visited on 06/18/2023).
- [93] *Upgrading from AngularJS to Angular*. Oct. 10, 2018. URL: <https://v6.angular.io/guide/upgrade#upgrading-from-angularjs-to-angular> (visited on 12/29/2024).
- [94] *Upgrading from AngularJS to Angular. PhoneCat Upgrade Tutorial*. Oct. 10, 2018. URL: <https://v6.angular.io/guide/upgrade#phonecat-upgrade-tutorial> (visited on 11/26/2025).
- [95] *Upgrading from AngularJS to Angular. Follow the AngularJS Style Guide*. Oct. 10, 2018. URL: <https://v6.angular.io/guide/upgrade#follow-the-angularjs-style-guide> (visited on 11/25/2025).
- [96] *Upgrading from AngularJS to Angular*. Oct. 10, 2018. URL: <https://v6.angular.io/guide/upgrade#using-a-module-loader> (visited on 11/26/2025).
- [97] Benoît Verhaeghe et al. "Migrating GUI behavior: from GWT to Angular". In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2021, pp. 495–504. DOI: 10.1109/ICSME52107.2021.00050. URL: <https://ieeexplore.ieee.org/document/9609198>.
- [98] *vlunn/angular-phonecat-modernization: A modernized version of angular-phonecat test app, where the modernization is built on top of a starter app*. Nov. 24, 2025. URL: <https://github.com/vlunn/angular-phonecat-modernization> (visited on 11/26/2025).
- [99] *vlunn/angular-phonecat-reference-modernization at 1.5-snapshot-modernized. github.com*. Nov. 5, 2025. URL: <https://github.com/vlunn/angular-phonecat-reference-modernization/tree/1.5-snapshot-modernized> (visited on 11/26/2025).
- [100] *vlunn/angular-phonecat-to-upgrade*. Nov. 21, 2025. URL: <https://github.com/vlunn/angular-phonecat-to-upgrade> (visited on 11/26/2025).

- [101] *vuejs/core: Vue.js is a progressive, incrementally-adoptable JavaScript framework for building UI on the web*. Sept. 18, 2025. URL: <https://github.com/vuejs/core> (visited on 09/18/2025).
- [102] *webpack - npm*. Oct. 10, 2018. URL: <https://www.npmjs.com/package/webpack/v/1.13.2?activeTab=versions> (visited on 11/26/2025).
- [103] L. Wood. "Programming the Web: the W3C DOM specification". eng. In: *IEEE internet computing* 3.1 (1999), pp. 48–54. ISSN: 1089-7801.
- [104] *XML Introduction - XML | MDN*. Sept. 29, 2025. URL: [https://developer.mozilla.org/en-US/docs/Web/XML/Guides/XML\\_introduction](https://developer.mozilla.org/en-US/docs/Web/XML/Guides/XML_introduction) (visited on 09/29/2025).
- [105] *XMLHttpRequest - Web APIs | MDN*. Sept. 20, 2025. URL: <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest> (visited on 09/20/2025).
- [106] S. Yakowitz. "Unsupervised learning and the identification of finite mixtures". eng. In: *IEEE transactions on information theory* 16.3 (1970), pp. 330–338. ISSN: 0018-9448.
- [107] Zhi-Hua Zhou. "Open-environment machine learning". eng. In: *National science review* 9.8 (2022), nwac123–. ISSN: 2095-5138.