

**FRAMEWORK TO MIGRATE ANGULARJS BASED  
LEGACY WEB APPLICATION TO REACT  
COMPONENT ARCHITECTURE**

Thilanka Kaushalya Lanka Geeganage

179328A

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

March 2021

# **FRAMEWORK TO MIGRATE ANGULARJS BASED LEGACY WEB APPLICATION TO REACT COMPONENT ARCHITECTURE**

Thilanka Kaushalya Lanka Geeganage

179328A

Thesis submitted in partial fulfillment of the requirements for the degree Master of  
Science

Department of Computer Science and Engineering

University of Moratuwa  
Sri Lanka

March 2021

## DECLARATION

I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: .....

Date:.....

Name: Thilanka Kaushalya Lanka Geeganage

I certify that the declaration above by the candidate is true to the best of my knowledge and that this report is acceptable for evaluation for the Masters thesis.

Signature of the supervisor: .....

Date:.....

Name: Dr. Indika Perera

## ABSTRACT

Due to the increased complexity and the high expectation of the modern web applications, the legacy applications built on top of previous generation frontend frameworks such as AngularJS fail to serve both end users and developers. The AngularJS based legacy applications start poorly performing when the application grows larger. At the same time they become highly challenging for developers to maintain and add new functionality over time. The next generation component based frontend frameworks such as React has outperformed AngularJS from both the end user and developer expectations. Since the whole industry has evolved to gain the benefits of the component architecture, sooner or later all these legacy applications need to migrate to new component based architecture. The concepts, patterns and the architecture of the old school AngularJS application and the component based React applications are different so the migration is not a straightforward step. The successor of AngularJS which is named as Angular2 provides some tooling to support migrating old AngularJS applications to Angular2 but those are not directly compatible with React migration. This makes the organizations and developers who are planning to migrate AngularJS application to stick with Angular2 which is a less popular and flexible framework when compared with React. This research tries to define a clear migration path from AngularJS applications to React applications which will allow organizations and developers to choose React also as a goto framework option and will save hundreds of research and development hours in setting up the migration.

This research compares and contrasts approaches of legacy frontend application migrations with respect to steps, best practices, technologies tools and pain points. By comparing different approaches, the research provides a step by step guideline which can be referred to and followed when migrating any AngularJS based legacy application to component base React application. These guidelines are organized as a migration framework with steps of refactorings and provide as the first outcome of this research. As the second step of this research, a migration assistant tool which is called as Ng-React Copilot was implemented to guide the developer through the migration steps and apply them to the existing code base. The tool was developed by converting the critical refactorings proposed in the framework into a set of detection algorithms and providing the ability to scan against the provided codebase as a command line tool as well as an integrated tool with popular IDEs.

The framework and the tool were validated by applying them to selected small, medium and enterprise level AngularJS legacy applications. One application was selected from each category and used the tool and the framework to step by step migrating to React where the small application was fully migrated to React while the medium and enterprise applications were migrated partially with the interest of the time. The results from the migration assistant tool and the framework were collected and validated the accuracy of them and except for a few false positives, the migration assistant tool was detecting the required refactoring accurately and was making the migration straight forward. Improving the algorithms and configurations to avoid false positives, improve the tool to guide the users as a step by step workflow rather than showing all the refactorings at once, automating few more time consuming manual refactorings, improving the framework to address external library coupling and improve the tool to guide the user on where in the application to start the migration were identified as future improvements for the identified limitations during the evaluation process. By looking at the evaluation results, the migration framework and the Ng-React Copilot tool can be considered as good starting materials to get assistance for AngularJS to React application migration and the effectiveness can be further improved by applying the identified future improvements.

## **ACKNOWLEDGEMENTS**

I would like to express my gratitude to Dr. Indika Perera, my project supervisor for providing advice and guiding the project into success. Even though I had an idea about the domain initially I didn't have a clear idea on how to converge the scope of the research into a meaningful outcome. Dr. Indika helped me and guided on how to narrow down the scope and focusing on building up the framework proposed in this report. Also I would like to express my gratitude to my family for supporting me in focusing on the research. Last but not least I wish to thankful for all my colleagues who have encouraged me to continue the research by expressing their experiences.

## TABLE OF CONTENTS

|                                                                              |           |
|------------------------------------------------------------------------------|-----------|
| DECLARATION .....                                                            | i         |
| ABSTRACT .....                                                               | ii        |
| ACKNOWLEDGEMENTS .....                                                       | iii       |
| TABLE OF CONTENTS .....                                                      | iv        |
| LIST OF FIGURES .....                                                        | vi        |
| LIST OF TABLES .....                                                         | viii      |
| LIST OF ABBREVIATIONS .....                                                  | x         |
| <b>1. INTRODUCTION</b> .....                                                 | <b>1</b>  |
| 1.1. From Javascript libraries to frameworks .....                           | 1         |
| 1.2. What did AngularJS offer .....                                          | 2         |
| 1.3. Limitations and anti-patterns in AngularJS applications .....           | 4         |
| 1.4. AngularJS successors which provides component architecture .....        | 8         |
| 1.5. Using React is a business decision, not a technology decision[15] ..... | 10        |
| 1.6. Problem .....                                                           | 12        |
| 1.7. Objectives .....                                                        | 13        |
| <b>2. LITERATURE REVIEW</b> .....                                            | <b>15</b> |
| 2.1 Choosing the migration strategy .....                                    | 15        |
| 2.2 Prepare AngularJS code for migration .....                               | 16        |
| 2.3. Migrate component by component .....                                    | 26        |
| 2.4. Migrate the forms .....                                                 | 29        |
| <b>3. METHODOLOGY</b> .....                                                  | <b>31</b> |
| 3.1. A migration framework from AngularJS to React .....                     | 31        |
| 3.2. Migration assistant tool .....                                          | 35        |
| 3.3. Set of algorithms to detect the required refactoring .....              | 47        |

|                                                                                |            |
|--------------------------------------------------------------------------------|------------|
| 3.4. Evaluating the migration framework and the assistant tool.....            | 48         |
| <b>4. IMPLEMENTATION.....</b>                                                  | <b>50</b>  |
| 4.1. Implementation of the Ng-React-Copilot tool .....                         | 50         |
| 4.2. Implementation of the refactoring detection rules .....                   | 52         |
| <b>5. RESULTS AND EVALUATION.....</b>                                          | <b>82</b>  |
| 5.1. Migrating small AngularJS applications having less complicated code ..... | 83         |
| 5.2. Migrating medium level of AngularJS applications.....                     | 87         |
| 5.3. Migrating Enterprise level of AngularJS applications .....                | 91         |
| 5.4. Evaluation of the results .....                                           | 96         |
| <b>6. CONCLUSION .....</b>                                                     | <b>100</b> |
| 6.1. Limitations .....                                                         | 103        |
| 6.2. Future work.....                                                          | 104        |
| <b>7. REFERENCES.....</b>                                                      | <b>106</b> |

## LIST OF FIGURES

|                                                                                                                    |    |
|--------------------------------------------------------------------------------------------------------------------|----|
| Figure 1: How complex are the AngularJS aspects and features [3] .....                                             | 6  |
| Figure 2: Data flow in a react application and a MV* application.....                                              | 12 |
| Figure 3: How data flow happens through React2Angular middleware.....                                              | 26 |
| Figure 4: Workflow of the migration framework.....                                                                 | 34 |
| Figure 5: Ng-React Copilot tool logo in the command line .....                                                     | 44 |
| Figure 6: Ng-React Copilot component diagram .....                                                                 | 45 |
| Figure 7: Ng-React Copilot flow diagram .....                                                                      | 46 |
| Figure 8: Ng-React Copilot select scan mode .....                                                                  | 50 |
| Figure 9: Ng-React Copilot highlights of the analysis results.....                                                 | 51 |
| Figure 10: Ng-React Copilot summary report.....                                                                    | 51 |
| Figure 11: Ng-React Copilot detailed report.....                                                                   | 52 |
| Figure 12: Ng-React Copilot rules highlighting a required refactoring in a code<br>opened with Visual Studio ..... | 52 |
| Figure 13: AST for detecting a directive.....                                                                      | 54 |
| Figure 14: AST for detecting a directive factory defined as a function.....                                        | 65 |
| Figure 15: AST for detecting a directive factory defined as an identifier .....                                    | 67 |
| Figure 16: AST for detecting a directive factory defined as function returned from a<br>function expression .....  | 69 |
| Figure 17: Initial scan results for Angular-Phonecat application.....                                              | 83 |
| Figure 18: Scan results for Angular-Phonecat application after generating<br>PhoneDetailsComponent.....            | 84 |
| Figure 19: Highlighted AngularJS specific feature refactoring in<br>PhoneDetailsComponent.....                     | 85 |
| Figure 20: Highlighted AngularJS scope reference refactoring in<br>PhoneDetailComponent .....                      | 85 |
| Figure 21: Migrated PhoneListComponent.....                                                                        | 86 |
| Figure 22: Migrated PhoneDetailedComponent.....                                                                    | 86 |
| Figure 23: Initial scan results for AngularJS CRUD application .....                                               | 87 |
| Figure 24: Scan results for ToolbarComponentof the AngularJS CRUD application                                      | 89 |
| Figure 25: Scan results for FormComponent of the AngularJS CRUD application ..                                     | 90 |

|                                                                                   |     |
|-----------------------------------------------------------------------------------|-----|
| Figure 26: Migration completed Login Form of the AngularJS CRUD application..     | 91  |
| Figure 27: Initial scan for OrangeHRM application .....                           | 92  |
| Figure 28: Scan results for List directive of the the OrangeHRM application ..... | 93  |
| Figure 29: Scan results for ListComponent of the the OrangeHRM application .....  | 94  |
| Figure 30: Migration completed ListComponent of the OrangeHRM application ....    | 96  |
| Figure 31: Workflow of the migration framework .....                              | 102 |

## LIST OF TABLES

|                                                                                                                                  |    |
|----------------------------------------------------------------------------------------------------------------------------------|----|
| Table 1: Comparison between Angular 2 and React[14] .....                                                                        | 9  |
| Table 2: Alternatives for AngularJS features[4].....                                                                             | 24 |
| Table 3: Refactoring and their groupings.....                                                                                    | 33 |
| Table 4: List of refactoring and their automation feasibility .....                                                              | 38 |
| Table 5: Sample AST for JS code.....                                                                                             | 42 |
| Table 6: Event list of the HTMLHint code scanner.....                                                                            | 43 |
| Table 7: Summary of selected technologies for the migration process .....                                                        | 44 |
| Table 8: Code samples to define an angular directive and a component .....                                                       | 53 |
| Table 9: Code samples to define an angular using different syntaxes.....                                                         | 56 |
| Table 10: Pseudo code for rule - Place all the code of a directive together .....                                                | 58 |
| Table 11: Code samples to define an angular directive restrict.....                                                              | 59 |
| Table 12: Pseudo code for rule - Favor directives as elements over attributes, comment and classes.....                          | 59 |
| Table 13: Code samples on how to use ng-transclude directive .....                                                               | 60 |
| Table 14: Pseudo code for rule - Eliminate transclusion .....                                                                    | 60 |
| Table 15: Code samples on how to use ng-include directive .....                                                                  | 61 |
| Table 16: Pseudo code for HTMLHint rule - Extract directive from large templates                                                 | 62 |
| Table 17: Pseudo code of the isTwoWayDataBindingNotUsed method .....                                                             | 64 |
| Table 18: Pseudo code of the extractFactoryFunctionObject method when directive factory defined as a function .....              | 66 |
| Table 19: Pseudo code of the extractFactoryFunctionObject method when directive factory defined as an identifier .....           | 68 |
| Table 20: Pseudo code of the extractFactoryFunctionObject method when directive factory returned from another function.....      | 70 |
| Table 21: Pseudo code of the extractFactoryFunctionObject method when directive function defined with dependency injection ..... | 71 |
| Table 22: Consolidated pseudo code of the isTwoWayDataBindingNotUsed method .....                                                | 72 |
| Table 23: List of AngularJS inbuilt directives selected to detect inside the JSX.....                                            | 74 |
| Table 24: AST generated for AngularJS directive usage detection in JSX .....                                                     | 75 |

|                                                                                                        |     |
|--------------------------------------------------------------------------------------------------------|-----|
| Table 25: Pseudo code for algorithm to detect AngularJS directive usage in JSX ....                    | 76  |
| Table 26: AST generated for variable references used in JSX.....                                       | 78  |
| Table 27: Pseudo code for algorithm to detect scope references in in JSX .....                         | 80  |
| Table 28: Scan results for Angular-Phonecat application after generating<br>PhoneDetailComponent ..... | 84  |
| Table 29: Initial scan results for AngularJS CRUD application .....                                    | 88  |
| Table 30: Scan results for ToolbarComponentof the AngularJS CRUD application.                          | 89  |
| Table 31: Scan results for FormComponent of the AngularJS CRUD application....                         | 90  |
| Table 32: Initial scan results for OrangeHRM application .....                                         | 92  |
| Table 33: Initial scan results for List directive in OrangeHRM application.....                        | 93  |
| Table 34: Initial scan results for ListComponent in OrangeHRM application .....                        | 94  |
| Table 35: Summary of all detected refactorings.....                                                    | 97  |
| Table 36: Migration Framework.....                                                                     | 101 |

## LIST OF ABBREVIATIONS

|      |                            |
|------|----------------------------|
| DOM  | Document Object Model      |
| MVC  | Model View Controller      |
| MVVM | Model View View Model      |
| MVP  | Model View Presenter       |
| MV*  | Model View Any             |
| AST  | Abstract Syntax Tree       |
| JS   | Javascript                 |
| CSS  | Cascading Style Sheets     |
| XML  | Extensible Markup Language |
| JSX  | Javascript XML             |

# 1. INTRODUCTION

## 1.1. From Javascript libraries to frameworks

After the first introduction of Javascript in the 90's, it has gone through many revolutions to become one of the most popular programming languages and technologies. Even though it has started to enrich the responsiveness of the web pages in early days, now it has come to a stage where you can do everything in the web using Javascript. During that journey, there are few key milestones we need to focus on in the context of this research.

In the early 2000's, when developers started implementing behaviors in Javascript, one of the main issues they faced is that the code they write started appearing differently in different browsers due to inconsistent implementations of the Javascript engines in different browsers[1]. Developers were supposed to know about these inconsistencies and handle them within the program to avoid surprises in production.

In the year 2006 JQuery was introduced to solve this problem as well with many richer features which made it really easy to work with Javascript. Developers didn't have to worry about browser compatibility anymore and JQuery was taking care of it. It introduced many useful methods to interact with DOM, perform AJAX queries and even handling which made it the de-facto way to write Javascript.

With the adoption of JQuery, web applications started to cater more interactive web applications and the complexity of the use cases, higher expectations and the increase of traffic increased the complexity of the web application. This created a vacuum of technologies which can make it easy to build and maintain complex web applications. Javascript frameworks came into the picture to help developers to build complex web applications in a consistent manner. Backbone, Knockout and Ember were some of the initial frameworks which came to address this problem. In late 2010, a game changing "AngularJS" framework came into the picture from a google

developer which was later backed by the Google team. It quickly created a hype among web developers and lots of people were bought into the framework since it provided all in one solution for most of the problems they have encountered in developing web applications at that time. In no time AngularJS became the most popular MVC frontend framework in the industry.

## **1.2. What did AngularJS offer**

AngularJS was an all in one framework which allowed rapid web application development. It followed MVC architecture which allowed developers to apply the same concept they have been using in server side frameworks into the client side development[2]. At the same time, AngularJS provides a bundle of new components and features with it to help the rapid application development. Following are the main building blocks of AngularJS[3].

1. Modules: provides a cleaner way to divide the whole application into manageable sections. As an example an application can be divided into modules by their business functionality such as Admin, Leave and Performance.
2. Services: are a collection of functions with a specific concern such as performing network operations, handling timeouts or array operations. These services are stateless objects which can be used by injecting to various parts of the angular application. AngularJS provides a set of inbuilt services which covers most common operations and developers can write their own custom services to cater their use cases.
3. Templates: allows to define how the web page UI is presented to the user. The templates are defined using HTML elements and attributes which finally renders to the browser DOM.
4. Directives: is a concept which was newly introduced with AngularJS to allow enhancing the functionality defined in the templates before rendering into the final DOM. Directives can perform changes such as updating the DOM structure and registering for events and recommended to use developing

reusable UI components. Directives is a very powerful feature as well as one of the least understood features in AngularJS.

5. Expressions: are used to pass data to the template which get resolved before rendering to the final DOM.
6. Controllers: act as the glue between the data model and the view templates in angular. Controllers use a special container called scope to communicate with templates and the data modal. Since controllers are very powerful and easy to code, lately most of the applications were built around controllers.
7. Digest Cycle: make sure the view is synchronized with the internal scope of the application. Digest Cycle compares the current value of the scope and the value in the DOM to determine the required DOM updates to keep the scope and the view synchronized.
8. Dependency injection: design pattern is used to decouple dependencies from their usages so it's possible to easily replace the dependencies as well as test the dependencies.
9. Pure old Javascript objects: were used to represent the models made it easy to understand the data.
10. Testing: was built into the framework with modules such as ngMock which allows to do the simulations.
11. Two-way data binding: allows angular to data from the module to view and view to model effortlessly. This was a very attractive feature in the early stages of the AngularJS application development but later identified as a big NO due to unpredictability of data flow caused by it.
12. Last but not least the framework was backed by the Google team which gave some extra assurance for the developers that it'll have long term support.

Apart from these main building blocks, AngularJS provides tons of other useful features which help developers to rapidly develop frontend web applications. Since more and more complex applications were built on top of AngularJS, after a few years it started revealing the limitations in the framework. Some of these issues were due to the way the community has adapted the framework and some were due the core architecture of the framework.

### 1.3. Limitations and anti-patterns in AngularJS applications

**Controllers and Scope:** At the very early stages of the framework, it was recommended to build the application around angular controllers because that is the natural way of developing things in a MVC architecture. This anti-pattern was named as ‘Controller bloat’[9][10] in AngularJS. The scope and the two-way data binding was heavily used to communicate between controller and the template and the other important elements in AngularJS such as Directives, services and filters were rarely used[4]. The main reason to adopt these anti-patterns was that the majority of the tutorials written by individuals were following these anti-patterns. When less experienced developers were put into the projects they were mainly referring to these ad hoc tutorials to get the job done. This anti-pattern was identified as Scope Soup code[5][6]. Even though it was possible to write component style[7] of code using directives in AngularJS, it was not enforced by the framework. Often these scopes were passed to children or other components which allows to use and modify the data in scopes. This makes it really hard to predict and debug the application data flow.

**Templates:** The templates were also introducing some unpredictability to the application. It was possible to code templates and controllers in confusing and uncoordinated ways where a single controller manages multiple templates and vice versa. Angular provided ways of extending the template behaviors using inbuilt directives such as ng-if, in-repeat and ng-class. These directives were actually injecting logic to the templates and we can only see the real output at runtime. It was very difficult to debug the behaviors of these directives within the templates since it was not possible to place breakpoints in the templates. The templates were not indicating errors in the data passed to it and they were failing silently without throwing any exception. The mix of directives and pure HTML elements were not very distinguishable because the directives were defined as elements, attributes, comment and a class. This reduces the readability of the template code drastically[3].

**Data Flow:** Even though the two-way data flow provided by AngularJS looked promising at the beginning, it made it very difficult to predict the data mutations in large applications. As mentioned in the above “Scope” section, the data were shared between different components using the scope and most of the time it was possible to get them mutated at any of these other components which affects all the other components which referred to these data. The effect is cascading and the application can become very fragile. Even though the isolated scope was a best practice recommended in AngularJS to avoid this ripple effect, it was not very well understood among new developers and was not followed most of the time.

**Directives:** A survey study has checked with bunch of AngularJS developers about their understandability of the concepts such as[3],

1. How to use isolated scopes in directives and their advantages
2. How to use object prototyping to simulate scope
3. Use of dependency injection
4. When to use compile, link and controller functions in directives
5. When to use transclusion directives
6. How to trigger the digest cycle manually and when to do it
7. How to avoid or minimize performance anti-patterns such as watchers, ng-repeat, repeated filters and ng-show[8]
8. How to integrate with external components

Following graph indicates the responses[3]

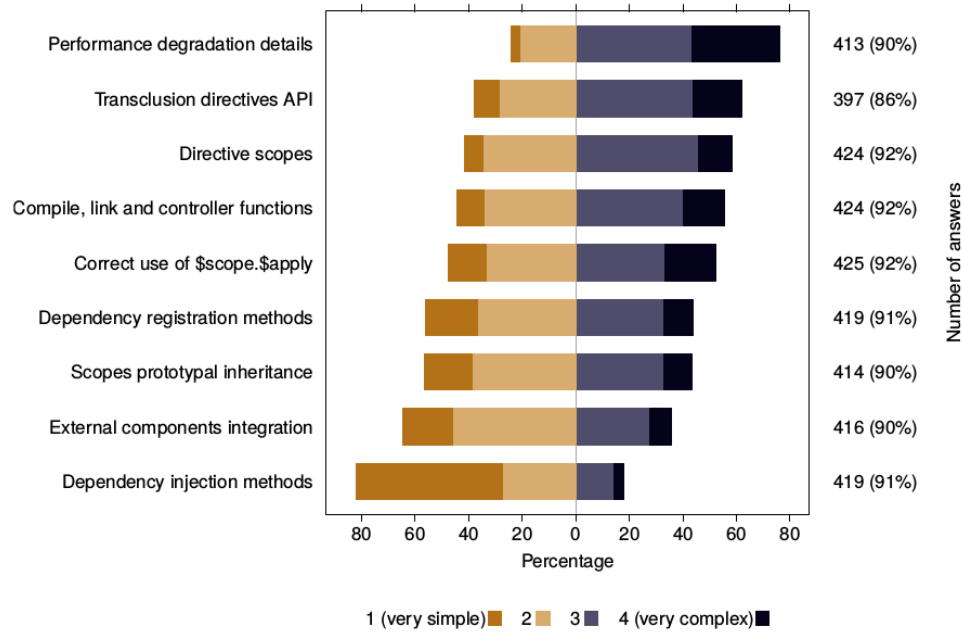


Figure 1: How complex are the AngularJS aspects and features [3]

Out of all the features, 76% of the developers have voted that the performance optimizations is the hardest thing they had to deal with in AngularJS. Transclusion directives, use of isolated scopes in directives and correct use of compile, link and controller functions were the next least understood features. These are directly related to directives and that indicates the directives are not well understood and properly used in AngularJS applications. At the same time 83.5% of the developers have responded that they often write their own directives within the angular application[3]. This is alarming since a very high percentage of developers have used one of the main concepts in AngularJS without really understanding it.

**Multiple choices:** AngularJS has provided tons of features and concepts to achieve the similar use cases. There are easy ways and right ways to do things in AngularJS. Whether to use Services or Factories is such a choice developers have to make which seems identical in outer appearance but has a core difference internally[9]. Often developers make the wrong choice without really understanding the impact. Similarly, a survey has identified that 26% of the developers have to put large HTMLs in Templates which is not a recommended way to implement AngularJS applications. Then the survey asked the reason behind it and 54.8% has responded

that the framework didn't restrict it and 43.5% has responded that lack of experience in best practices caused it. Some other reasons were lack of experience in web architecture and Javascript, tight deadlines and laziness[3]. This clearly indicates the importance of having a simple architecture with well-known rules which can be uniformly applied to make choices when using the framework.

**Performance Anti-patterns:** When an application grows, performance bottlenecks start to appear due to anti-patterns followed by the developers in the early stages of the application. Sometimes these may require entire revamps of the application to get rid. Unfortunately, there are well known performance anti-patterns in AngularJS applications and most of them are coming from important elements of the framework. Some of them are as follows[8][11].

1. Having too much of scope watchers
2. Use of ng-repeat and ng-show directive
3. Not using one time binding when possible
4. Watching object deeply
5. Use of filters.

These elements introduce performance lags to the application and since these are core elements in the framework, by the time developers find the performance issues, the code is highly coupled with these elements which need major refactoring for the application.

These limitations introduced a new vacuum in the web technologies and many alternative solutions started emerging. Most of these solutions proposed a new style of web development which follows Component Architecture. Even the next version of AngularJS which is Angular 2.0 has abandoned its 1.x MVC architecture and moved to a component based architecture where Angular community is asked to refactor their 1.x based applications into components using directives which will make the migration easier[12]. Let's explore more about component architecture.

#### 1.4. AngularJS successors which provides component architecture

As the name implies, Components are atomic pieces which are easily composable. In web development, these components represent pieces of the UI which are composable to build feature rich UIs. The applications which follow component architecture will not be built around controllers and templates as in AngularJS but will be built as independent components which encapsulate its logics and behaviors with a clear interface to communicate. These components are closer to native HTML components such as `<input>` elements which are having a clearly defined set of inputs and outputs. There are three main aspect of a component[5],

1. The interface which defines the boundary of the component with clear inputs and outputs.
2. The UI representation of the component which defines how it's presented to the end user
3. Logic which defines how the component interacts with the end user

Almost all the AngularJS successors such as React, Vue, Ember, Polymer and Angular 2 implement these aspects of components within their architecture.

Out of these mainstream libraries and frameworks, React and Angular 2 were the main candidates to replace AngularJS code in the course of this research. Following two tables compare and contrast the features offered by these two frameworks.

| Angular 2                                                                                                        | React                                                             |
|------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------|
| Is HTML centric so developers have to write Javascript inside HTML                                               | Is Javascript centric so developers have write HTML in Javascript |
| A full framework with all necessary features                                                                     | A library to handle view layer                                    |
| Two way data binding                                                                                             | One-way data binding                                              |
| Verity of options were provided such as Injectable, Components, Pipes and Modules which makes it harder to learn | Only provides Components which makes it easy to learn             |
| Relatively slower in performance                                                                                 | Relatively fast rendering due to virtual DOM                      |
| Used by Microsoft, Autodesk, MacDonald's, UPS, Cisco Solution                                                    | Used by Facebook, Instagram, Netflix, New York Times, Yahoo, Khan |

|                                                                                                                                                                                          |                                                                                                         |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| Partner Program, AT&T, Apple, Adobe, GoPro, ProtonMail, Clarity Design System, Upwork, Freelancer, Udemy, YouTube, Paypal, Nike, Google, Telegram, Weather, iStockphoto, AWS, Crunchbase | Academy, Whatsapp, Codecademy, Dropbox, Airbnb, Asana, Atlassian, Intercom, Microsoft, Slack, Storybook |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|

*Table 1: Comparison between Angular 2 and React[14]*

Just because Angular 2 is the successor of AngularJS, these are totally different frameworks and moving from one to another requires rewrites and major refactoring. Even though Angular 2 has resolved many drawbacks which were there in the 1.x version, it still continues to have some of the issues which make it lag behind React. Angular 2 also needs a steep learning curve because it provides many ways of achieving the same thing. It continues to support two-way data binding which is proven to reduce the predictability of the application. It continues to be HTML centric where developers have to write Javascript inside HTML. This makes it very difficult to debug. Also Angular 2 has not performed well in benchmarks done to performance testing.

At the same time React seems to win over Angular 2 in many aspects. Everything we do in React is a component and it's very easy to understand and learn. React uses one-way data binding which avoids the unpredictability that comes with two-way data binding. It's Javascript centric so developers have to write HTML code inside Javascript. This makes React applications relatively easy to test. React has a different DNA than other competitors. Its core architecture is borrowed from the PHP Hypertext Preprocessor framework which is built to render the whole web page from scratch based on the current state of the application. This makes it more like a state machine which represents different states of the application in UI[13]. When rerendering the whole web application based on the application state, React avoids direct modifications to the browser DOM to avoid performance degradation. It uses a Virtual DOM and a differ algorithm to determine which changes to sync with the actual browser DOM efficiently which makes it perform superior to other competitors in performance benchmarking. Even though React only provides UI

components, there is a large community and a whole echo system built around it which makes it very powerful.

### **1.5. Using React is a business decision, not a technology decision[15]**

Even though React seems to be technically superior to its competitors, choosing React as the way forward cannot be decided only from the technical angle. At the end, no matter how React wins in industry benchmarks, it should align with the business goal of your organization and it should make a positive impact on achieving the organizational goals. Following are some of the points which favors adapting React within an organization.

**Stability** - React is built and invested by the Facebook team and they heavily use it in the mainstream projects such as Newsfeed, Instagram, Messenger and Ads Marketplace. Since React is used in mainstream projects at Facebook scale, it proves React's ability to scale into billion user use cases. This isn't the case for other competitor libraries and specially Google is not using Angular 2 for main components in their flagship projects. React is used by many other industry leading companies such as AirBnB, Asana, Atlassian, BBC, CloudFlare, CNN.com, Codecademy, Coursera, Dropbox, Expedia, HipChat, IMDb, Imgur, Instagram, Khan Academy, Netflix, Rackspace, Reddit, Salesforce, The New York Times, Twitter, Uber, WhatsApp, WordPress, Yahoo and Zendesk[14].

**Community** - React has a very healthy echo system with 1,379 contributors and 3.6 million downloads in Github. There are other tools built from the React community such as Redux which has become the de-facto way to handle the application states and overruns the popularity of the original Flux library built by Facebook. This proves that React has a great community and will continue to get the support and maintenance in the long run without restricting Facebook's decisions.

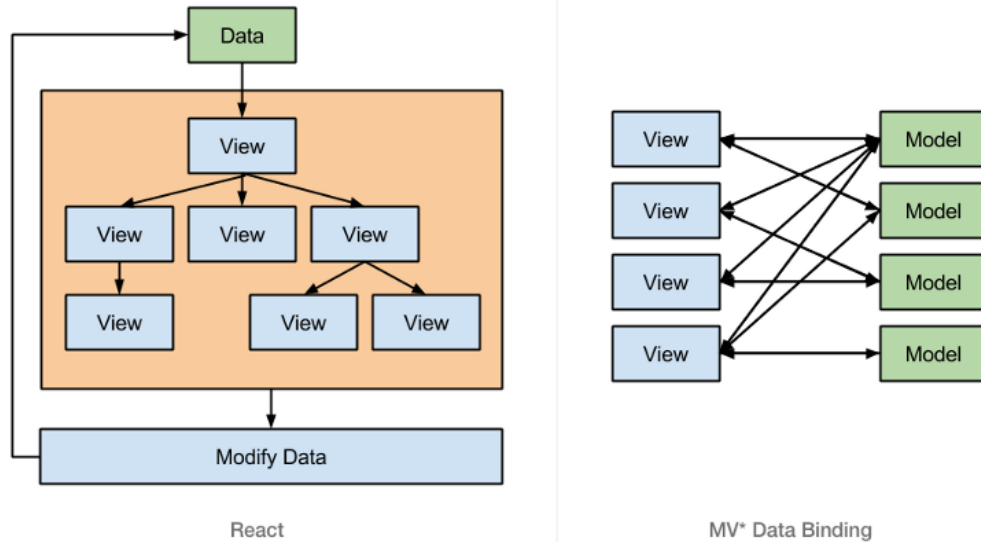
**Easy to migrate** - Since React is lightweight and represents the view layer of the application, it can be used independently and progressively without requiring a huge rewrite for the entire application. This allows React to fit into any existing application and move forward with existing technology stack.

**High Performance** - React uses the advanced concepts such as virtual DOM to perform well in large scale high interactive applications. Since React has a very simple but powerful set of rules, it's hard for developers to forget them and also write react components in a non-performant way. Due to its Isomorphic feature, React applications can be rendered at the server side to reduce the time for the initial drawing in the browser.

**Code Reusability** - Since React is all about reusable and composable UI components, it promotes developers to write isolated reusable components with generic interfaces. This drastically improves the reusability of the code which leads to saving lots of development time in larger projects.

**Increase developer effectiveness** - When developers write React components they have to separate the concerns not the technologies which makes all the HTML, JS and CSS code live together[16]. Since all the code is encapsulated inside a component with well-defined interfaces and the whole application is built as a tree of such components, it's really easy to read and understand the behaviors of the individual components and their data flows to understand the whole application. Facebook has provided a bunch of developer tools to monitor the lifecycle of react components which makes it further easy to track the dataflow and spot the errors in the components. The encapsulated React components make it easy to test them independently which allows the developers to follow TDD from the beginning leading to a highly tested code. Following diagram shows the difference of data flows between react and a traditional MV\*(MVC, MVVM or MVP) application. The data flow in the traditional MV\* applications become very complex by passing models back and forth between many views. It makes the application hard to predict when the application grows. React provides a much cleaner top down unidirectional

data flow which makes it easy to predict and debug. Facebook Ads team and Netflix developer team have confirmed that they saved tons of developer hours and increased the team effectiveness after migrating to React[15].



*Figure 2: Data flow in a react application and a MV\* application*

Faster developer onboarding - Since React provides a simple set of concepts, any developer can easily absorb its concepts which keeps the learning curve very minimal compared to other frameworks. At the same time, since React is in mainstream web development, lots of developers are already familiar with the library and the concepts[17]. These two reasons drastically reduce the cost of new developer onboarding.

All these facts confirm that React is the way to move forward and its effectiveness from both technological and business point of view.

## 1.6. Problem

Due to the increased complexity and the high expectation of the modern web applications, the legacy applications built on top of previous generation frontend frameworks such as AngularJS fail to serve both end users and developers. The AngularJS based legacy applications start poorly performing when the application grows larger. At the same time, they become highly challenging for developers to

maintain and add new functionality over time. The next generation component based frontend frameworks such as React has outperformed AngularJS from both the end user and developer expectations. Since the whole industry has evolved to gain the benefits of the component architecture, sooner or later all these legacy applications need to migrate to new component based architecture.

Even though there are plenty of tutorials and guides to learn and kick start new projects using the latest component based frontend frameworks, there is very limited research done on how to migrate existing legacy frontend applications to newer component based architecture. There are lesser resources and guides available over the internet which describes the approaches, steps, best practices, technologies, tools and pain points of this migration. Even the available few resources are full of application specific details and approaches which are very hard to generalize as guidelines for industry standard applications.

### **1.7. Objectives**

The main objective of this research is to provide a framework and a tool to migrate legacy AngularJS applications to component based React Architecture. The main objective is broken down into sub objectives as follows.

1. Propose a migration framework
  - a. Compare approaches of legacy application migrations and propose the overall approach to the migration
  - b. Extract the steps, best practices, technologies and tools followed and pain points encountered by the previous migration attempts
  - c. Generalize these extracted information as refactoring steps so they can be commonly applied in any AngularJS to React application migration
  - d. Group these migration steps based on their purpose and the stage of applicability
  - e. Organize and present these refactoring steps and the groups in a workflow so it gives a clear direction to the lifecycle of migration

2. Build an assistant tool for developers to automate the migration steps in the framework
  - a. Select the automation strategy to follow when building the assistant tool
  - b. Select automatable refactoring steps out of the giving refactoring steps in the framework with reasoning
  - c. Select technologies and tools which can be used to build the assistant tool and scan the code and detect the refactoring in both JS and HTML code
  - d. Develop the assistant tool to support both command line execution as well as integrated with popular IDEs
  - e. Analyze different coding styles followed by the AngularJS developers and come up with set of algorithms to detect the refactoring in the code
3. Evaluate the framework and the tool by using it to migrate selected AngularJS applications
  - a. Define and select the applications to evaluate the proposed migration framework and the assistant tool
  - b. Define the criteria and the metrics to evaluate the migration
  - c. Follow the migration framework and use the assistant tool to migrate the selected applications and collect the values for defined metrics
  - d. Analyze the results collected and validate the effectiveness and the accuracy of the migration framework and the assistant tool

## 2. LITERATURE REVIEW

### 2.1 Choosing the migration strategy

With all the advantages discussed in previous sections, though React seems to be the way forward, there is no silver bullet to migrate a large scale legacy codebase to React. It requires conscious planning about which code migration approach to follow, which supportive technologies and tooling to be adopted, how to align the teams around it and more importantly how to minimize the business impact of the migration.

The most impactful decision is to decide which approach to follow when migrating the code because it defines all the subsequent activities. There are few well known code migrations strategies practiced in the industry which are,

**Bing-Bang approach:** which requires to convert the whole code base from the old technology to the new technology in a single attempt. This approach gives more flexibility to layout a fresh architecture and tooling which is totally independent from the old architecture and the tooling. Even though it seems much cleaner, this type of a project incurs a huge amount of resources and cost and a high failure rate. This approach prevents the product getting new features because the entire development team is rewriting the existing features which doesn't really add a visible benefit to the existing or new customers. This makes a huge impact to the business and more often the business will not agree to this approach [18].

**Side-by-Side approach:** which uses entry level separations to the old application and new application which makes it possible both the applications run side by side [18]. Developers will develop the new feature in the new architecture and will maintain the old features using the old technology. Old features will be rewritten using the new stack phase by phase aligning them with future proposed improvements. This doesn't have the disadvantage of the Big-Bang approach and it allows the developers to buy more time from business to complete the migration

without really impacting the business. But this approach forces the developers to duplicate components, codes and logics which were already there in the old application to the new application. As an example if there is a list component in the old technology, the developers will have to build an exact similar list component in the new technology stack to cater new pages. Duplication is a huge disadvantage of this approach. Also the developers have to be familiar with both technology stacks until they get rid of the old. This issue will be there when training the hires.

**Bottom Up approach:** which allows the developers to slowly integrate the new technology into the existing technology stack. The new technology will be applied to small parts of the application where the changes can be isolated without breaking the full application. This requires refactoring the existing code but once a particular component or a part of logic is moved to a new stack, it can cooperate with the rest of the application without a visible impact for the features [18] [19]. This allows to avoid the drawback of code and feature duplication which was there in the side by side approach. As an example, if there is a list component in the old technology stack, a developer can refactor it and move to a new technology stack while keeping its interfaces unchanged which allows it to work the same as before inside the old technology stack. This approach will allow the developers to deal with manageable size of problem without duplicating the code while keeping the business impact low. Not all the technology migrations can follow this approach and it'll only work if the new technology stack can coexist within the old technology stack.

## **2.2 Prepare AngularJS code for migration**

Usually a migration will start with the less complicated pieces in the legacy application. The easiest code to start the migration will be the view layer of the legacy application which are templates and directive. Directives in AngularJS are closer to the components in React and if the templates can be broken into meaningful directives, those directives can be easily migrated to components. If the directive templates are larger to manage, those can be broken further into child directives. These small changes performed in the application don't change its functionality but

makes its code easy to work with. These small steps are called refactoring which is a concept popularized by Martin Fowler with his pioneering book called “Refactoring” [5]. There are few simple but effective refactoring which are needed to make the existing application ready for the migration. Those are explained below [5].

### **2.2.1. Place all the code of a directive together.**

Directives are future candidates to React components and all the pieces of a React component lives inside a single JS file. It makes the future migration easier if we have the directive code in a single place in the codebase. This step proposes you to move existing directive code for separate folders and follow the same for the newly introducing directives. The folder structure and the file distribution will be as follows.

components/like\_button

-> like\_button\_directive.js

-> like\_button\_controller.js

-> like\_button.htm

### **2.2.2. Favor directives as elements over attributes, comment and classes**

In AngularJS the directives can be defined as elements, attributes, comments or classes. This makes it very difficult to read and understand the templates and in modern libraries the custom components can be defined only as elements. There are many inbuilt directives provided by AngularJS which are defined as attributes and at this stage we don't expect to refactor them. Check whether custom directives are defined as attributes, comments or classes and if there is no blocking reason, convert them as element directives and try to achieve the functionality using plain JavaScript. This'll make the future migration process straightforward [4].

### **2.2.3. Eliminate Transclusion**

Transclusion is one of the most complicated features provided by AngularJS and it makes the template behavior very inefficient and difficult to understand [20]. The React doesn't natively support Transclusion or promotes it so better to avoid Transclusion as early as possible. Check the existing AngularJS directives and check

whether they perform Transclusion. If yes, alter the directive behavior to avoid Transclusion and use data passed from the inbound scope to achieve the same behavior [4].

#### **2.2.4. Replace ng-include with a directive**

The ng-include directive is used to inject a separate html template within an existing template. This clearly indicates that the injected html is a piece of UI which is an ideal candidate to extract a separate component in future [21]. Extracting it as a separate directive is an ideal step to follow with lesser risk of breaking the existing functionality. In AngularJS it's possible to have the ng-include templates as expressions so sometimes the template URL can change dynamically loading different templates at different times. If the application has such a behavior, as the first step, a conditional logic can be used to switch between templates inside the derived directive. Create a directive named according to the template, and inherit the scope from the parent so you don't want to worry about isolating the scope or debugging which data from the scope are referred inside the template. Move the template to the same folder dedicated to the directive and set the template as the directives template. Finally replace the ng-include with the new directive.

#### **2.2.5. Extract root controller of a directive**

If there is an already existing directive which has a controller bind to the root or very near to the root of its template, extract that controller as the controller of the directive. Move the controller file inside the directive folder, rename to match the directive name, refer it in the directive with the Controller Alias and finally remove the controller directive from the template. This helps the application to follow a consistent pattern of directives with controllers and templates.

#### **2.2.6. Extract controller as a directive**

If there are controllers bind using ng-controller directive other than at the root of a template, those controllers are clear indications to components which can exist on their own. Those segments should be extracted as directives. Create a new directive with a closer name to the controller name and inherit the scope from the parent.

Move the controller to the same directive folder and set the controller to the directive with Controller Alias. Move the HTML content of the child elements into the directive template. At the end replace the ng-controller directive usage with the new directive.

#### **2.2.7. Extract directive from large templates**

One main best practice of components is that a component should be small enough to manage and reuse easily. If there are large templates which do many things in the AngularJS application, there is no way to migrate that into a single component in React. It's obvious that these templates should be broken into directives with a manageable size. Analyze the template and identify segments which are reusable and composable. Use those segments to create directives and move that HTML segment to the directive template and inherit the parent scope. As an example, if the application has a large list directive with pagination options, pagination options can be extracted as a pagination directive which can be reused in any other list. This practice increases the structure and the reusability of the code in the application.

By applying above 7 changes, the code structure can be drastically improved towards component style. But still we didn't touch the data flow of the application which is the most complex area to refactor. Without refactoring the data flow, still the application follows the scope soup pattern. Let's look at what sort of refactoring is available to make the data flow better.

#### **2.2.8. Replace external reference with bound inputs**

The extracted directives in the previous refactoring steps are separated from code but still they may be referring to the various external data and functions which prevents them becoming isolated. After converting directives to React components, React components retrieve all its data and functions as properties passed from parent. This makes it required to improve the directives in a way where all the necessary data and functions are explicitly passed to them. The way to pass these inputs to the directives is as scope bindings[22]. Firstly, you have to go through the directive controller and template to identify places such as expressions, \$watches or any other piece of logic

which refers to the parent scope. Use explicit binding(= binding) for these data in the bindToController section of the directive and replace the external references in the directive with the bindings directly. Then update the template which includes the directive to pass the values to bindings.

#### **2.2.9. Replace external effect with bound outputs**

External references are used not only as inputs but also as methods to notifying outside about the changes done inside the directives. Previously these might have been directly done as executing external functions. The React components pass the changes back to parent components using callbacks passed from the parents as props so to make our directives migratable to React components, we have to make the directive outputs also explicit. Firstly, go through the controllers and templates to identify places which call external functions and add as expression bindings(& binding) inside the bindToController section of the directive. Update the external function calls to refer to new bindings finally update the template to pass the binding to the directive[23].

#### **2.2.10. Isolate directive scope**

Once the refactoring steps mentioned in 2.2.8 and 2.2.9 is followed, your directives automatically transferred into a state that its internal code doesn't deal with external context except for the data passed and retrieved through bind variables. This makes it possible to isolate the scope of the directive[24]. This can be done by updating the scope: true to scope: {} in the directive definition.

#### **2.2.11. Replace state mutation with bound output**

In the 2.2.10 we were setting the obvious outputs as the bound outputs but it doesn't cover all the external effects from a directive. As mentioned in previous sections, one of the main anti-patterns in AngularJS is that the state is mutated all over the application including directives [24]. This makes it really hard to predict the impact of a directive and prevent it from being reused in other places of the application without reviewing the internal code of the directive. In order to make the directive composable and reusable, it's a must to make these state mutations explicit. The next

step of this refactoring is to identify what are the state mutations done inside the directive and move them as bounded output to the scope so the rest of the application knows the directive depends on this particular scope attribute. Go through the directive controller and the templates to find the state mutations, add them as new bound outs in the scope and update the directive code to refer to the new output bindings instead of direct parent scope. Finally update the template to pass the value for bound output.

#### **2.2.12. Replace two-way data binding with one-way data binding**

All the last 4 change steps made for the data flow made external effects from a directive explicit but it doesn't guarantee what will be used as inputs and what will be used as outputs from the directive because the binding we used still uses two-way data binding with '=' expression. Without restricting the binding type, there is a possibility that future modifications for the directive can mix the bounded inputs with outputs and lead to the scope soup back. So it's time to eliminate two-way data binding in your directives and enforce one-way data binding which will align with React component style. In AngularJS directives, you can define the one-way data binding using '<' syntax at the scope binding so update the "=" bindings to "<" bindings [25].

One major side effect of this is that the updated values from the parent scope no longer automatically updates the references inside the directive after the first time. So it requires additional watches to be placed inside the directive for the bindings which are supposed to change over the time in the parent scope.

These 5 refactoring steps make the directives easily composable and reused in the rest of the application without worrying about its internal references.

After following the refactoring steps mentioned in the above sections, now the AngularJS application is ready to get its first touch with React components. As explained in the previous sections, next step is to introduce React for the simplest directives in the application. If we consider the application as a tree of directives, the

directives at the bottom of the tree are considered as the lesser complex directives so the conversion should start from those directives.

### **2.2.13. Convert the directive template as JSX in react component**

As the first step of the conversion, we should move the directive template which is written in HTML to the component which is written in JSX. JSX is an XML-like syntax you can write in Javascript which will be interpreted to HTML at runtime and React is mostly recommending to use it in their components. The converted JSX code should be moved to the render method in the component. The JSX syntaxes and the React component guidelines are out of this research scope so will not cover them in detail here [26].

### **2.2.14. Replace the AngularJS specific features used in the converted component**

The directive templates will have many AngularJS features which need to be replaced by JS specific features or the react alternatives. These angular specific features are not available as it is in the native Javascript or React. Before starting the conversion, it's better to get some understanding about the alternatives which can be used instead of the angular specific features. Following table list the alternatives [4].

| <b>AngularJS</b>         | <b>React</b>          | <b>Description</b>                                                                                                                                                                                                     |
|--------------------------|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \$http                   | Fetch or Axios        | \$http is used to perform remote http calls. This is one of the most used services in AngularJS. Fetch library will be the better choice but if you still need to use a closer library to \$http you can choose Axios. |
| \$q                      | Native support in ES6 | \$q is used to get promised support in AngularJS. In ES6 promises are supported natively so that would be the default fallback.                                                                                        |
| \$timeout/<br>\$interval | Native timeout        | This was mostly used in AngularJS to synchronize with the digest cycle and there is no need to                                                                                                                         |

|                                                      |                                                                         |                                                                                                                                                                                                |
|------------------------------------------------------|-------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                      |                                                                         | do so in React. If you need timeout functionality you can use the native function.                                                                                                             |
| \$document/<br>\$window                              | Native document and window                                              | \$document and \$window were wrappers for native document and window objects. There is not much value for them outside Angular so better to fallback to native.                                |
| \$formatters/<br>\$validators/<br>\$parsers/         | Vanilla JS to format, validate and parse data.                          | The formatters, validators and parsers were widely used due to the nature of two-way data binding in AngularJs. With React, it's possible to use vanilla JS.                                   |
| ng-form                                              | Vanilla JS                                                              | Ng-form is used to have logical groupings inside forms to determine the form validity based on individual group validity. This should be achieved with Vanilla JS or using Redux at the store. |
| Angular.element                                      | React-dom or JQuery                                                     | Dom manipulations are strongly discouraged in the React but if you continuously need to do dom manipulations, you can use React-dom or JQuery.                                                 |
| angular.forEach,<br>angular.extend,<br>angular.merge | Lodash                                                                  | These functions are used to maintain objects which can be now achieved through Lodash.                                                                                                         |
| Filters                                              | Vanilla JS                                                              | Use vanilla js Array.filter() option to archive the behavior needed.                                                                                                                           |
| ng-repeat                                            | Vanilla JS                                                              | Use vanilla js Array.map() option to archive the behavior needed.                                                                                                                              |
| ng-class, ng-show,<br>ng-if,<br>ng-hide              | Vanilla JS conditions                                                   | All these directives were used to manipulate the UI in Angular which can be achieved through vanilla JS conditions inside the React component.                                                 |
| Ng-minlength,<br>ng-maxlength,<br>ng-disabled, ng-   | Use minlength, maxlength, disabled and required Default HTML validation | These validations are available in HTML api so better to switch back to these native validations. If any                                                                                       |

|                                           |                                                |                                                                                                                                                                                                                                                                                           |
|-------------------------------------------|------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| required                                  |                                                | advanced validations are needed, a proper validation library can be integrated.                                                                                                                                                                                                           |
| ngRoute/<br>angular-ui-router             | React router                                   | Use React router for routing your single page application                                                                                                                                                                                                                                 |
| Protractor                                | Enzyme,<br>Nightwatch.js                       | Protractor was used as the testing tool which is coupled with AngularJS and you can use Enzyme for unit tests and Nightwatch.js for end-to-end tests which works well with React.                                                                                                         |
| \$httpBackend,<br>Dependency<br>Injection | Jest or Rewire                                 | There is no dependency injection natively supported in React, but it has Jest and rewire which allows to mock the js modules which was the main use of dependency injection.                                                                                                              |
| ng-animate                                | react-addons-css-transition-group              | ng-animate is used to achieve css based animations and reacts react-addons-css-transition-group can be used as an alternative.                                                                                                                                                            |
| Ng-init,<br>\$onDestroy                   | componentDidMount(),<br>componentWillUnmount() | Ng-init was used in angular forms frequently to execute the initialization code of the form and \$onDestroy when directive scope is destroyed. Now the initialization can be done at the componentDidMount() function and handle the unmounting logic at componentWillUnmount() function. |
| ng-model                                  | defaultValue                                   | Since two way data binding is not used, the value of a react component is set as defaultValue and any changing values are handled through internal state                                                                                                                                  |
| ng-click                                  | onClick                                        | Ng-click was used to handle the click event in AngularJS and this can be handled using the onClick attribute in react components.                                                                                                                                                         |

*Table 2: Alternatives for AngularJS features[4]*

### **2.2.15. Migrate the directive bindings as component props**

Since the previous refactorings help to isolate the directive scope, define the input and output bindings and enforce one-way data binding, the directive was self-explaining the exact inputs and outputs of it. When converting the directive to a component we have to maintain the same inputs and outputs as it was there in the directive. React components use props to define the inputs and outputs of a component so the directive bindings can be directly converted to props in the component. When defining the props of a component the recommendation is to define their types in the PropTypes section which help the program to validate as well as programmers to identify the inputs and outputs supported by the component [27]. Once those props are defined in the component, the internal references of them in the component should be updated.

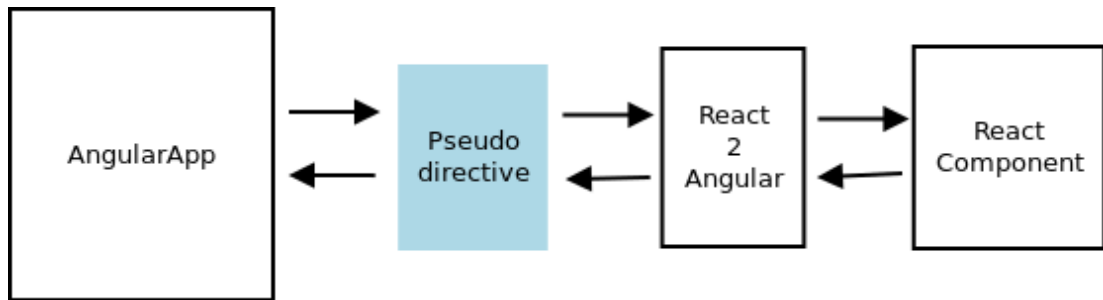
### **2.2.16. Migrate the directive logic into the component**

The final step of the component migration is to move the internal logic in the directive into the component. The directives generally do not depend only on inputs from their parent directives but also have their own logic to process inputs and the data to define how the directive should operate. These logics mostly reside on the functions inside the directive controller or in link function. Some of these logic may use angular specific features as well pure JS features. Moving this logic into the AngularJS component should be carefully done because the lifecycle event system is totally different between AngularJS and React. Due to these reasons, migrating the logic would be the most complicated as well as most error prone step of the whole migration so it should be done very carefully.

The above refactoring will result in a react component which is ready to be used in a react application. Now let's discover how that can be used with the non-converted AngularJS application.

### 2.2.17. Bind the React component to AngularJS using React2Angular middleware

When following the process of migrating the application component by component, converted React components need to work with AngularJS application for a while. Higher order AngularJS directives need to manage state and communicate the state to React components which will render the UI. There should be a bridge between these two technologies to achieve this cohesiveness. React2Angular[31] is an angular module which offers a solution for this problem by wrapping the React component as an angular directive. During this research we will consider React2Angular as the bridging library between angular and react components. Following diagram explains how the Angular2React middleware operates.



*Figure 3: How data flow happens through React2Angular middleware*

Primarily AngularJS directives and React components operate in two separate architectures and technologies. React2Angular middleware helps to wrap a React component in a pseudo Angular directive so the existing AngularJS application can communicate with an Angular directive instead of a React component. The pseudo directive exposes the react component props as scope bindings and acts as the bridge between Angular application and the React component. This makes it possible for React components to seamlessly interoperate within AngularJS architecture.

### 2.3. Migrate component by component

Based on the functionality and purpose, there are two types of components in a react application.

1. Presentational Components which are called as dumb components where their responsibility is to render the UI according to the passed state through properties of the component and pass the user interactions back to the application through the events. If we visualize the application as a tree of components, presentational components will be there at the bottom of the tree and will worry only on how things look [28].
2. Container Components which are called as smart components will usually communicate with backend servers and the services. They are used to communicate with services which manage the application state and pass them to presentational components. The container components will retrieve the actions coming from the presentational components and will dispatch back to services. Container components will be mostly there on top of the component tree and will mainly worry about how things work [28].

To make things simple the plan is to start the migration with less complicated components. The presentational components are the easiest to migrate because they are just responsible for the UI of the application and don't directly deal with the backend services. Presentation components staying at the leaves of the component tree. In order to identify the presentational component, following rules can be used.

1. The templates of the angular controllers
2. The directives with templates

The container components are higher order components which involve backend services and routing. In order to identify the container component, following rules can be used.

1. The directives with controllers attached to the routes but don't have templates
2. The directives which contain business logic
3. Event handling of a directive will be candidates for container components.

Once you categorize the directives as candidates for the presentational components and container components, start the migration with presentational components. Use the information in Table 2 to identify the alternatives for the existing angular specific

features and replace them with the alternatives. Once the React component is ready to deploy, inject the React component to the AngularJS application using React2Angular bridge. We can further look into the technical details of this in the Implementations chapter.

### **2.3.1. Extract container components from presentational components when too many properties passed down through the intermediate components**

When converting your presentational components, you may realize at some point that you are passing too many properties down through the intermediate components without them being used internally. This makes it worse whenever the child needs some additional data and you have to rewire all the intermediate components to pass that piece of new data. At this point you can introduce some container components to fetch the data from the store and pass it to the presentational component without messing up the unrelated intermediate components [28].

Once the Presentational Components are migrated the next step is to move into Container components. These container components will have to communicate with external services. Apart from those aspects, the rest of the container component migrations will be similar to presentational component migration.

When migrating container components, you'll find services which are used in the angular directives. There are few approaches to follow in the service migration based on the nature of the service.

### **2.3.2. Convert the angular service as a Javascript module and import it in the container component**

If the service is no longer used in the AngularJS application after migrating this particular container component, there is no use of having it as an AngularJS service anymore. Remove it from angular service registration, convert this service as a Javascript module and import it to the container component. When converting the service, eliminate the references for angular specific features such as \$q, \$log and \$http used inside the service by referring to Table 2.

### **2.3.3. Wrap the angular service as a Javascript module and import it in the container component**

If the service has functionality which is used in many other places of the application, there is no easy way to migrate the entire service to new architecture. There we have to make sure we keep the existing AngularJS service while reusing it in the React application till all the references are migrated to new architecture. This can be achieved by creating a wrapper Javascript module which wraps the AngularJS service and imports to the container component. React2Angular middleware also supports to directly pass the AngularJS services as props to the bind react components which can be used as an alternative approach [29]. This will allow us to continue the migration without disrupting the existing application functionality.

## **2.4. Migrate the forms**

With the migration of presentational components, container components and services, the application will have the majority of its code converted to React. The converted parts mostly represent the features which display the data passed from the services. Remaining features represent the data mutations in the application and out of them Forms are the most difficult part to deal with as usual. Forms contain various types of widgets in it and these widgets belong to various types of explicit and implicit behaviors including validations. The pure HTML widgets such as inputs, selects and text areas used in the existing forms doesn't need any migration and can be directly used in React. Let's look at how to approach migrating the remaining elements in the form.

### **2.4.1. Introduce a UI library which provide React favored form widgets**

If the AngularJS application contained many forms and complicated widget behaviors in it, definitely the application should have been using a custom UI component library such as MaterializeCSS[32] or Angular Material[33]. There are React alternatives for these libraries such as Material UI[34] so the most logical and

low cost choice would be to switch your component library rather than migrating widget by widget.

#### **2.4.2. Migrate widgets with custom directives to components**

If there are in house developed widgets which use custom directives, those need to be migrated to new architecture and most of them will fit into the presentational components and few of them will fit into container components. If the widgets contained attribute directives, those should be migrated into vanilla JS inside the new components. It's important to migrate these widgets as individual components so they will be composable when building present and future forms.

#### **2.4.3. Migrate formatting, sanitizing and validations to services**

There might have been many formatting rules and validation rules implemented within the custom widgets or services. Those should be moved to application services with the idea of reusing them in other parts of the forms.

Once the individual form elements are converted to React, create a container component which can consolidate the individual widget components into a meaningful form structure. Since it's a common pattern in any form used in the product, implement the container component in a generic level which is reusable throughout the product and can accommodate the different form structures using configurations.

These micro refactoring steps give step by step guidance to follow when the path is not clear and risk is high. These steps will be helpful for someone to kick start the migration project. Once the developers are familiar with the process and when they get to spread they can combine few of these steps in a single go and do things faster but always they can fall back to these micro refactoring whenever they find the next steps are not clear.

### **3. METHODOLOGY**

First objective of the research is to choose the migration approach. Due to the high competition in the web application industry, it's not advisable to follow the Big-Bang approach to do this technology migration which will bring deadly consequences. Since the side by side approach promotes huge duplications in the code bases, it can impact the stability of the product in the long run if the business priorities change from the originals. One main advantage of the Angular and React is that React can coexist inside AngularJS seamlessly by representing the View layer. At the same time React is a very lightweight library which doesn't impact the resource sizes by having it inside the Angular. By considering all these approaches, it seems that the way to move forward is the Bottom Up approach which brings less consequences and a smoother migration.

As described above when migrating an AngularJS application to react using a bottom up approach, there are a series of incremental changes to be followed. Currently there is no consolidated knowledge based to refer when migrating AngularJS applications to React where different developers and teams have followed their own practices or sub set of practices explained in this research but on their own order. Following ad hoc steps can lead to confusions, time wastes, low quality and buggy code. Under this research, the migration steps followed by various parties were analyzed, processed and consolidated as a single knowledge base to guide the teams or individuals who are migrating AngularJS applications to React.

#### **3.1. A migration framework from AngularJS to React**

The identified incremental changes in the migration process can be abstracted into a series of refactoring and present as a migration workflow. The first outcome of this research is to come up with a framework which provides a clear workflow for someone who is migrating an AngularJS application to React. This framework will be the go to place and the reference knowledgebase for the migration process.

The incremental changes discussed in the literature review section were broken down to 23 steps and generalized them as universal refactorings which can be followed when migrating AngularJS application to React.

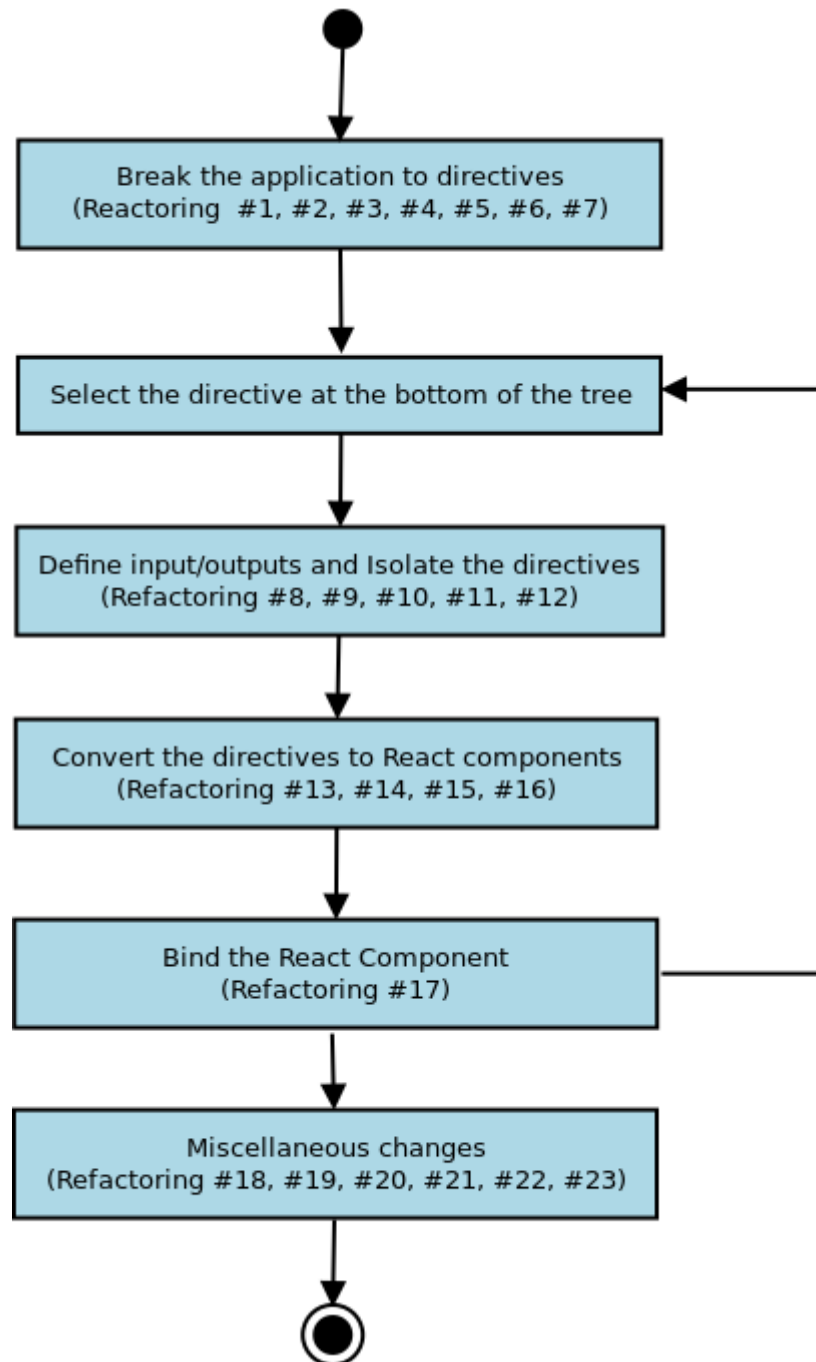
Then, those refactoring were grouped to get a meaningful and easily rememberable framework. Based on the context where this framework is applied, the most appropriate grouping criteria was to group based on their purpose and the time they are applied. Table 3 shows the refactoring and their groupings which are the basis for the migration framework.

| <b>Refactoring</b>                                                                | <b>Purpose</b>                                     |
|-----------------------------------------------------------------------------------|----------------------------------------------------|
| Refactoring #1: Place all the code of a directive together                        | Break the application to directives                |
| Refactoring #2: Favor directives as elements over attributes, comment and classes |                                                    |
| Refactoring #3: Eliminate transclusion                                            |                                                    |
| Refactoring #4: Replace ng-include with a directive                               |                                                    |
| Refactoring #5: Extract root controller of a directive                            |                                                    |
| Refactoring #6: Extract controller as a directive                                 |                                                    |
| Refactoring #7: Extract directive from large templates                            |                                                    |
| Refactoring #8: Replace external reference with bound inputs                      | Define the input/outputs and isolate the directive |
| Refactoring #9: Replace external effect with bound outputs                        |                                                    |
| Refactoring #10: Isolate directive scope                                          |                                                    |
| Refactoring #11: Replace state mutation with bound output                         |                                                    |
| Refactoring #12: Replace two-way data binding with one-way data binding           |                                                    |

|                                                                                                                                                       |                                            |
|-------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------|
| Refactoring #13: Convert the directive template as JSX in react component                                                                             | Convert the directives to React components |
| Refactoring #14: Replace the AngularJS specific features used in the converted component                                                              |                                            |
| Refactoring #15: Migrate the directive bindings as component props                                                                                    |                                            |
| Refactoring #16: Migrate the directive logic into the component                                                                                       |                                            |
| Refactoring #17: Bind the React component to AngularJS using React2Angular middleware                                                                 | Bind the React component to AngularJS      |
| Refactoring #18: Extract container components from presentational components when too many properties passed down through the intermediate components | Miscellaneous changes                      |
| Refactoring #19: Convert the angular service as a javascript module and import it in the container component                                          |                                            |
| Refactoring #20: Wrap the angular service as a javascript module and import it in the container component                                             |                                            |
| Refactoring #21: Introduce a UI library which provide React favored form widgets                                                                      |                                            |
| Refactoring #22: Migrate widgets with custom directives to components                                                                                 |                                            |
| Refactoring #23: Migrate formatting, sanitizing and validations to services                                                                           |                                            |

*Table 3: Refactoring and their groupings*

The identified purpose based groups let us map them to an easily understandable migration workflow. The workflow is designed based on the time each refactoring is applied in the process. Following diagram abstracts the proposed workflow of the migration framework.



*Figure 4: Workflow of the migration framework*

This workflow indicates which group of refactoring should be followed in which order to make the migration process easier. Since this is designed on top of the purpose based groups, it's easy for anyone to follow it once and remember during the remaining time of the project. According to the proposed workflow, anyone who is starting to migrate an AngularJS application to React, the initial effort should be put

into making the existing application structured around directives and break the larger parts of the application into small manageable directives. Once the application is structured around directives, its structure will look like a directive tree. Then the directives at the bottom of the tree should be picked for the migration since they are less complicated and more suited for as presentational components. If the selected directive contains many dependencies with other components and if the input/output interfaces are not properly defined, then that should be done as the next step in the migration process. Once the directive becomes an isolated entity with properly defined inputs and outputs it's possible to convert it to a React component. Once the directive is converted to a React component, it will still maintain the same input/output interface as it was in the directive. Last step of this cycle is to replace the angular directive with the converted React component and use the React2Angular middleware to inject the React component at the places where the directive was in use. Once this cycle is done for one directive, the same can be applied to other directives over and over until the entire tree gets converted to React. The miscellaneous changes can be applied independently if those cases are appearing in the projects.

The defined refactorings and workflow are main elements of the migration framework which were derived from the conducted research. The next part of the research is to identify how to automate this migration workflow.

### **3.2. Migration assistant tool**

#### **3.2.1. Choosing refactorings for migration assistant tool**

Following the proposed workflow manually by referring to the framework will be appropriate for very small applications, but it'll be very hard when it comes to complex applications. When there is a large code base, there may be hundreds of places which need refactoring and the developers will not be able to follow the workflow step by step correctly. In order to address this concern, the research was extended to analyze the proposed refactoring and identify what can be automated up to which extent.

The first step to automate the migration process is to identify what can be automated and what cannot be automated. The automation can be done in different levels in the context of this project.

- Fully automated - Automate the workflow fully without intermediate human intervention. By fully automating the workflow, it's possible to save lots of time and energy on the migration. But the risk here is that, since the two technologies operate in totally different ways, it's very difficult to fully automate the workflow without introducing errors. Introducing an error on automatically generated code can mess up the entire workflow since the developer won't be able to figure it out.
- Semi-automated - When it's difficult to fully automate a process without introducing errors, still there is lots of value that can be added by semi automating the process. The isolated areas of the process can be automated as well as suggestions can be provided to the areas which need to be manually performed. This will let the process gain advantages of the automation while having the manual control over the risky areas.

As per the scope of this project, semi automation is picked as the approach to automate the workflow which can avoid introducing unexpected errors to automated code. The refactoring proposed in each workflow stage were analyzed to identify what to be automated. Following criteria was used to determine which refactorings will be incorporated in the assistant tool.

1. Selection Criteria #1: Possibility of detecting and performing refactorings automatically without introducing errors. This is must to satisfy.
2. Selection Criteria #2: High time-consuming refactorings when performed manually. This is good to satisfy.
3. Selection Criteria #3: Frequency of each refactoring required to be performed in a code base. This is good to satisfy.

Following table list down the summary of possible automations.

| Refactoring                                                                              | Can auto detect | Can auto correct | Satisfying Selection Criteria |     |     |
|------------------------------------------------------------------------------------------|-----------------|------------------|-------------------------------|-----|-----|
|                                                                                          |                 |                  | #1                            | #2  | #3  |
| Refactoring #1: Place all the code of a directive together                               | Yes             |                  | Yes                           |     | Yes |
| Refactoring #2: Favor directives as elements over attributes, comment and classes        | Yes             |                  | Yes                           |     | Yes |
| Refactoring #3: Eliminate transclusion                                                   | Yes             |                  | Yes                           |     |     |
| Refactoring #4: Replace ng-include with a directive                                      | Yes             |                  | Yes                           |     |     |
| Refactoring #5: Extract root controller of a directive                                   |                 |                  | No                            |     |     |
| Refactoring #6: Extract controller as a directive                                        | Yes             |                  | Yes                           |     | Yes |
| Refactoring #7: Extract directive from large templates                                   | Yes             |                  | Yes                           |     |     |
| Refactoring #8: Replace external reference with bound inputs                             |                 |                  | No                            |     |     |
| Refactoring #9: Replace external effect with bound outputs                               |                 |                  | No                            |     |     |
| Refactoring #10: Isolate directive scope                                                 |                 |                  | No                            |     |     |
| Refactoring #11: Replace state mutation with bound output                                |                 |                  | No                            |     |     |
| Refactoring #12: Replace two-way data binding with one-way data binding                  | Yes             |                  | Yes                           |     | Yes |
| Refactoring #13: Convert the directive template as JSX in react component                |                 | Yes              | Yes                           | Yes | Yes |
| Refactoring #14: Replace the AngularJS specific features used in the converted component | Yes             |                  | Yes                           | Yes | Yes |

|                                                                                                                                                       |     |  |     |     |     |
|-------------------------------------------------------------------------------------------------------------------------------------------------------|-----|--|-----|-----|-----|
| Refactoring #15: Migrate the directive bindings as component props                                                                                    | Yes |  | Yes | Yes | Yes |
| Refactoring #16: Migrate the directive logic into the component                                                                                       |     |  | No  |     |     |
| Refactoring #17: Bind the React component to AngularJS using React2Angular middleware                                                                 |     |  | No  |     |     |
| Refactoring #18: Extract container components from presentational components when too many properties passed down through the intermediate components |     |  | No  |     |     |
| Refactoring #19: Convert the angular service as a javascript module and import it in the container component                                          |     |  | No  |     |     |
| Refactoring #20: Wrap the angular service as a javascript module and import it in the container component                                             |     |  | No  |     |     |
| Refactoring #21: Introduce a UI library which provide React favored form widgets                                                                      |     |  | No  |     |     |
| Refactoring #22: Migrate widgets with custom directives to components                                                                                 |     |  | No  |     |     |
| Refactoring #23: Migrate formatting, sanitizing and validations to services                                                                           |     |  | No  |     |     |

*Table 4: List of refactoring and their automation feasibility*

According to the analysis, out of the 23 refactoring 9 were possible to detect using automation without false positives. Other refactoring can differ based on the application context and the coding style so they were not picked for the initial phase of the automation process. One another refactoring were identified as possible automatic code generations which will help the user to save time and avoid human errors.

### **3.2.2. Technology stack for automating the migration workflow**

At a high level following were identified as the requirements to stratify from the tool.

1. The tool should be able to execute in the command line as well as integrated with popular IDEs since that gives a higher usability. This requires the tool to be built using technologies supported working in both command line and IDEs. Also the internal components of the tool should be implemented in a loosely coupled manner where some of them can be used in both command line and IDEs.
2. The rule implementation in the tool should be done following open closed principles where if a specific team needs to implement their own rules to make the migration easy, they should be able to do it without touching the core of the application.
3. The tool should be able to scan a single file or an entire codebase to find out the recommended refactoring as part of the migration process. When the scan is completed the developer should be able to get the highlights of the scan to assess the code base status and the size of the project he is going to deal with.
4. Once the code base is scanned, the results should be presented as reports so the user can get to know about refactoring needed for the codebase. The report should have a summary view which will give a quick insight about the code base he is dealing with and a detail view which will be used to pinpoint the refactoring needed.
5. There will be certain code generations which can be done safely without introducing bugs. The tool should provide an option to perform such code generations automatically to save the time spent on code migration.

The tool should satisfy the above objectives from it and the technologies were chosen to satisfy the mentioned requirements.

The refactoring detection could be done in two different methods.

1. Static code analysis - scan the source code and detect the refactoring.
2. Runtime code analysis - watch the runtime code and detect the refactoring.

Out of these two methods, static code analysis is considered as the most effective method for refactoring since it gives feedback to developers based on the written

code as it is. Due to that reason, this research implementation uses a static code analysis approach to detect the refactoring.

The refactoring which were picked for automatic detection involved with 3 different technologies which are JS, HTML and JSX. In order to detect these refactoring, it was required to scan the code written in these different technologies. There were two approaches to automate the code scanning.

1. Write a scanner to scan the source code files, extract the content and process the text using a custom set of rules to detect the refactoring. This approach may look like the fastest way to kick-start but when more and more complex refactoring need to be detected, the implementation will become exponentially complex as well as inaccurate.
2. Integrate with existing code scanners which provides the ability to write our own rules to detect the refactoring. This approach needs a bit more research to compare and contrast the off the shelf code scanning tools and their extension capabilities. But once the tools are determined and the learning phase is completed, it'll allow to add new rules faster and even will give the possibility to reuse the existing rules to help out.

For this research implementation reusing the existing code scanner approach was selected by considering the advantages even though they have a rigorous learning curve. There were no existing tools which provided the code scanning for all JS, HTML and JSX code at the same time.

ESLint[35] was the industry standard tool for JS code scanning and there were plugins to enable the JSX scanning in it. The extension capability of ESLint is very powerful where custom rules can be implemented and plugin so the scanner automatically picks them up and incorporates them in the scanning process. Furthermore, it was possible to integrate with almost all the modern IDEs which will highlight the results generated by the ESLint in the IDE itself. ESLint provides a powerful API which allows to integrate it within custom applications so it can be

used in the proposed command line application to scan code. All these advantages, ESLint was selected as the JS and JSX scanner for this research project.

When scanning source code with ESLint, it first parses the code using espree[37] Javascript parser. The parser returns the Abstract Syntax Tree(AST) of the source code where ESLint goes through the AST to detect the patterns in the code. When writing rules for ESLint, the rules should be written based on the generated AST which adds a considerable learning curve when extending the tool. In the example given below, if we have to check whether the name of the variable declaration according to standards we have to write a rule against “VariableDeclarator ” node type and get the node name and compare with the defined standard. The ESLint will run that rule every time it finds a “VariableDeclarator” node in the given AST. There are many node types which the AST provides and the ones which were related to the implemented rules will be discussed under the relevant sub topic.

| JS Code                     | AST                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| var a = "This is a sample"; | <pre>{   "type": "Program",   "start": 0,"end": 27,   "range": [ 0, 27 ],   "body": [     {       "type": "VariableDeclaration",       "start": 0, "end": 27,       "range": [ 0, 27 ],       "declarations": [         {           "type": "VariableDeclarator",           "start": 4,"end": 26,           "range": [ 4, 26 ],           "id": {             "type": "Identifier",             "start": 4,"end": 5,</pre> |

|  |                                                                                                                                                                                                                                             |
|--|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  | <pre> "range": [4, 5 ], "name": "a" }, "init": {   "type": "Literal",   "start": 8,"end": 26,   "range": [8,26 ],   "value": "This is a sample",   "raw": "\"This is a sample\"" } }, ], "kind": "var" } ], "sourceType": "module" } </pre> |
|--|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

*Table 5: Sample AST for JS code*

HTMLHint[36] is a tool which was developed to scan HTML code and that was the only lightweight library which was available to suit this purpose. It provides the extensibility where custom rules can be implemented and injected to the scanner but there were some limitations to incorporate it with the IDEs when using custom rules. Out of the popular IDEs, HTMLHint supports integration with Atom, Brackets, Sublime and VS Code. HTMLHint also provides an API which allows it to be used in a 3rd party application so it was possible to integrate with the proposed command line application to scan HTML code. By considering these facts HTMLHint was selected as the HTML code scanner for this research project.

Similar to ESLint, HTMLHint also uses an HTML parser to parse the given source code. When the parser generates the AST, HTMLHint provides an event system which triggers when AST is traversed. The rules can be registered as callback listeners for these events and event data can be used inside the rule to validate the source code. As an example if we need to detect whether there are “ng-repeat”

attributes defined for any HTML tag, we have to register a listener to “tagstart” event and iterate through the attributes in the element and match the attribute name as “ng-repeat”. Following are the events provided by HTMLHint.

| Event    | Description                                  |
|----------|----------------------------------------------|
| tagstart | Triggered when a HTML start tag is detected  |
| tagend   | Triggered when a HTML end tag is detected    |
| start    | Triggered at the start of the HTML document  |
| end      | Triggered at the end of the HTML document    |
| text     | Triggered when a text is detected            |
| comment  | Triggered when a comment is detected         |
| cdata    | Triggered when a cdata is detected           |
| all      | Triggered at any of the above event detected |

*Table 6: Event list of the HTMLHint code scanner*

The next step is to select the technologies for the 2 automatic code generations identified in the scope. HTMLtoJSX[38] tool provides a powerful way to convert HTML to JSX which was exactly matching the refactoring step of “Refactoring #13: Convert the directive template as JSX in react component”. It also provides an API to integrate with external tools so that was picked as the tool to migrate HTML code to JSX. Other code migration is to automatically bind the converted React component to AngularJS application which doesn’t have any available tools so it was decided to implement from scratch.

All these tools are using the Javascript technology and at the same time they are providing Javascript APIs to integrate with external applications. Due to these reasons NodeJS technology was selected to implement the consolidated command line tool.

Following table summarizes the technology and tool selection to automate the migration process.

| Technology | Purpose                                                  |
|------------|----------------------------------------------------------|
| ESLint     | Static code analyze for JS and JSX code                  |
| HTMLHint   | Static code analyze for HTML code                        |
| HTMLtoJSX  | Convert the directive template as JSX in react component |
| NodeJS     | Command line tool to automate the process                |

*Table 7: Summary of selected technologies for the migration process*

Now let's see how all these things were put together as a single consolidated tool which can be used in the migration process.

### 3.2.3. Consolidated migration tool

There is an existing tool called Angular Copilot[42] which automates the migration process of AngularJS to Angular which is the latest Angular framework. With the inspiration of this, I have named the AngularJS to React migration tool built under this research as “Ng-React Copilot” where it'll help the developer to do the migration as a copilot while keeping the developer as the pilot.

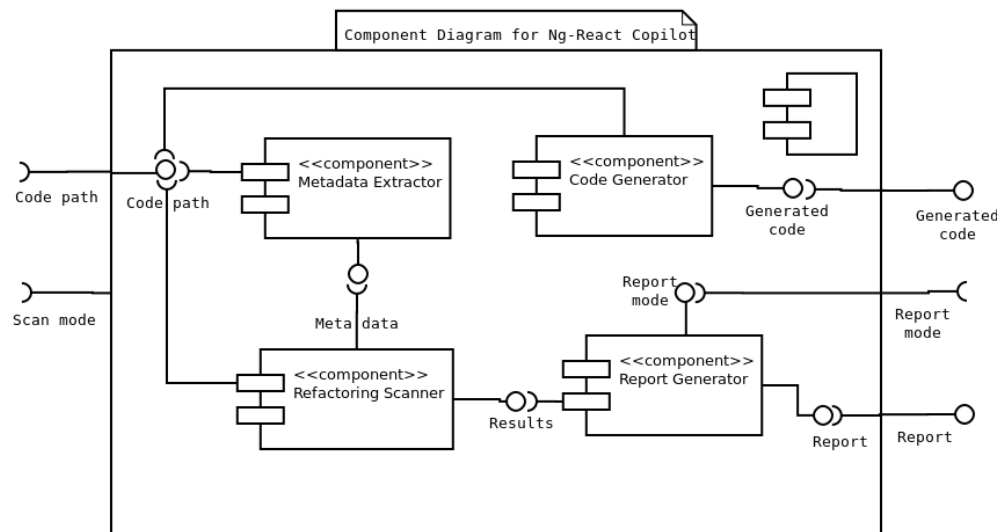


*Figure 5: Ng-React Copilot tool logo in the command line*

As mentioned in the previous sections, this tool contains the following main components.

- Metadata Extractor - analyze the codebase and extract metadata. These metadata are used in the Refactoring Scanner to make the scanning process faster and effective.

- Refactoring Scanner - which scans the given code base and detects the refactoring needed for JS and HTML code. As mentioned in the previous sections ESLint and HTMLHint are used as the scanning technologies for JS and HTML respectively. This component is decoupled so it can be used with popular IDEs without Ng-React copilot.
- Code Generator - do the automated code generation for React Components and inject them into AngularJS. HTMLtoJSX and NodeJs are the base technologies used for this component.
- Report Generator - is responsible to collect the results from the Refactoring Scanner and present them to the user.



Even though the component diagram gives a static view of Ng-React Copilot tool, let's analyze the run time behavior of the tool and the components using a flow diagram.

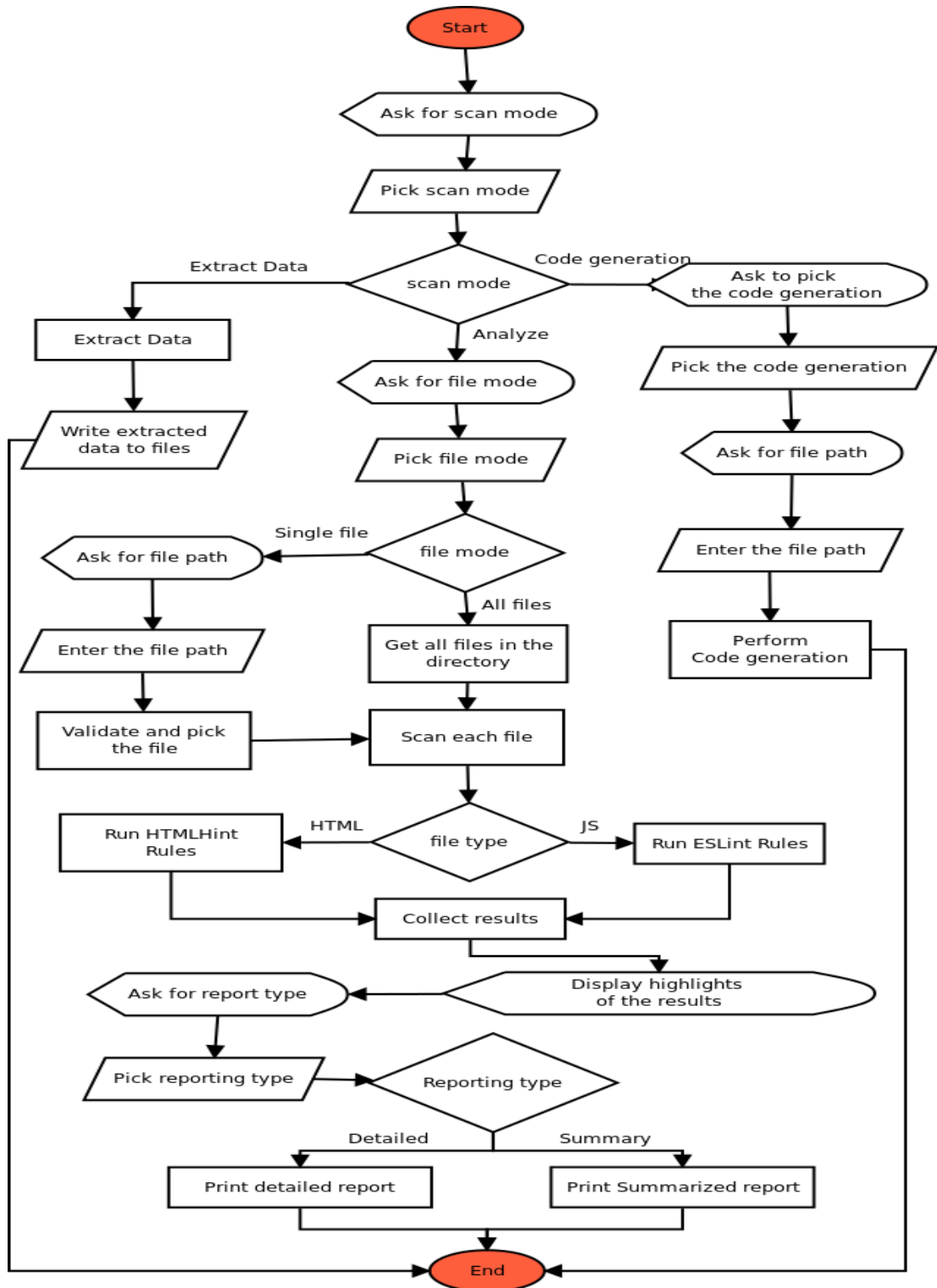


Figure 7: Ng-React Copilot flow diagram

Ng-React Copilot runs on three scan modes as Extract Data, Analyze and Code generation. Even though these modes can be executed independently, the recommended way is to first extract the necessary metadata from a given code base

using the Extract Data option, then scan and detect the refactorings using the Analyze option and then use the Code Generation option to auto generate the code. The implementation details of this flow will be explained in the implementation section.

### **3.3. Set of algorithms to detect the required refactoring**

Even though the rules are part of the implemented tool, they need special attention under this research since each rule implementation needs its own research based on the way and where the code appears. The code scanning and the algorithm detection can be done in different approaches such as plain text matching or use a more structured Abstract Syntax Tree(AST) based walkthrough. By considering the advantages mentioned in the 3.2.2 section, AST based code scanning was selected as the code scan method since it provides a higher accuracy over the plain text based code scanning.

Even though AngularJS is a single framework and all the teams share the same functionalities, the coding style of the projects differs a lot from each other. There are well known style guides to follow in AngularJS applications and the majority of the tutorials also have structured around these style guides. Even with these common tutorials, based on the language familiarity of the developers they tend to use different syntaxes to achieve the same thing. As an example, some developers will use a callback function to register a directive while others will use an arrow function. The “Table 9: Code samples to define an angular using different syntaxes” presents 5 different syntaxes of registering an AngularJS directive so these different coding styles should be considered when defining the detection algorithms. These code samples were gathered by referring the code in Github projects, searching in the internet and using my own experience on how some of the JS coding can be done. Once those coding samples were gathered, the AST for those code segments were generated using the AST generator and then algorithms were developed to walk through all the AST structures and detect the refactorings in a generic way.

As mentioned in the 3.2.1 section, these algorithms were written in ESLint and HTMLHint tools to detect refactorings in the JS and HTML codes separately.

The detection algorithms are presented as pseudo code in the implementation section of the thesis so if any team needs to implement their own rules, they can refer to the pseudo code to come up with their own rules.

### **3.4. Evaluating the migration framework and the assistant tool**

Once the framework is defined and the tool is built, it was evaluated against a selected set of existing AngularJS applications. These application will be selected from three categories which are

1. AngularJS official demo application which contained 27 total files with it so considered under small application category.
2. Applications built by individual developers which contained 89 total files with it so considered under medium application category.
3. Enterprise level applications which contained 1002 total files with it so considered under enterprise application category.

Objective of this evaluation is to identify the applicability and the limitations of the migration framework and the tool. Migrating an application is a highly complex task so it's required to have a clear set of metrics for the evaluation criteria. Following are the metrics defined to evaluate the Migration Framework and the Assistant Tool.

1. **File Search Efficiency** is determined by the number of seconds to complete a single step. As an example how many seconds it took to scan the codebase. This was only applicable in the automation tool.
2. **Ease of following the migration workflow** is determined by whether the developer faces any deadlocks or whether the next step is clear while following the migration framework or the assistant tool.
3. **Accuracy of the refactoring detection** is determined by whether there are any false positives detected during the scans. This was only applicable in the automation tool.

4. **Top time savers in automated migration** is determined by identifying the top time saver migration steps automated in the assistant tool. This helps to determine the effectiveness of the automation process.

Once the evaluation is completed, the research can recommend which sort of application migration can follow the proposed framework and the tool. The evaluation will further look into the manual tasks involved in the migration process and will determine what could be proposed as future automation and enhancements.

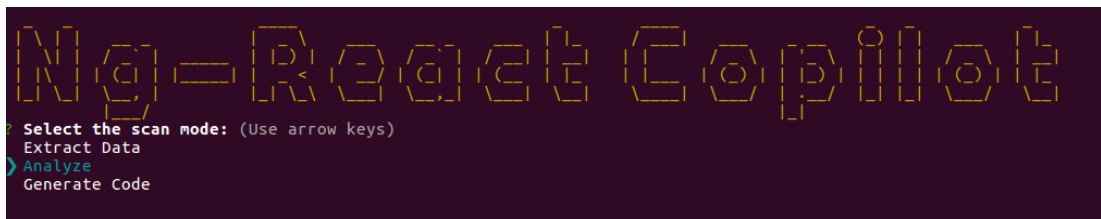
## 4. IMPLEMENTATION

This section explains the implementation of the migration assistant tool and the implementation of the refactoring detection algorithms.

### 4.1. Implementation of the Ng-React-Copilot tool

Following is the high level technical details of the Ng-React-Copilot tool.

- A developer can install the Ng-React Copilot tool by directly downloading from Github and installing the necessary packages by running “npm install” command. The developer can input the directory path to scan as the first argument when running the tool as “node index.js <file path>”.
- Ng-React Copilot runs on three scan modes as Extract Data, Analyze and Code generation. When the tool is executed, it asks the user to pick the scan mode.



*Figure 8: Ng-React Copilot select scan mode*

- As the first step, the developer has to run the tool in the Extract Data mode which will extract the metadata about the project which will be used in the Analysis mode. Those extracted metadata will be written to a file and kept for future references. After migrating each directive in the code base, these metadata should be regenerated using the command.
- When the developer is done with extracting data, he can rerun the tool with the Analyze mode. The analysis can be done for a single file or for the entire code base. The developer will be given an option to pick and then the scan will happen file by file. If the selected file is a JS file, the file content will be passed to ESLint scanner and if the file is an HTML file, the file content will be passed to HTMLHint scanner.

- Once the scan is completed, the tool will collect all the results and show highlights of the results to the user. The highlights contain the number of files scanned, number of files passed and number of files the refactoring needed.

| Highlights (Number of files) |        |                      |
|------------------------------|--------|----------------------|
| Total                        | Passed | Refactoring required |
| 23                           | 19     | 4                    |

The scan results are ready to be exported.  
 ? Select the report type: (Use arrow keys)  
 > Summary report  
 Detailed reoprt

Figure 9: Ng-React Copilot highlights of the analysis results

- Then the tool asks the user which report type to use to present the results. There are two report types available as Summary and Detailed. The summary report groups the results by the refactoring type and shows how many occurrences each of it occurred.

The scan results are ready to be exported.  
 ? Select the report type: Summary report

| Summary Report                                                        |                 |
|-----------------------------------------------------------------------|-----------------|
| Refactoring Id                                                        | Number of files |
| ng-react-copilot/missing-required-props-in-components                 | 1               |
| replace-directive-with-react-component                                | 1               |
| missing-required-props-in-components                                  | 2               |
| break-large-templates-to-directives                                   | 1               |
| ng-react-copilot/replace-angular-inbuilt-directives                   | 8               |
| ng-react-copilot/replace-angular-scope-references-in-react-components | 34              |
| react/prop-types                                                      | 1               |

Figure 10: Ng-React Copilot summary report

- The user can get a detailed list of refactoring detected in each file by selecting the detailed report type. This shows each refactoring separately for each occurrence with the line number so it helps the developer to pinpoint the refactoring.

```

The scan results are ready to be exported.
? Select the report type: Detailed report

/home/thilanka/www/orangehrm-dev-environment/html/OHRMstandalone/TEST/js/angular-phonecat/app/app.config.js
12:21 error The required ctrl,data-type prop(s) should be pass to the phone-detail-component component
rops-in-components

✖ 1 problem (1 error, 0 warnings)

../angular-phonecat/app/phone-detail/phone-detail.template.html
10:1 error Replace angular phone-detail directive with react phone-detail-component replace-directiv
11:1 error The required "ctrl,dataType" prop(s) should be pass to the react component missing-required
119:1 error Break large templates into directives break-large-temp

✖ 3 problems (3 errors, 0 warnings)

```

Figure 11: Ng-React Copilot detailed report

- The third mode of the tool is the Generate code mode where it asks the developer to pick which code generation needs to be done and then some other inputs based on the type of the selected code generation.

That concludes the overall flow of the Ng-React Copilot tool and how it works. In addition to the Ng-React Copilot tool, the implemented rules can be run on IDEs such as Visual Studio by configuring the ESLint and HTMLHint to work with the IDE. The rules are written adhering to the standards provided by these tools so they can be installed as plugins to IDEs. Following Figure 12 shows how a Ng-React Copilot rule is highlighting a required refactoring in a code opened with Visual Studio.

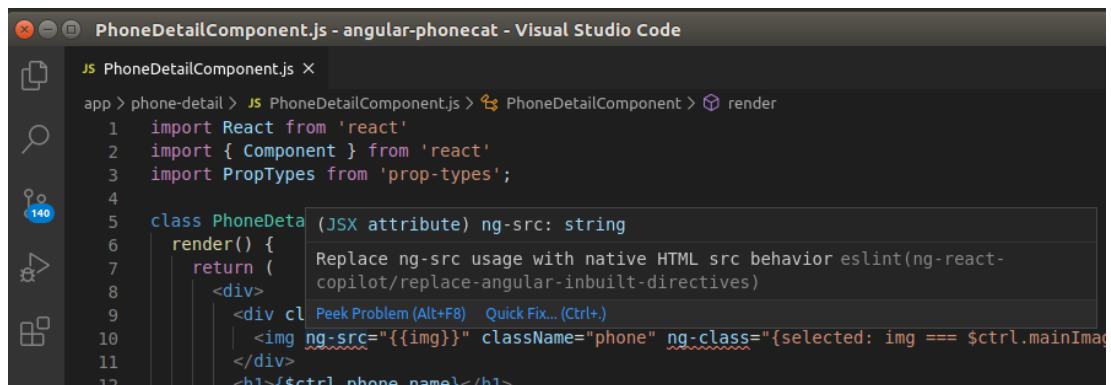


Figure 12: Ng-React Copilot rules highlighting a required refactoring in a code opened with Visual Studio

## 4.2. Implementation of the refactoring detection rules

Once the main architecture for the consolidated tool is put up, the next step was to implement the rules to detect the refactoring. As mentioned before, these refactoring span in three different languages so it was required to use different technologies. This made it required to conduct an analysis for each refactoring before jumping into

the implementation. Following sections explain the details of the analysis and the implementation details of each refactoring. The first rule was explained in detail so the approach taken and the identified obstacles will be clear. Other rules will be explained at a high level using Pseudo code.

#### 4.2.1. Refactoring #1: Place all the code of a directive together

Under this refactoring, it's required to detect the available directives in a given file or a code base. There are two types of directives in an AngularJS application where general directives which were available in AngularJS from the beginning and components which are a special type of directives introduced after Angular 1.5.

| Defining a directive                                                                                                                                        | Defining a component                                                                                                      |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| <pre>angular.module('App', []) .directive('header', function() {   return {     templateUrl: 'header.html',     data: {       data: '='     }   } });</pre> | <pre>angular.module('App') .component('header', {   templateUrl: 'header.html',   bindings: {     data: '='   } });</pre> |

*Table 8: Code samples to define an angular directive and a component*

The AST for the code samples were analyzed using the ASTexplorer[39] tool to identify how to detect the keywords in a given JS file. Following figure shows the AST of a directive code.

```

- body: [
  - ExpressionStatement {
    type: "ExpressionStatement"
    start: 0
    end: 144
    + range: [2 elements]
  - expression: CallExpression {
    type: "CallExpression"
    start: 0
    end: 143
    + range: [2 elements]
  - callee: MemberExpression {
    type: "MemberExpression"
    start: 0
    end: 36
    + range: [2 elements]
  + object: CallExpression {type, start, end, range, callee, ... +1}
  - property: Identifier {
    type: "Identifier"
    start: 27
    end: 36
    + range: [2 elements]
    name: "directive"
  }
]

```

*Figure 13: AST for detecting a directive*

According to the AST the “directive” keyword can be identified by writing a rule to scan for all the “Identifier” node types and checking whether the name is equivalent to “directive”. But just looking for the directive keyword will be somewhat error prone if the same keyword is used for some other purpose in the code. Due to that following further conditions were considered to detect a directive.

- Detect all the nodes with type “CallExpression” in the code
- Check whether the node contains two arguments
- Check where the first argument is a ‘Literal’ type
- Check whether the second argument is a ‘FunctionExpression’
- Check whether the call expressions callee is a ‘MemberExpression’
- Check whether the callee property name is equal to ‘directive’

This gave a higher accuracy to detect the directive keyword which is exactly used to register a directive. Once the rule was implemented, it was tested with different types

of directive registering codes and found few additional syntaxes to follow when defining a directive. Following are the identified patterns of it.

| Directive factory as an arrow function                                                                                                                                                                                               | Directive factory as an identifier                                                                                                                                         |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>angular.module('App', []) .directive('header', () =&gt; {   return {     templateUrl: 'header.html',     data: {       data: '='     }   } });</pre>                                                                            | <pre>angular.module('App', []) .directive('header', header);  function header() {   return {     templateUrl: 'header.html',     data: {       data: '='     }   } }</pre> |
| Directive factory with dependency injection                                                                                                                                                                                          | Directive name as a variable                                                                                                                                               |
| <pre>angular.module('App', []) .directive('header', ['\$interval', function(\$interval) {   return {     templateUrl: 'header.html',     data: {       data: '='     }   } }]);</pre>                                                | <pre>angular.module('App', []) .directive(directiveName, function() {   return {     templateUrl: 'header.html',     data: {       data: '='     }   } });</pre>           |
| Directive factory returned from another function                                                                                                                                                                                     |                                                                                                                                                                            |
| <pre>angular.module('App', []) .directive('header', factoryFunction());  function factoryFunction() {   return function() {     return {       templateUrl: 'header.html',       data: {         data: '='       }     }   } }</pre> |                                                                                                                                                                            |

Table 9: Code samples to define an angular using different syntaxes

- **Directive factory as an arrow function:** Instead of passing a regular function it's possible to pass an arrow function as a directive factory. This changes the second argument's node type "ArrowFunctionExpression"
- **Directive factory as an identifier:** It's possible to define a function and pass the function name as the directive factory. This changes the second argument's node type to "Identifier".
- **Directive factory with dependency injection:** Similar to controller factories, it's possible to inject the dependencies to directive factory as well. That is done by passing an array as the second argument with the injected dependencies. This changes the second argument's node type to "ArrayExpression".
- **Directive name as a variable:** Instead of defining the directive name as a string, it can be passed as a variable. This makes the first argument's node type equal to "Identifier".
- **Directive factory returned from another function:** As shown in the example, it's possible to make a function call which returns an anonymous function with the directive object. This makes the second argument's node type equal to "CallExpression".

Those were the common ways of defining directives in an angular application. Then let's look at the angular component definition. Components are a special type of a directive where it promotes practices in component based application architecture. According to the code sample given in Table 8, following are the additional conditions applicable to detect a component.

- The callee property name will be equal to 'component'
- The second argument of the component definition will be an object with properties so the node type will be equivalent to "ObjectExpression";

By combining all these conditions, the final logic to identify a directive was defined as follows.

- Detect all the nodes with type "CallExpression" in the code

- Check whether the node contains two arguments
- Check where the first argument is a 'Literal' type or an 'Identifier' type
- Check whether the second argument is a 'FunctionExpression', 'ArrowFunctionExpression', 'ArrayExpression', 'ObjectExpression', 'Identifier' or a 'CallExpression'
- Check whether the callee property name is equal to 'directive' or 'component'

Once a directive is detected, the next step is to check whether the related JS code and the HTML code is placed together. There are two ways a template can be defined in a directive. A template can be defined as an inline template within the directive code using "template" parameter or it can be defined as an external resource using "templateUrl" parameter. Almost all the guidelines recommend moving the template as a separate HTML file and import using templateUrl parameter but there can be exceptional cases with inline templates. So under this rule we have to consider both. So we first check whether an external template is defined with the directive. If an external template is defined, then we get the directive file name and convert it to a possible template name by removing and adding the extension and the other recommended suffixes suggested by the popular style guides. Here we mainly referred to Johnpapa's[40] style guide to derive the possible template names. Then a file search is conducted to see whether a file with the derived template name exists. If it doesn't exist, the rule will fail and notify the user to rename the template with the suggested name so the directive and the template will place next to each other in the code base. Following is the pseudo code for the implemented rule.

```

FUNCTION isDirectiveCodePlacedTogether(file_name, ast_node) {
  IF
    ast_node type EQUALS 'CallExpression'
    AND ast_node number of arguments EQUALS 2
    AND ast_node first argument type EQUALS Literal OR 'Identifier'
    AND ast_node second argument type EQUALS
      'FunctionExpression' OR 'ArrowFunctionExpression' OR
      'ArrayExpression'
      OR 'ObjectExpression' OR 'Identifier' OR 'CallExpression'

```

```

    AND ast_node callee property EQUALS 'directive' OR 'component'
  THEN
    template_file_name <= remove extension and suffixes in file_name
    IF fileExists(template_file_name) THEN
      RETURN TRUE;
    ELSE
      RETURN FALSE;
  ELSE
    RETURN TRUE;
}

```

*Table 10: Pseudo code for rule - Place all the code of a directive together*

#### **4.2.2. Refactoring #2: Favor directives as elements over attributes, comment and classes**

Under this refactoring, we have to detect whether defined directives are defined as elements but not as attributes, comments or classes. Even though directives can be restricted as attributes, comments or classes, the component directives can only be defined as an element. Because of that we don't need to worry about components here. The 4 types of restricted options available in directives are represented as follows.

- 'A' - restrict to attribute name
- 'E' - restrict to element name
- 'C' - restrict to class name
- 'M' - restrict to comment

These representations can be used combinedly in a directive where if we need to restrict an a directive as an element and an attribute 'EA' can be specified as the restrict type. If any restric type is not defined in a directive, the default value will be 'AE' which is restricted to attributes and elements.

Following sample code explains how a directive can be restricted to element and any other type.

| Directive restricted to element | Directive which can be used in all 4 types |
|---------------------------------|--------------------------------------------|
| angular.module('App', [])       | angular.module('App', [])                  |

|                                                                                                                                                        |                                                                                                                                                           |
|--------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> .directive('header', function() {   return {     templateUrl: 'header.html',     restrict: 'E',     data: {       data: '='     }   } }); </pre> | <pre> .directive('header', function() {   return {     templateUrl: 'header.html',     restrict: 'EACM',     data: {       data: '='     }   } }); </pre> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|

*Table 11: Code samples to define an angular directive restrict*

Following Pseudo code explains the algorithm to detect whether the directive is restricted to an element or not.

```

FUNCTION isDirectiveRestrictedToElement(ast_node) {
  IF directive exists THEN
    restrict <= extract the restrict option
    IF restrict NOT_EQUALS NULL
      AND restrict EQUALS 'E' THEN
        RETURN TRUE;
    ELSE
      RETURN FALSE;
  ELSE
    RETURN TRUE;
}

```

*Table 12: Pseudo code for rule - Favor directives as elements over attributes, comment and classes*

The logic is to first check whether a directive is defined in a given source file and if so check whether the restrict attribute is set for the directive. If the restrict attribute is set then check whether it's equal to 'E' and if it equal the rule passes. If the value is not defined or the value is different than 'E' the rule will fail.

When implementing the rule I encountered that there is a similar rule implemented in `eslint-plugin-angular`[41] which is the Angular plugin for ESLint. That plugin is developed to provide guidelines to write better AngularJS code according to Johnpapa's style guide. That rule allowed us to provide the restrict option we want to validate and then check it against the given source code. The rule followed the same logic so without implementing it by myself I integrated the `eslint-plugin-angular`

### 4.2.3. Refactoring #3: Eliminate transclusion

Under this refactoring, we have to detect whether the ‘ng-transclude’ or ‘ng-transclude-slot’ directives are used in the AngularJS templates. ‘ng-transclude’ directive can be used in an element, attribute or a class and ‘ng-transclude-slot’ directive can be used as an attribute so we have to detect all three types of usages of the ‘ng-transclude’. Following code samples show how the ‘ng-transclude’ is used in the code.

|                 |                                                           |
|-----------------|-----------------------------------------------------------|
| As an element   | <ng-transclude ng-transclude-slot="body"></ng-transclude> |
| As an attribute | <div ng-transclude="body"></div>                          |
| As a class      | <div class="ng-transclude: body"></div>                   |

*Table 13: Code samples on how to use ng-transclude directive*

Since ng-transclude directive is used inside an AngularJS template, the rule needs to be implemented in the HTMLHint code scanner. Following Pseudo code explains the algorithm to detect whether the ng-transclude directive is used in the templates.

```
FUNCTION isNgTranscludeNotUsed(event) {
  IF event EQUALS 'tagstart' THEN
    tagname <= extract tag name
    IF tagname EQUALS 'ng-transclude' THEN
      RETURN FALSE;
    ELSE
      attributes <= extract attributes of the element
      FOR EACH attribute in attributes
        attribute_name <= get attribute name
        IF attribute_name EQUALS 'ng-transclude' OR 'ng-transclude-slot'
      THEN
        RETURN FALSE;
      ELSE IF attribute_name EQUALS 'class' THEN
        class_text <= extract the class text
        IF class_text INCLUDES 'ng-transclude' THEN
          RETURN FALSE
      ELSE
        RETURN TRUE;
}
```

*Table 14: Pseudo code for rule - Eliminate transclusion*

The algorithm and its implementation is straightforward where it checks every HTML element in a document, validates whether the element name, an attribute or a class matches with ‘ng-transclude’ keyword. Additionally, it checks whether an attribute matches with ‘ng-transclude-slot’ keyword.

#### 4.2.4. Refactoring #4: Replace ng-include with a directive

Under this refactoring, we have to detect whether the ‘ng-include’ directive is used in any of the AngularJS templates. It is similar to the ‘ng-transclude’ directive usage detection where ‘ng-include’ directive also can be used as an element, attribute or a class. Following code samples show how the ‘ng-include’ is used in the code.

|                 |                                                                        |
|-----------------|------------------------------------------------------------------------|
| As an element   | <code>&lt;ng-include src="template.html"&gt;&lt;/ng-include&gt;</code> |
| As an attribute | <code>&lt;div ng-include="template.html"&gt;&lt;/div&gt;</code>        |
| As a class      | <code>&lt;div class="ng-include:template.html"&gt;&lt;/div&gt;</code>  |

*Table 15: Code samples on how to use ng-include directive*

The algorithm and the implementation is the same as the “Eliminate transclusion” section so it’s not repeated here.

#### 4.2.5. Refactoring #6: Extract controller as a directive

‘ng-controller’ is a directive which is used to bind a controller to a view. This directive can be used as an element and an attribute. Expect for not having the support as a comment, rest of the code samples and detection algorithm is very similar to Refactoring #3 and #4. Due to that, the implementation of the rule followed the same approach as Refactoring #3 and #4 so it’s not repeated under this section.

#### 4.2.6. Refactoring #7: Extract directive from large templates

According to Johnpapa's AngularJS style guide, the max number of lines per a file should be less than 400 lines[40]. AngularJS recommends to have the directive code and the template separately but React recommends to have the component logic and the JSX code in a single file. The approach for migration is to combine both directive

code and the template into a single React component. This can lead to components which are higher up to 800 lines of code if both the directive and the template contains 400 lines in each and such React components will be extremely harder to use as well as difficult to maintain. React community doesn't recommend any max number of lines for its components and recommends to have a limit based on each team's need. By considering all these facts, I have decided to reuse the recommendation of the AngularJS style guide for the migrated react components. The 400 lines of max limit was enforced as 200 per directive code and 200 per template code. Any directive or template having more than 200 lines can result in larger React components so it's better to break such directives and templates into separate directives before converting to React.

The rule needs to be applied in both JS and HTML so the rule was added to both ESLint and HTMLHint. HTMLHint provides an event called 'end' which gets triggered when a file scan reaches its end so the rule was written by tapping to it. Following is the pseudo code for the rule implemented for templates.

```
FUNCTION isTemplateNotLarge(event) {
  IF event EQUALS 'end' THEN
    line_number <= extract line of the end line of the file
    IF line_number LESS THAN 200 THEN
      RETURN TRUE;
    ELSE
      RETURN FALSE;
  ELSE
    RETURN TRUE;
}
```

*Table 16: Pseudo code for HTMLHint rule - Extract directive from large templates*

When the AST is generated from ESLint, it provides a 'Program' node to represent the whole program in the file. When walking through the AST, the end of the program node can be listened to at the 'Program:exit' event so the rule was implemented as a listener for that event. The logic in both template and directive rules are similar except for the check on directive exists condition in the directive rule.

#### 4.2.7. Refactoring #12: Replace two-way data binding with one-way data binding

The bindings were defined in AngularJS directives to restrict what to be accepted as inputs and outputs of the directive. There are 4 types to define the scope bindings in AngularJS and based on the binding type how the binded data is going to behave within the directive.

- '@' - bind dom property value which is a strings from parent scope to directive scope
- '=' - binds scope values allowing bidirectional data flow which is called two-way data binding. Instead of a string value, an evaluated expression is passed via this binding.
- '<' - binds the scope value in a single direction where the data flow happens from parent to child scope but not the other way. This is called one-way data binding. An evaluated expression is passed via this binding as well.
- '&' - binds expressions from the parent scope which can be executed in the directive with the context of parent scope.

Under this refactoring we have to detect whether the defined scope bindings of directives avoid scope bindings with '=' and favour '@', '<' and '&'. The AngularJS components which are a special type of a directive doesn't support '=' type of bindings so we don't need to consider it as part of the scan. As discussed in the implementation of the first rule which is for "Refactoring #1: Place all the code of a directive together" it's possible to define the directive factory function in multiple styles. Due to that, all these styles should be considered when extracting the directive definition object and the bindings defined inside it. The code samples presented in Table 8 and Table 9 are valid for this refactoring as well. Out of the presented code samples in Table 8 and Table 9, following are the relevant styles for this and let's look into each style and see how to extract the scope bindings.

- Directive factory as a function
- Directive factory as an arrow function
- Directive factory as an identifier
- Directive factory returned from another function

- Directive factory with dependency injection

Even though there are differences in the binding extraction algorithm based on the used coding style, the abstracted algorithm can be defined as the following pseudocode.

```

FUNCTION isTwoWayDataBindingNotUsed(ast_node) {
  IF directive exists THEN
    directive_factory_type <= detect the type of the second argument type in
    ast_node
    factory_options <= extractFactoryFunctionObject(ast_node,
    directive_factory_type)
    scope_bindings = extractScopeBindings(factory_options)
    FOR EACH binding in scope_bindings
      IF first character of the binding EQUALS '=' THEN
        RETURN FALSE
    RETURN TRUE
  ELSE
    RETURN TRUE
}

FUNCTION extractFactoryFunctionObject(ast_node, directive_factory_type) {
  // to be defined
}

FUNCTION extractScopeBindings(factory_options) {
  bindings <= extract binding object from the factory options
  RETURN object values of bindings;
}

```

*Table 17: Pseudo code of the isTwoWayDataBindingNotUsed method*

The logic in the “extractFactoryFunction” depends on the coding styles mentioned before so let’s analyse each of the styles separately to define the function extraction algorithm for that function.

#### **4.2.7.1. Directive factory as a callback function or an arrow function**

In this style, a callback function is directly passed to the second argument of the detective defining function. So the directive factory type becomes ‘FunctionExpression’. The function body is enclosed with a node type equal to ‘BlockStatement’ and inside its body the ‘ReturnStatement’ can be found. If the argument of the ‘ReturnStatement’ is equivalent to ‘ObjectExpression’ that can be

returned as the factory options. When an arrow function passes the AST is almost similar except for the second argument to become 'ArrowFunctionExpression'. Figure 6 shows how the AST looks in when the factory is a callback function.

```

- arguments: [
  + Literal {type, value, raw}
  - FunctionExpression {
    type: "FunctionExpression" Factory function expression
    expression: false
    generator: false
    async: false
    params: [ ]
    - body: BlockStatement { Block of the function body
      type: "BlockStatement"
      - body: [
        - ReturnStatement = $node {
          type: "ReturnStatement"
          + argument: ObjectExpression {type, Returned
            properties} Object
        }
      ]
    }
  ]
]

```

Figure 14: AST for detecting a directive factory defined as a function

According to the generated AST, the algorithm of the extractFactoryFunctionObject will look like follows.

```

FUNCTION extractFactoryFunctionObject(ast_node, directive_factory_type) {
  SWITCH directive_factory_type EQUALS
    CASE FunctionExpression AND 'ArrowFunctionExpression' THEN
      RETURN extractReturnObjectFromFunction(ast_node)
    DEFAULT
      RETURN NULL
  ELSE
    RETURN NULL;
}

FUNCTION extractReturnObjectFromFunction(ast_node) {
  return_statement <= extract the return statement of the function
  IF return_statement argument type EQUALS 'ObjectExpression' THEN
    return_statement_argument <= extract the return statement argument
    RETURN return_statement_argument
  ELSE

```

|                            |
|----------------------------|
| <pre> RETURN NULL } </pre> |
|----------------------------|

*Table 18: Pseudo code of the `extractFactoryFunctionObject` method when directive factory defined as a function*

#### **4.2.7.2. Directive factory as an identifier**

In this style, the factory function is defined separately and its name is passed as the second argument. Due to this the second argument node type becomes an 'Identifier'. Not like in previous cases, the function declaration is not available under the given node so extracting the function becomes more complex than before. A function declaration can be identified by node type 'FunctionDeclaration' in the AST so the approach is to go through all the nodes which has the node type as 'FunctionDeclaration' and check whether name of is matching with the identifier passed to the directive factory registration. The name of the function can be extracted by the name of the id attribute in the node. If the name matches, the factory options should be extracted from that function as before. Figure 7 shows how the AST looks in this case.

```

- expression: CallExpression {
  type: "CallExpression"
  + callee: MemberExpression {type, object, property, computed}
  - arguments: [
    + Literal {type, value, raw}
    - Identifier = $node {
      type: "Identifier"
      name: "header" Function name identifier
    }
  ]
}
}
}
- FunctionDeclaration {
  type: "FunctionDeclaration"
  - id: Identifier {
    type: "Identifier"
    name: "header" Matching function name
  }
  expression: false
  generator: false
  async: false
  params: [ ]
  - body: BlockStatement {
    type: "BlockStatement"
    - body: [
      - ReturnStatement {
        type: "ReturnStatement"
        + argument: ObjectExpression {type, properties} Returned object
      }
    ]
  }
}

```

*Figure 15: AST for detecting a directive factory defined as an identifier*

Following is the extended pseudocode of `extractFactoryFunctionObject` function to support the new logic.

```

FUNCTION extractFactoryFunctionObject(ast_node, directive_factory_type) {
  SWITCH directive_factory_type EQUALS
    CASE FunctionExpression AND ArrowFunctionExpression THEN
      RETURN RETURN extractReturnObjectFromFunction(ast_node)
    CASE 'Identifier' THEN
      function_name <= extract the identifier name
      RETURN extractFactoryFunction(function_name)
    DEFAULT
      RETURN NULL;
  ELSE
    RETURN NULL;
}

FUNCTION extractFactoryFunction(node_identifier_name) {
  full_ast <= get full abstract syntax tree of the document
  FOR EACH node in full_ast
    IF node type EQUALS 'FunctionDeclaration' THEN
      node_identifier_name <= extract the node identifier name
      IF function_name EQUALS node_identifier_name THEN
        RETURN extractReturnObjectFromFunction(node)
}

```

*Table 19: Pseudo code of the extractFactoryFunctionObject method when directive factory defined as an identifier*

#### 4.2.7.3. Directive factory returned from another function

In this style, the factory function is returned as a result of another function execution. This is not a very common pattern. Here the node type of the second argument passed to the directive definition becomes a 'CallExpression'. If a call expression is detected, then the name of the function can be extracted from the callee name. Similar to the previous case, the matching 'FunctionDeclaration' needs to be fetched by traversing through the AST. This function doesn't directly return the factory object, instead it returns another function with the type of 'FunctionExpression'. Once that inner function is extracted the return statement of it gives the factory defining object. Figure 8 shows the AST for this case.

```

- arguments: [
  + Literal {type, value, raw}
  - CallExpression {
    type: "CallExpression"
    - callee: Identifier {
      type: "Identifier"
      name: "header" Call expression name
    }
    arguments: [ ]
  }
]
}
}
}
- FunctionDeclaration {
  type: "FunctionDeclaration"
  - id: Identifier {
    type: "Identifier"
    name: "header" Matching function name
  }
  expression: false
  generator: false
  async: false
  params: [ ]
  - body: BlockStatement {
    type: "BlockStatement"
    - body: [
      - ReturnStatement {
        type: "ReturnStatement"
        - argument: FunctionExpression { Returned function
          type: "FunctionExpression"
          expression: false
          generator: false
          async: false
          + params: [1 element]
          - body: BlockStatement {
            type: "BlockStatement"
            - body: [
              + ReturnStatement {type, argument} Returned object
            ]
          }
        }
      }
    ]
  }
}

```

Figure 16: AST for detecting a directive factory defined as function returned from a function expression

The improved logic to cater the new function detection is presented in the below pseudocode.

```

FUNCTION extractFactoryFunctionObject(ast_node, directive_factory_type) {
  SWITCH directive_factory_type EQUALS
    CASE FunctionExpression AND ArrowFunctionExpression THEN
      RETURN RETURN extractReturnObjectFromFunction(ast_node)
    CASE 'Identifier' THEN
      function_name <= extract the identifier name
      RETURN extractFactoryFunction(function_name)
    CASE 'CallExpression' THEN
      function_name <= extract the callee identifier name
      RETURN extractFactoryFunction(function_name)
    DEFAULT
      RETURN NULL;
  ELSE
    RETURN NULL;
}

FUNCTION extractReturnObjectFromFunction(ast_node) {
  return_statement <= extract the return statement of the function
  IF return_statement argument type EQUALS 'ObjectExpression' THEN
    return_statement_argument <= extract the return statement argument
    RETURN return_statement_argument
  ELSE IF return_statement argument type EQUALS 'FunctionExpression'
  THEN
    function_node <= extract the return statement argument
    extractReturnObjectFromFunction(return_statement_argument)
  ELSE
    RETURN NULL
}

```

*Table 20: Pseudo code of the extractFactoryFunctionObject method when directive factory returned from another function*

#### 4.2.7.4. Directive factory with dependency injection

When directive factory is passed with dependency injection, the second argument of the directive definition becomes an array. The last element of the array defines the factory and all the other elements become the injected dependencies. The last element of the array can be in any form of the above mentioned styles such as a callback function, arrow function, identifier or a function expression. So the already written extractFactoryFunctionObject function algorithm can be improved to check whether the second argument type is equal to 'ArrayExpression' and if so the last element of the array should be passed to the existing object extraction algorithm. Following is the improved algorithm for this case.

```

FUNCTION extractFactoryFunctionObject(ast_node, directive_factory_type) {
  SWITCH directive_factory_type EQUALS
    CASE FunctionExpression AND ArrowFunctionExpression THEN
      RETURN RETURN extractReturnObjectFromFunction(ast_node)
    CASE 'Identifier' THEN
      function_name <= extract the identifier name
      RETURN extractFactoryFunction(function_name)
    CASE 'CallExpression' THEN
      function_name <= extract the callee identifier name
      RETURN extractFactoryFunction(function_name)
    CASE 'ArrayExpression' THEN
      factory_element_node <= fetch the last element of the array
      factory_type <= detect the type of the second argument type in ast_node
      RETURN extractFactoryFunctionObject(factory_element_node,
factory_type)
    DEFAULT
      RETURN NULL;
    ELSE
      RETURN NULL;
}

```

*Table 21: Pseudo code of the extractFactoryFunctionObject method when directive function defined with dependency injection*

By consolidating all these algorithm pieces, following pseudo code presents the full algorithm to detect two way data bindings.

```

FUNCTION isTwoWayDataBindingNotUsed(ast_node) {
  IF directive exists THEN
    directive_factory_type <= detect the type of the second argument type in
ast_node
    factory_options <= extractFactoryFunctionObject(ast_node,
directive_factory_type)
    scope_bindings = extractScopeBindings(factory_options)
    FOR EACH binding in scope_bindings
      IF first character of the binding EQUALS '=' THEN
        RETURN FALSE
    RETURN TRUE
  ELSE
    RETURN TRUE
}

FUNCTION extractFactoryFunctionObject(ast_node, directive_factory_type) {
  SWITCH directive_factory_type EQUALS
    CASE FunctionExpression AND ArrowFunctionExpression THEN

```

```

    RETURN RETURN extractReturnObjectFromFunction(ast_node)
CASE 'Identifier' THEN
    function_name <= extract the identifier name
    RETURN extractFactoryFunction(function_name)
CASE 'CallExpression' THEN
    function_name <= extract the callee identifier name
    RETURN extractFactoryFunction(function_name)
DEFAULT
    RETURN NULL;
ELSE
    RETURN NULL;
}

FUNCTION extractReturnObjectFromFunction(ast_node) {
    return_statement <= extract the return statement of the function
    IF return_statement argument type EQUALS 'ObjectExpression' THEN
        return_statement_argument <= extract the return statement argument
        RETURN return_statement_argument
    ELSE IF return_statement argument type EQUALS 'FunctionExpression' THEN
        function_node <= extract the return statement argument
        extractReturnObjectFromFunction(return_statement_argument)
    ELSE
        RETURN NULL
}

FUNCTION extractScopeBindings(factory_options) {
    bindings <= extract binding object from the factory options
    RETURN object values of bindings;
}

```

*Table 22: Consolidated pseudo code of the isTwoWayDataBindingNotUsed method*

#### **4.2.8. Refactoring #14: Replace the AngularJS specific features used in the converted component**

By the time we come to this refactoring, the Angular directive is partially converted to a React component where the directive template is converted to the component JSX. When converting the directive template into JSX, the standard html elements and attributes are covered but still AngularJS specific inbuilt directives, custom directives and scope references will be used in the component JSX.

Under this refactoring, the most commonly used AngularJS inbuilt directives inside the converted component code are detected and indicated to the developer with a

suggestion on how to migrate so he can start replacing the occurrences one by one. Out of the AngularJS features mentioned in the Table 2, following directives were selected to detect automatically.

| Directive                  | Usage                         | Suggestion                                                                                                                            |
|----------------------------|-------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| ng-if                      | Attribute                     | Replace ng-if with native conditions. Ex: <code>this.props.showMe ? &lt;button /&gt;</code>                                           |
| ng-show, ng-hide           | Element, Attribute            | Replace ng-show with native conditions. Ex: <code>this.props.showMe ? &lt;button /&gt;</code>                                         |
| ng-repeat                  | Attribute                     | Replace ng-repeat with array iterators. Ex: <code>this.props.data.map(function(item) { return &lt;div&gt;{item}&lt;/div&gt; })</code> |
| ng-class                   | Attribute, CSS class          | Replace ng-class with native conditions. Ex: <code>className={this.props.isActive? "is-active" : "inactive"}</code>                   |
| ng-maxlength, ng-minlength | Attribute                     | Replace ng-maxlength with native HTML validations. Ex: <code>&lt;input maxLength="10"/&gt;</code>                                     |
| ng-disabled                | Attribute                     | Replace ng-disabled with native HTML functionality. Ex: <code>&lt;button disabled={condition} /&gt;</code>                            |
| ng-required                | Attribute                     | Replace ng-required with native HTML5 validations. Ex: <code>&lt;input required={condition} /&gt;</code>                              |
| ng-init                    | Attribute, CSS class          | Replace ng-init with component lifecycle methods. Ex: <code>componentDidMount()</code> or <code>componentWillUnmount()</code>         |
| ng-form                    | Element, Attribute, CSS class | Refactor ng-form usage with HTML native behaviors                                                                                     |
| ng-model                   | Attribute                     | Refactor ng-model usage with component scope                                                                                          |
| ng-click                   | Attribute                     | Replace ng-click usage with <code>onClick</code> behavior                                                                             |
| ng-src                     | Attribute                     | Replace ng-src usage with native HTML <code>src</code>                                                                                |

|          |                                 |                                                                                   |
|----------|---------------------------------|-----------------------------------------------------------------------------------|
|          |                                 | behavior                                                                          |
| ng-<any> | Element,Attribute,<br>CSS class | Refactor the inbuilt angular directive usage<br>with appropriate native behaviors |

*Table 23: List of AngularJS inbuilt directives selected to detect inside the JSX*

Apart from detecting the most commonly used AngularJS directives, a wildcard matching will be done to detect any inbuilt directives by matching the prefix ‘ng-’. When detecting these directive usages, we have to be careful not to highlight them when used inside AngularJS code but only highlight when those are used inside JSX code. In order to achieve this, the AST for the same HTML in an AngularJS template and a JSX code was analysed.

|                     |                                                                                                                                                                                                                                                                                                                                                                  |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>JSX<br/>code</b> | <pre> class PhoneDetailComponent extends Component {   render() {     return (       &lt;div&gt;         &lt;div className="phone-images"&gt;           &lt;img ng-src="{{img}}" className="phone" ng-class="{selected: img === \$ctrl.mainImageUrl}" ng-repeat="img in \$ctrl.phone.images" /&gt;         &lt;/div&gt;        &lt;/div&gt;      );   } } </pre> |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| AST | <pre> - body: [   - ReturnStatement {     type: "ReturnStatement"     Return statement   - argument: JSXElement {     type: "JSXElement"     + openingElement: JSXOpeningElement {type, attributes, name, selfClosing}     + closingElement: JSXClosingElement {type, name}     - children: [       + JSXText {type, value, raw}       - JSXElement {         type: "JSXElement"         + openingElement: JSXOpeningElement {type, attributes, name, selfClosing}         + closingElement: JSXClosingElement {type, name}         - children: [           + JSXText {type, value, raw}           - JSXElement {             type: "JSXElement"             - openingElement: JSXOpeningElement {               type: "JSXOpeningElement"               - attributes: [                 + JSXAttribute {type, name, value}                 + JSXAttribute {type, name, value}                 - JSXAttribute {                   type: "JSXAttribute"                   - name: JSXIdentifier = \$node {                     type: "JSXIdentifier"                     name: "ng-class"                     JSX attribute name                   }                 }               ]             }             JSX element           }         ]       }     ]   } ] </pre> |
|-----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Table 24: AST generated for AngularJS directive usage detection in JSX

According to the AST, the elements and attributes in JSX can be identified with ‘JSXElement’ and ‘JSXAttribute’ node types. This will avoid the algorithm to detect any similar directives used in non JSX code. Since the detected directives can be used as elements, attributes or classes, the algorithm should traverse through the AST to identify matching directives names in elements, attributes and comments. Following is the pseudo code of the algorithm used to detect the AngularJS directive used in JSX.

```

FUNCTION isDirectivesNotInJSX(ast_node) {
  IF ast_node type EQUALS ‘JSXElement’ THEN
    element_name <= extract the element name

```

```

    RETURN notDetectAnAngularDirective(element_name)
ELSE IF ast_node type EQUALS 'JSXAttribute' THEN
    attribute_name <= extract the element name
    IF (attribute_name EQUALS 'ng-class' OR 'class' OR 'className') THEN
        class_string <= extract the class string
        RETURN notDetectAnAngularDirective(class_string)
    ELSE
        RETURN notDetectAnAngularDirective(attribute_name)
ELSE
    RETURN TRUE;
}

FUNCTION notDetectAnAngularDirective(string) {
    IF string EQUALS ONE OF directive name array
        RETURN FALSE
    ELSE IF string CONTAINS ON OF directive name array
        RETURN FALSE
    ELSE IF string CONTAINS 'ng-'
        RETURN FALSE
    ELSE
        RETURN TRUE
}

```

*Table 25: Pseudo code for algorithm to detect AngularJS directive usage in JSX*

#### 4.2.9. Refactoring #15: Migrate the directive bindings as component props

When converting the directive template to JSX, the scope variables referred to in the template HTML should be replaced with the props or state defined in the component. This is not automatically happening in the conversion tool so it needed to be detected using custom algorithms. First we have to identify all the scope references in JSX. Since the scope references are referred to as variables, the node type of them are represented as “Identifier” in AST. Following are the ways variables can be referred to in JSX and their respective ASTs.

| Identifiers which should be refactored |                          |
|----------------------------------------|--------------------------|
| <dt>{name}</dt>                        | <dt>{employee.name}</dt> |

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> - JSXExpressionContainer = \$node {   type: "JSXExpressionContainer"   - expression: Identifier {     type: "Identifier"     name: "name"   } } </pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | <pre> - JSXExpressionContainer {   type: "JSXExpressionContainer"   - expression: MemberExpression {     type: "MemberExpression"     - object: Identifier = \$node {       type: "Identifier"       name: "employee"      When referring                              an object     }     - property: Identifier {       type: "Identifier"       name: "name"          Object property     }     computed: false   } } </pre>                                                                                                                                                                                                                                     |
| <p align="center"><b>Identifiers which should not be refactored</b></p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <p>&lt;dt&gt;{this.props..name}&lt;/dt&gt;</p> <pre> - JSXExpressionContainer {   type: "JSXExpressionContainer"   - expression: MemberExpression {     type: "MemberExpression"     - object: MemberExpression {       type: "MemberExpression"       - object: ThisExpression {         type: "ThisExpression"      This                                    keyword       }       - property: Identifier {         type: "Identifier"         name: "props"               props                                    identifier       }       computed: false     }     - property: Identifier {       type: "Identifier"            prop name       name: "name"     }   } } </pre> | <p>&lt;dt&gt;{this.props..name}&lt;/dt&gt;</p> <pre> - JSXExpressionContainer {   type: "JSXExpressionContainer"   - expression: MemberExpression {     type: "MemberExpression"     - object: MemberExpression {       type: "MemberExpression"       - object: ThisExpression {      This         type: "ThisExpression"        kayword       }       - property: Identifier {         type: "Identifier"         name: "state"                 state                                    identifier       }       computed: false     }     - property: Identifier {       type: "Identifier"       name: "name"                    state name     }   } } </pre> |
| <pre> &lt;dl&gt;   {     this.props.names.map((name, i) =&gt; {       return (&lt;dt&gt;{name}&lt;/dt&gt;)     })   } &lt;/dl&gt; </pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |

```

JSXExpressionContainer {
  type: "JSXExpressionContainer"
  - expression: CallExpression {
    type: "CallExpression"
    + callee: MemberExpression {type, object, property, computed}
    - arguments: [
      - ArrowFunctionExpression {
        type: "ArrowFunctionExpression"      Arrow function
        expression: false
        generator: false
        async: false
      + params: [2 elements]
      - body: BlockStatement {
        type: "BlockStatement"
        - body: [
          - ReturnStatement {
            type: "ReturnStatement"          Return statement
            - argument: JSXElement {
              type: "JSXElement"
              + openingElement: JSXOpeningElement {type,
                attributes, name, selfClosing}
              + closingElement: JSXClosingElement {type, name}
              - children: [
                - JSXExpressionContainer = $node {
                  type: "JSXExpressionContainer"
                  - expression: Identifier {
                    type: "Identifier"      Identifier
                    name: "name"
                  }
                }
              ]
            }
          }
        ]
      }
    ]
  }
}

```

Table 26: AST generated for variable references used in JSX

By analyzing the above code samples following conditions were derived to use in the detection algorithm.

- The variables referred to in the code can be identified by the node type 'Identifier'. When an identifier is detected, the algorithm will traverse up in the AST and check whether the identifier is referred inside a 'JSXExpressionContainer' and if not we ignore such identifier.

- The parent of the filtered identifiers again checked to see if there any 'FunctionExpression' or an "ArrowFunctionExpression" can be found. If found, the parameters passed to the function will be cross checked with the identifier name since those are legitimate variable references and no need to convert as props or states.
- If an identifier is an object reference, the identifier's parent node type will be 'MemberExpression'. If the member expression has multiple levels such as "employee.personalDetails.name" the first level can be determined by checking whether it has a property called 'object' and its type is also an 'Identifier'. Only such member expressions will be filtered and extract the identifier from it and then repeat the checks mentioned in the previous two steps to detect the identifiers needed refactoring.
- The member expressions which are already converted will have 'this.props' or 'this.state.' prefixes. Such member expressions will not have a property called 'object' and object type will match 'ThisExpression' instead of 'Identifier'. Due to that, such member expressions will be ignored in the previous step so they'll not be highlighted as identifiers which need refactoring.

Following is the pseudo code for the above explained algorithm.

```

FUNCTION isNoAngularScopeReferencesDetectedInJSX(ast_node) {
  IF (ast_node type EQUALS 'Identifier'
    AND ast_node parent type NOT EQUALS 'MemberExpression')
  OR
    (ast_node type EQUALS 'MemberExpression'
    AND ast_node object type EQUALS 'Identifier') THEN
    identifier_name <= extract identifier name
    parent_node_tree <= extract the parent nodes of the given identifier

    isDefinedInsideJSXExpression <=
isDefinedInsideJSXExpression(parent_node_tree)
    isIdentifierDefinedAsAFunctionParameter
      <= isIdentifierDefinedAsAFunctionParameter(parent_node_tree,
identifier_name)

    IF isDefinedInsideJSXExpression EQUALS TRUE

```

```

        AND isIdentifierDefinedAsAFunctionParameter EQUALS FALSE THEN
        RETURN FALSE
    ELSE
        RETURN TRUE;
}

FUNCTION isDefinedInsideAJSXExpression(parent_node_tree) {
    FOR EACH parent_node IN parent_node_tree
        IF parent_node type Equals 'JSXExpressionContainer' THEN
            RETURN TRUE
    RETURN FALSE
}

FUNCTION isIdentifierDefinedAsAFunctionParameter(parent_node_tree,
identifier_name) {
    FOR EACH parent_node IN parent_node_tree
        IF parent_node type Equals 'FunctionExpression'
            OR 'ArrowFunctionExpression' THEN
            parameters <= extract parameters
            IF (identifier_name IS IN parameters) THEN
                RETURN TRUE
    RETURN FALSE
}

```

*Table 27: Pseudo code for algorithm to detect scope references in in JSX*

With that, algorithms which are defined to detect the refactoring needed during the AngularJS to React are completed. Next we will look into the two automated code generations implemented under this research project.

#### **4.2.10. Refactoring #13: Convert the directive template as JSX in react component**

Migrating the directive template into a React component is a time consuming task. When looking into this migration in a more granular level, the initial conversion of the pure HTML to JSX step was identified as a possible step to automation. The HTMLtoJSX[38] was an already existing JS based tool which provided the support to convert an html to JSX. This tool was integrated with the consolidated migration tool and provided the ability to specify an AngularJS directive template path as the input so the tool will read the content of the template and convert it to JSX wrapped inside a React component. The generated React component name and the file name was derived by formatting the template file name. As an example when migrating a

template with the name “phone-detail.template.html”, the component was named as “PhoneDetailComponent” and the component will be saved to a file called “PhoneDetailComponent.js”. It was decided to have the “Component” suffix for all the automatically converted React components to avoid them conflict with the existing AngularJS directives when embedding them into the AngularJS using React2Angular middleware.

This concludes the implementation of the AngularJS to React migration workflow automation. The main outcome of this implementation is the Ng-React Copilot tool which automates certain steps in the migration workflow. Following section will evaluate the results of Ng-React Copilot tool.

## 5. RESULTS AND EVALUATION

The effectiveness of the Ng-React Copilot tool can be evaluated by applying the tool in migrating real AngularJS based applications. It'll allow us to identify whether the tool gives better guidance for the migration process as well as it'll allow us to compare it with fully manual conversion proposes and identify what are the improvements needed for the tool to improve its effectiveness. In order to evaluate Ng-React Copilot tool we identified 3 different levels of benchmark applications based on their size and the scope. The categories are,

- Small applications which are having less complicated AngularJS code. Tutorial applications given in AngularJS official sites are selected as the benchmarking applications for this category.
- Medium level applications which are not starter applications but complete applications for server specific purpose. The AngularJS applications listed in Github[43] are selected as the benchmarking applications for this category.
- Enterprise level applications which are having larger and complex code bases are candidates for this category. OrangeHRM enterprise is an HR application which is built on AngularJS so it was selected as the benchmarking application for this category.

When evaluating Ng-React Copilot, following steps will be followed.

- The benchmark application will be installed locally and verify it's functionality.
- Install NG-React Copilot and scan the applications AngularJS code using it.
- Analyse the highlights and the summary of the scan results to get an idea about the scale of the migration project.
- Install necessary libraries to set up React for the application and include the compiled code in the application main HTML.
- Select a starting point for the migration based on the scanning results and suggested refactorings from Ng-React Copilot.
- Perform the refactorings suggested and convert each AngularJS directive to React components.

- Verify the related functionality of the application after converting each directive.
- Continue the process till we collect the required level of data about the migration for the evaluation.

Let's look at how this process is applied for each benchmark application.

### 5.1. Migrating small AngularJS applications having less complicated code

Under this category, I have selected the AngularJS PhoneCat[44] application which is given in the AngularJS official tutorials. This application is a sample application built to educate the developers on how to develop AngularJS applications. Since this is the sample application introduced in the official tutorials, it is the most referred AngularJS application codebase out there. By picking it as the first application to benchmark, we can assume it will cover some common patterns used in other AngularJS applications out there.

After setting up the PhoneCat application and the Ng-React Copilot application, the initial scanning was done to identify the scale of the migration. There were 22 scannable files in the code base but no refactoring was detected.

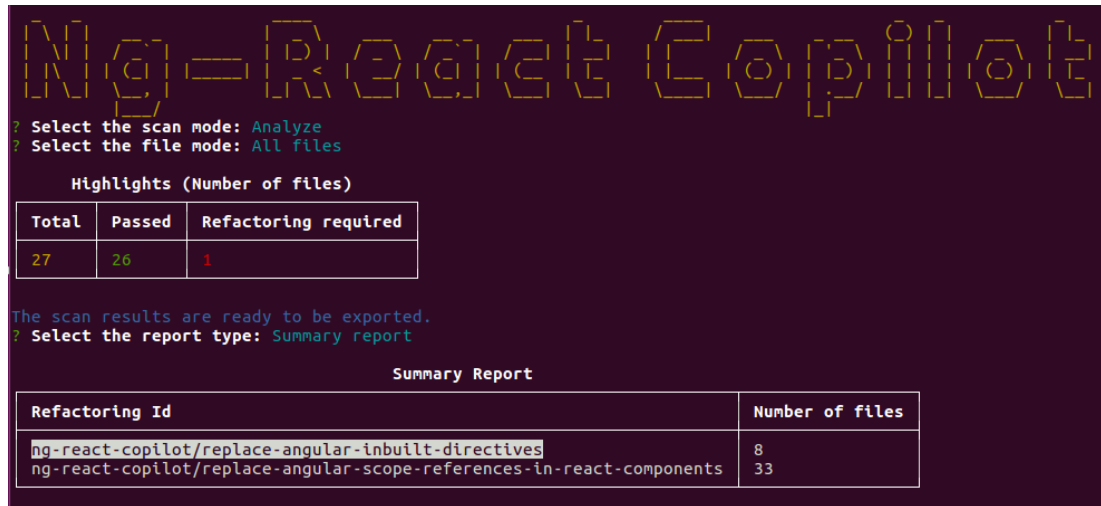


| Highlights (Number of files) |        |                      |
|------------------------------|--------|----------------------|
| Total                        | Passed | Refactoring required |
| 22                           | 22     | 0                    |

*Figure 17: Initial scan results for Angular-Phonecat application*

Then the next step was to pick the existing AngularJS directives and start migrating them one by one. There were two directives in the application which are phone-list-directive which renders the list of phones and phone-details-directive which renders individual phone details. The migration started with the phone-details directive. Ng-React Copilot code generation feature was used to generate PhoneDetailComponent by specifying the path of phone-details-template and the generated file was moved

into the react application folder. The component was registered within the index.js file of the react application and compiled the React application code as a single production file which can include in the main Angular-Phonecat application. After setting up the basic environment for the React component, the Ng-React Copilot tool was executed again to scan the code. Following are the scanning results of it.



*Figure 18: Scan results for Angular-Phonecat application after generating PhoneDetailsComponent*

|                                      |                                                                                                                                                                      |
|--------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Number of totally scanned files      | 27                                                                                                                                                                   |
| Number of files require refactorings | 1                                                                                                                                                                    |
| Number of refactoring categories     | 2                                                                                                                                                                    |
| Detected refactoring categories      | 1. Refactoring #14: Replace the AngularJS specific features used in the converted component<br>2. Refactoring #15: Migrate the directive bindings as component props |
| Number of refactoring occurrences    | 41                                                                                                                                                                   |

*Table 28: Scan results for Angular-Phonecat application after generating PhoneDetailComponent*

According to the scanning results there were two main refactoring detected in the PhoneDetailComponent file which are Refactoring #14 and Refactoring #15. The

react component had AngularJS specific features such as ng-src, ng-class, ng-if and ng-click which required it to be replaced by the ReactJS alternatives. The scope references which were there in the component code also were detected and had to be replaced with component scope and the props. Following figure shows the highlighted refactoring in the Visual Studio for the PhoneDetailComponent.

```

JS PhoneDetailComponent.js X
app > react > src > JS PhoneDetailComponent.js > ...
1 import React from 'react'
2 import { Component } from 'react'
3
4 class PhoneDetailComponent (JSX attribute) ng-src: string
5 render() {
6   return (
7     <div>
8       <div cl
9         
11     <h1>{{ctrl.phone.name}}</h1>
12     <p>{{ctrl.phone.description}}</p>
13     <ul className="phone-thumbnails">
14       <li ng-repeat="img in $ctrl.phone.images">
15         
16       </li>
17     </ul>

```

Figure 19: Highlighted AngularJS specific feature refactoring in PhoneDetailsComponent

```

JS PhoneDetailComponent.js X
app > react > src > JS PhoneDetailComponent.js > ...
1 import React from 'react'
2 import { Component } from 'react'
3
4 class PhoneDetailComponent extends Component {
5   render() {
6     return (
7       <div>
8         <div>
9           <im
10         </div>
11         <h1>{{ctrl.phone.name}}</h1>
12         <p>{{ctrl.phone.description}}</p>
13         <ul className="phone-thumbnails">
14           <li ng-repeat="img in $ctrl.phone.images">
15             
16           </li>
17         </ul>

```

Figure 20: Highlighted AngularJS scope reference refactoring in PhoneDetailComponent

Once all the highlighted refactoring are completed, PhoneDetailComponent is registered with AngularJS using react2angular middleware. Then AngularJS route which was rendering the “phone-detail” directive was replaced with the “phone-detail-component” and the application was able to successfully render the phone details. The directive migration took nearly 6 hours including setting up ReactJS for the application. Once the migration for the first directive is completed, the same process is followed for the next available directive which is “phone-list”. That migration could complete within 2 hours. With that the majority of the Angular-PhoneCat migration work was completed and following screenshots indicate the

working application and the component tree in the chrome dev tool after the migration.

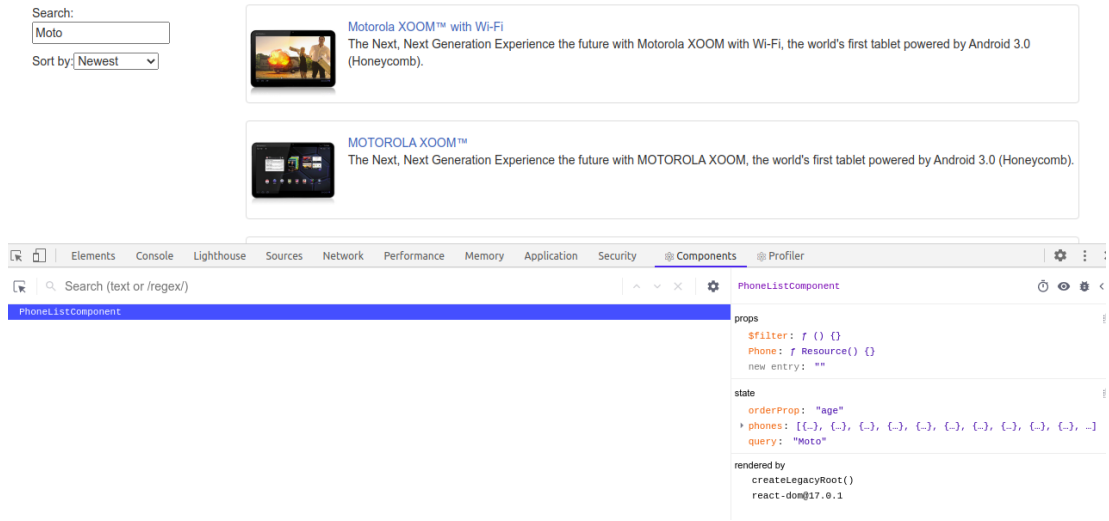


Figure 21: Migrated PhoneListComponent

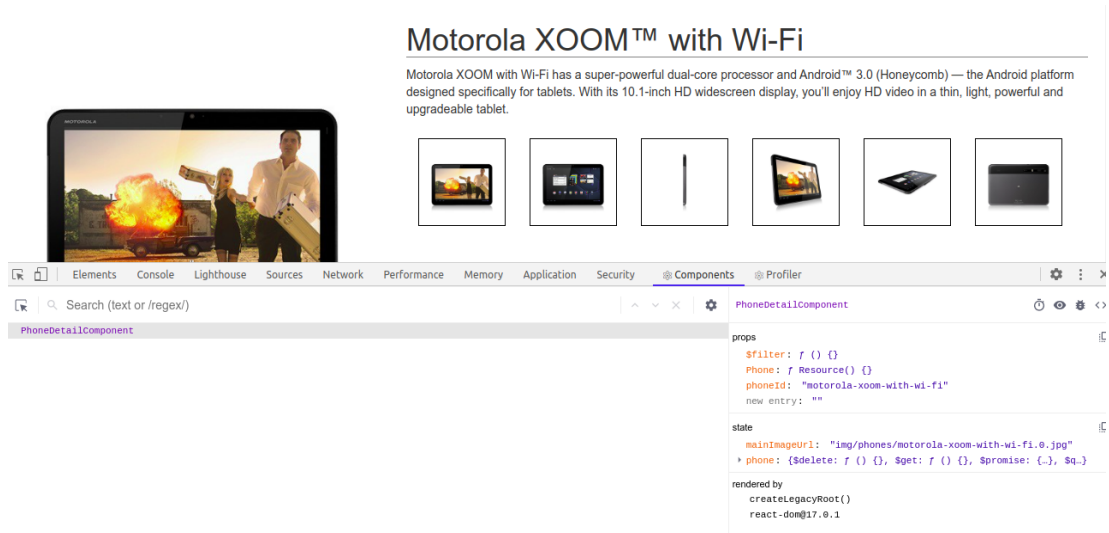


Figure 22: Migrated PhoneDetailedComponent

With this evaluation we can confirm the proposed migration approach and the tool suits to migrate small AngularJS applications. Since the small AngularJS applications are not very complicated and common, the evaluation doesn't need to extend for more small size benchmark applications so the evaluation was extended to the next level of applications.

## 5.2. Migrating medium level of AngularJS applications

The “AngularJS CRUD application demo”[45] application was selected as the medium size benchmark application. Once the application is set up, the Ng-React Copilot tool was used to scan the codebase and check whether there are initial refactoring to be done. Figure 23 shows results of the initial scan and refactoring were found in 11 files out of 89 scanned files.

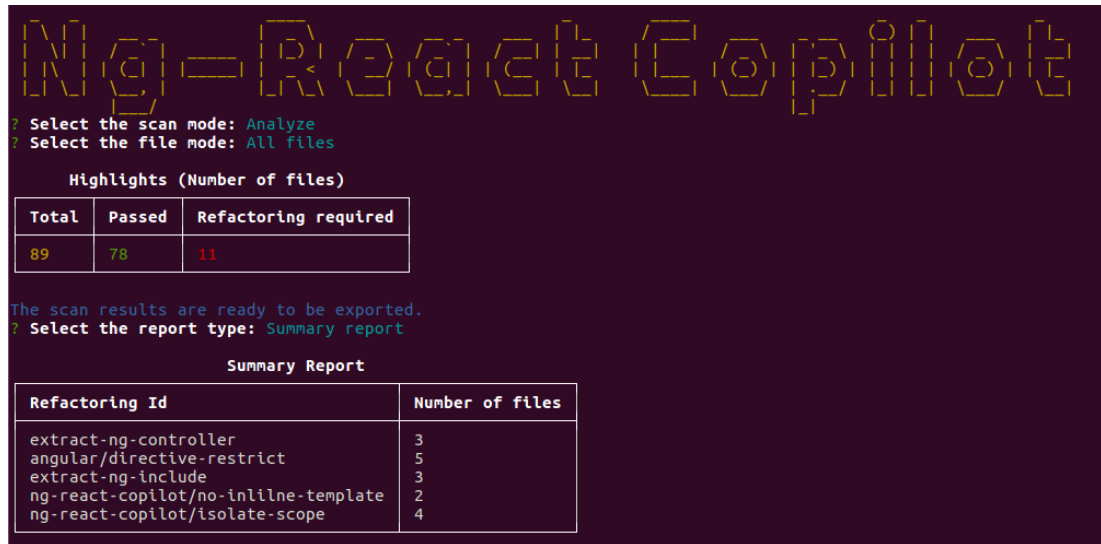


Figure 23: Initial scan results for AngularJS CRUD application

|                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|--------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Number of totally scanned files      | 89                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Number of files require refactorings | 11                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Number of refactoring categories     | 5                                                                                                                                                                                                                                                                                                                                                                                                                               |
| Detected refactoring categories      | <ol style="list-style-type: none"> <li>1. Refactoring #1: Place all the code of a directive together</li> <li>2. Refactoring #2: Favor directives as elements over attributes, comment and classes</li> <li>3. Refactoring #4: Replace ng-include with a directive</li> <li>4. Refactoring #6: Extract controller as a directive</li> <li>5. Refactoring #12: Replace two-way data binding with one-way data binding</li> </ol> |
| Number of refactoring                | 17                                                                                                                                                                                                                                                                                                                                                                                                                              |

|             |  |
|-------------|--|
| occurrences |  |
|-------------|--|

*Table 29: Initial scan results for AngularJS CRUD application*

After the initial scan the “crudButtons” directive was selected as the first directive to migrate. The template code was extracted to a template file and used the Ng-React Copilot tool to generate the CrudButtonComponent. After the component generation, the generated component had 6 refactoring and all of them were to replace AngularJS inbuilt directives. Once those were corrected, the React2Angular middleware was used to bind the component to AngularJS application. The React2Angular middleware didn’t support AngularJS versions lower than 1.5 and the AngularJS CRUD application had 1.2 version of AngularJS. This introduced a new challenge to this migration.

There were two options to followed here,

1. Upgrade the AngularJS version used in the application to 1.5 version and then apply the same refactoring methods.
2. Find out a React2Angular alternative which will support AngularJS versions prior to 1.5.

By looking at a few libraries, I was able to find NgReact[46] library which acts as a middleware between AngularJS and React applications while supporting prior versions than AngularJS 1.5. Since it’s a bit old library, the official documentation of the NgReact suggests to use React2Angular for latest versions of React so NgReact will be recommended only in situations where React2Angular cannot be used.

The migrated CrudButton was injected to the AngularJS application using NgReact middleware and the functionality was working as expected. After the CrudButton migration, the Toolbar was migrated. The toolbar was a directive and it’s template was used to generate the ToolbarComponent. Once it was generated, Ng-React Copilot tool highlighted 7 refactoring to be done.

```

Ng-React Copilot
? Select the scan mode: Analyze
? Select the file mode: Single file
? Enter the file path: ../angular-app/client/src/common/security/login/ToolbarTplComponent.js

Highlights (Number of files)


| Total | Passed | Refactoring required |
|-------|--------|----------------------|
| 1     | 0      | 1                    |



The scan results are ready to be exported.
? Select the report type: Detailed report

/home/thilanka/www/orangehrm-dev-environment/html/OHRMStandalone/TEST/js/angular-app/client/src/common/security/login/ToolbarTplComponent.js
9:13 error Replace ng-show with native conditions. Ex: this.props.showMe ? <button /> ng-react-copilot/replace-angular-inbuilt-directives
10:24 error Refactor the Angular scope references with props or state variables ng-react-copilot/replace-angular-scope-references-l
n-react-components
10:48 error Refactor the Angular scope references with props or state variables ng-react-copilot/replace-angular-scope-references-l
n-react-components
12:13 error Replace ng-show with native conditions. Ex: this.props.showMe ? <button /> ng-react-copilot/replace-angular-inbuilt-directives
14:44 error Replace ng-click usage with onclick behavior ng-react-copilot/replace-angular-inbuilt-directives
17:13 error Replace ng-hide with native conditions. Ex: this.props.showMe ? <button /> ng-react-copilot/replace-angular-inbuilt-directives
19:43 error Replace ng-click usage with onclick behavior ng-react-copilot/replace-angular-inbuilt-directives

✖ 7 problems (7 errors, 0 warnings)

```

Figure 24: Scan results for ToolbarComponent of the AngularJS CRUD application

|                                      |                                                                                                                                                                                                                              |
|--------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Number of totally scanned files      | 1                                                                                                                                                                                                                            |
| Number of files require refactorings | 1                                                                                                                                                                                                                            |
| Number of refactoring categories     | 2                                                                                                                                                                                                                            |
| Detected refactoring categories      | <ol style="list-style-type: none"> <li>1. Refactoring #14: Replace the AngularJS specific features used in the converted component</li> <li>2. Refactoring #15: Migrate the directive bindings as component props</li> </ol> |
| Number of refactoring occurrences    | 7                                                                                                                                                                                                                            |

Table 30: Scan results for ToolbarComponent of the AngularJS CRUD application

Once those 7 refactoring were done, the component was injected to AngularJS using NgReact middleware. NgReact provided a method to inject the AngularJS services into React components so that mechanism was used to call AngularJS services within ToolbarComponent.

The Login form was selected as the next component to be migrated. The Login form was not a directive but a controller and a template. The controller was used without sharing the scope with any other code so it was decided to migrate the controller and the template directly as a React component. Here I bypassed the step of first

migrating the code to an AngularJS directive because I was sure about the isolation of the code. But if it wasn't the case, first the code and the scope had to be isolated as a directive before generating the React component. The login form template was used to generate the FormComponent and once the component was generated, there were 10 refactoring highlighted from the Ng-React Copilot tool.

```

? Select the scan mode: Analyze
? Select the file mode: Single file
? Enter the file path: ../angular-app/client/src/common/security/login/FormTplComponent.js

Highlights (Number of files)


| Total | Passed | Refactoring required |
|-------|--------|----------------------|
| 1     | 0      | 1                    |



The scan results are ready to be exported.
? Select the report type: Detailed reoprt

/home/thilanka/www/orangehrn-dev-environment/html/DHRMStandalone/TEST/js/angular-app/client/src/common/security/login/FormTplComponent.js
12:48 error Replace ng-show with native conditions. Ex: this.props.showMe ? <button /> ng-react-copilot/replace-angular-inbui
lt-directives
13:14 error Refactor the Angular scope references with props or state variables ng-react-copilot/replace-angular-scope
-references-in-react-components
15:46 error Replace ng-show with native conditions. Ex: this.props.showMe ? <button /> ng-react-copilot/replace-angular-inbui
lt-directives
16:14 error Refactor the Angular scope references with props or state variables ng-react-copilot/replace-angular-scope
-references-in-react-components
20:44 error Refactor ng-model usage with component scope ng-react-copilot/replace-angular-inbui
lt-directives
22:46 error Refactor ng-model usage with component scope ng-react-copilot/replace-angular-inbui
lt-directives
25:53 error Replace ng-click usage with onClick behavior ng-react-copilot/replace-angular-inbui
lt-directives
25:72 error Replace ng-disabled with native HTML functionality. Ex: <button disabled={condition} /> ng-react-copilot/replace-angular-inbui
lt-directives
26:41 error Replace ng-click usage with onClick behavior ng-react-copilot/replace-angular-inbui
lt-directives
27:54 error Replace ng-click usage with onClick behavior ng-react-copilot/replace-angular-inbui
lt-directives

✖ 10 problems (10 errors, 0 warnings)

```

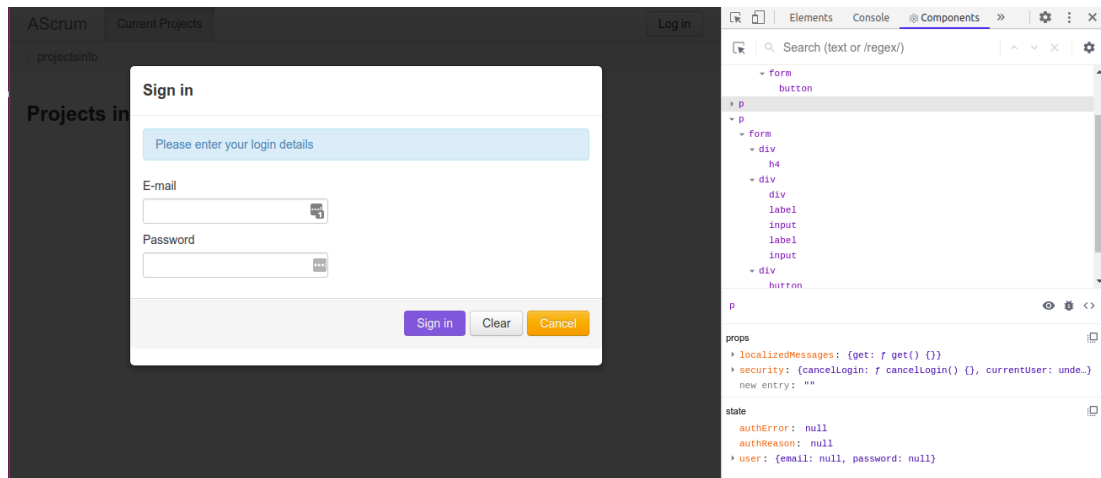
Figure 25: Scan results for FormComponent of the AngularJS CRUD application

|                                      |                                                                                                                                                                                                                              |
|--------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Number of totally scanned files      | 1                                                                                                                                                                                                                            |
| Number of files require refactorings | 1                                                                                                                                                                                                                            |
| Number of refactoring categories     | 2                                                                                                                                                                                                                            |
| Detected refactoring categories      | <ol style="list-style-type: none"> <li>1. Refactoring #14: Replace the AngularJS specific features used in the converted component</li> <li>2. Refactoring #15: Migrate the directive bindings as component props</li> </ol> |
| Number of refactoring occurrences    | 10                                                                                                                                                                                                                           |

Table 31: Scan results for FormComponent of the AngularJS CRUD application

Once those refactoring were done, the FormComponent was also injected to AngularJS application using NgReact copilot. With that, the full login form

functionality was successfully migrated to React and integrated into the AngularJS existing application. Following screenshot shows how the UI looked after the migration and how it's indicated in React Dev Tool.



*Figure 26: Migration completed Login Form of the AngularJS CRUD application*

Initially I had to spend around 20 hours resolving the compatibility issue of AngularJS 1.2 with React2Angular and adapting NgReact as the middleware. Once it's done, I was able to complete the migration of the toolbar and the login components within 5 hours.

With this evaluation we can confirm the proposed migration approach and the Ng-React Copilot tool suits to migrate medium size of AngularJS applications. Since this was done as a proof of concept and since it requires lots of time and effort to understand the internal logics of the application, the time was not spent to migrate the full application but selected areas were migrated to confirm the migration approach can handle medium size of AngularJS application migrations.

### **5.3. Migrating Enterprise level of AngularJS applications**

The “OrangeHRM Enterprise”[47] application was selected as the Enterprise level benchmark application for the migration. After the unexpected version conflict identified in the previous application migration, the current AngularJS version of OrangeHRM was checked to avoid future issues. It was using an AngularJS version

higher than 1.5 so then the Ng-React Copilot was used to scan the frontend code of OrangeHRM. Figure 27 shows the initial scanning results.

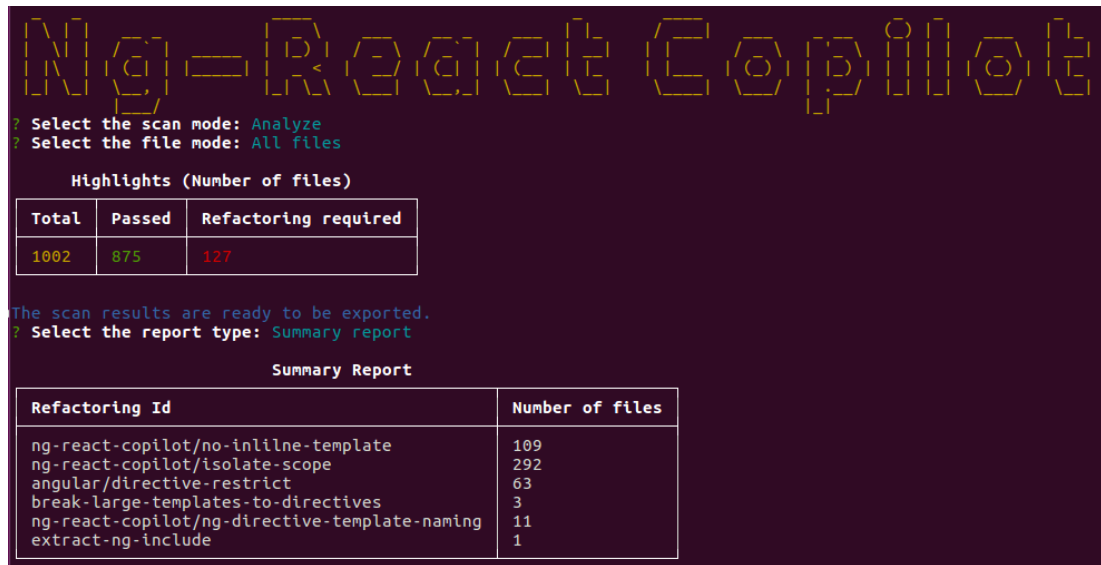


Figure 27: Initial scan for OrangeHRM application

|                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Number of totally scanned files      | 1002                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| Number of files require refactorings | 127                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| Number of refactoring categories     | 6                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Detected refactoring categories      | <ol style="list-style-type: none"> <li>1. Refactoring #1: Place all the code of a directive together</li> <li>2. Refactoring #2: Favor directives as elements over attributes, comment and classes</li> <li>3. Refactoring #4: Replace ng-include with a directive</li> <li>4. Refactoring #7: Extract directive from large templates</li> <li>5. Refactoring #6: Extract controller as a directive</li> <li>6. Refactoring #12: Replace two-way data binding with one-way data binding</li> </ol> |
| Number of refactoring occurrences    | 479                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |

Table 32: Initial scan results for OrangeHRM application

After the initial scan, since there were many refactoring to be done, it was decided to start the conversion with a relatively complex and widely used directive in OrangeHRM so it can cover a major part of the application. List directive was a widely used directive to display all the tabular data and it was used 87 times within the OrangeHRM application. Since the list component was designed in a way to display many complex types and formats of data which supporting other functionalities such as sorting, paginating, selecting, clicking and navigating, that had a relatively a complex code which gives us a good opportunity to evaluate the capabilities of Ng-React Copilot tool. As the next step, the List directive was scanned using Ng-React Copilot tool to identify the refactoring in it and following were the results.

| Highlights (Number of files) |        |                      |
|------------------------------|--------|----------------------|
| Total                        | Passed | Refactoring required |
| 1                            | 0      | 1                    |

The scan results are ready to be exported.  
? Select the report type: Detailed report

```

/home/thilanka/www/orangehrm-dev-environment/html/OHRMStandalone/TEST/ohrmreact/synfony/web/client/app/directives/list.directive.js
20:17 error Avoid two-way data binding in directive scope. Use "<" or "@" for inputs and "&" for callback functions ng-react-copilot/isola
te-scope
21:17 error Avoid two-way data binding in directive scope. Use "<" or "@" for inputs and "&" for callback functions ng-react-copilot/isola
te-scope
22:17 error Avoid two-way data binding in directive scope. Use "<" or "@" for inputs and "&" for callback functions ng-react-copilot/isola
te-scope
23:17 error Avoid two-way data binding in directive scope. Use "<" or "@" for inputs and "&" for callback functions ng-react-copilot/isola
te-scope
24:17 error Avoid two-way data binding in directive scope. Use "<" or "@" for inputs and "&" for callback functions ng-react-copilot/isola
te-scope
✖ 5 problems (5 errors, 0 warnings)

```

Figure 28: Scan results for List directive of the the OrangeHRM application

|                                      |                                                                            |
|--------------------------------------|----------------------------------------------------------------------------|
| Number of totally scanned files      | 1                                                                          |
| Number of files require refactorings | 1                                                                          |
| Number of refactoring categories     | 1                                                                          |
| Detected refactoring categories      | 1. Refactoring #12: Replace two-way data binding with one-way data binding |
| Number of refactoring occurrences    | 4                                                                          |

Table 33: Initial scan results for List directive in OrangeHRM application

The “Refactoring #12: Replace two-way data binding with one-way data binding” was detected in the List directive for 4 times and that was corrected to make it ready for the migration. After that the List directive template was used to generate the React ListComponent and then scanned it for obvious refactoring to perform using Ng-React Copilot. Following were the scanning results for the ListComponent.

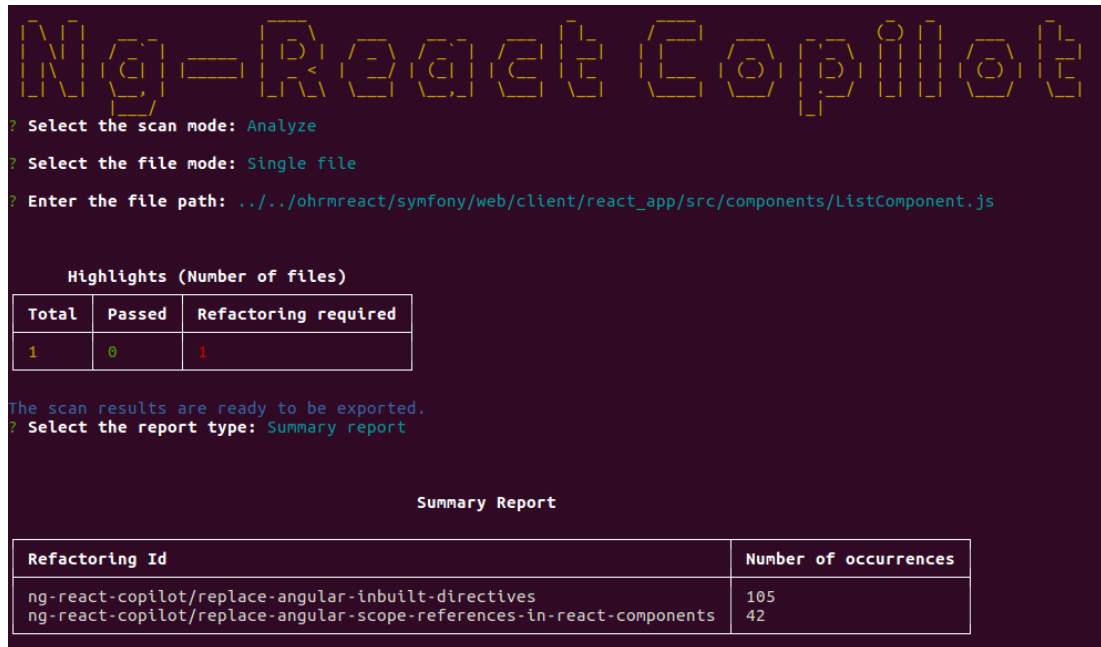


Figure 29: Scan results for ListComponent of the the OrangeHRM application

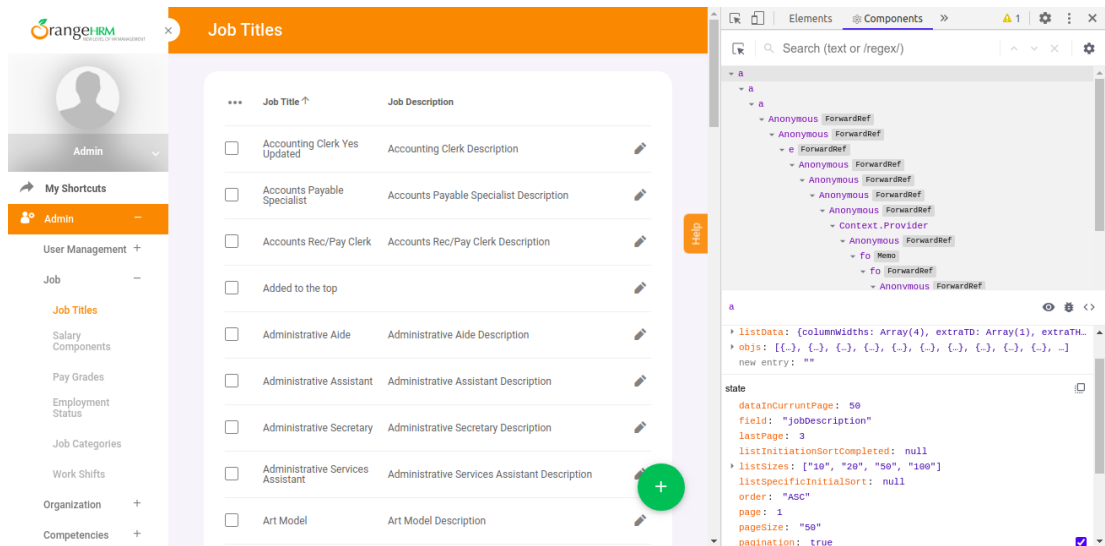
|                                      |                                                                                                                                                                                                                              |
|--------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Number of totally scanned files      | 1                                                                                                                                                                                                                            |
| Number of files require refactorings | 1                                                                                                                                                                                                                            |
| Number of refactoring categories     | 2                                                                                                                                                                                                                            |
| Detected refactoring categories      | <ol style="list-style-type: none"> <li>1. Refactoring #14: Replace the AngularJS specific features used in the converted component</li> <li>2. Refactoring #15: Migrate the directive bindings as component props</li> </ol> |
| Number of refactoring occurrences    | 147                                                                                                                                                                                                                          |

Table 34: Initial scan results for ListComponent in OrangeHRM application

There were many places with AngularJS specific directive usage and scope usages in the template and also there were many duplicated code blocks. While converting the AngularJS specific directives, the duplicate code was removed from the ListComponent. After fully converting the JSX code, the List directive logic was moved to ListComponent one by one. Most of the functions were able to migrate to the component just by replacing the scope variables with props and state variables. Following are some of the main steps followed during ListComponent migration.

1. The pagination widget and select widget in the ListComponent were identified as reusable functionalities so those were extracted as ListPagerComponent and SelectComponent.
2. The select dropdown had the dependency with Jquery based material select behavior. This was violating the React way of doing things so the select dropdown was replaced with the select widget provided by material-ui library. To make the select option UI look same as the existing select dropdown, the extension points provided by MaterialUI library to style it was used.
3. Some changes were done to the parent controller logic to support the new callback behavior avoiding two way data binding references.
4. There was an issue where the react2angular middleware doesn't rerender the react components when updating the props. This is because react2angular creates PureComponents and it does a shallow comparison to identify prop changes. Due to this, I had to do a total replace of the props in the angular application when doing a mutation. This was achieved using angular.copy the same scope variable which was passed to the react component and assigning it back to the same variable.

Once the ListComponent migration was done, the necessary tools and libraries for React to work were installed in the OrangeHRM system. Then the ListComponent was injected to Angular using react2angular middleware. Following image shows how the migrated ListComponent is working in the OrangeHRM system.



*Figure 30: Migration completed ListComponent of the OrangeHRM application*

This migration took nearly 32 hours to complete because there were unexpected complications found during the migration.

With this evaluation we can confirm the proposed migration approach and the Ng-React Copilot tool suits to migrate enterprise level AngularJS applications, but the time taken to migrate components increases with the complexity of the application. The migration of the directive logic took the majority of the time since it is done as a manual step, it would have been easy if Ng-React Copilot can provide some guidance on migrating that logic. The migration was not done for the whole application with the interest of time and only done as proof of concept.

#### 5.4. Evaluation of the results

By scanning and refactoring code in small, medium and large AngularJS applications, we evaluated the effectiveness of the Ng-React Copilot tool and the migration process.

|                                | Small | Medium | Enterprise |
|--------------------------------|-------|--------|------------|
| <b>Total Files</b>             | 27    | 89     | 1002       |
| <b>Refactoring occurrences</b> | 41    | 44     | 630        |

|                 |   |   |   |
|-----------------|---|---|---|
| Refactoring #1  |   | ✓ | ✓ |
| Refactoring #2  |   | ✓ | ✓ |
| Refactoring #3  |   |   |   |
| Refactoring #4  |   | ✓ | ✓ |
| Refactoring #6  |   | ✓ |   |
| Refactoring #7  |   |   | ✓ |
| Refactoring #12 |   | ✓ | ✓ |
| Refactoring #13 | ✓ | ✓ | ✓ |
| Refactoring #14 | ✓ | ✓ | ✓ |
| Refactoring #15 | ✓ | ✓ | ✓ |

*Table 35: Summary of all detected refactorings*

Following are the extracted evaluation of the observed results.

File search efficiency of the Ng-React Copilot: The time to scan was under 10 seconds for all three cases so the speed of the tool was at an okay state. But there was an issue where it was not omitting library files, when those were not on standard folders and led to many false positives. As an example, the tool was omitting folders such as 'bower\_components' and 'node\_modules' to avoid library files, but some applications have them in custom folders such as 'lib'. This would have been avoided by providing a configuration option to enter the paths of folders to skip and include.

Ease of following the migration workflow: The migration of small applications was relatively straightforward since the logic and the code was not complicated in them. But when migrating enterprise level applications, the logics and the complexity of the code increases which decreases the ability to straightforward migrations. In enterprise level application scanning, the tool highlights hundreds of refactoring to be done at all the stages in the migration process at once, so it's not obvious where to

start. So a thorough application knowledge is needed to decide where to start the migration. If the Ng-React Copilot tool could have given a step by step guidance highlighting refactoring needed for each stage rather than highlighting all the refactoring at once, the migration process would have been easier. Also if Ng-React tool could have shown a graph of the directives of the application, it would have been easier to identify the leaf nodes of the component tree to start the migration.

Top time savers in automated migration: The most efficient guidance of the Ng-React Copilot tool is provided to auto generate the React component from the directive template and highlight the AngularJS specific usages. This saves a considerable amount of manual and repeated work.

Even though The Ng-React Copilot tool assists to do the migration, still it takes a considerable amount of time to complete the migration for a small part of the application. The majority of the time is taken to refactor the existing AngularJS code to become isolated directives and once the isolated directives are there, the migration can be done faster. In order to make the AngularJS code as isolated directives, we have to spend lots of time to solve the scope soup inefficiencies those are having. Ng-React Tool covers only very limited refactoring on this scope soup resolution so the developer has to have a thorough understanding about the data flow and the code to correct this issue. Also when the AngularJS code is written around controllers and templates a lot rather than directives, converting those controllers to directives also takes a considerable amount of time and the Ng-React Copilot tool doesn't provide any assistance in that area. The Ng-React Copilot tool should be improved to cover these aspects of the migration process.

The next time consuming migration step is to migrate the logics in the application and directives. Currently this is done manually and Ng-React Copilot only helps to identify the scope references and AngularJS specific features but unless the logic is encapsulated in pure functions, those require changes during the migration. If the Ng-React tool could have been improved to guide the logic migration as well it could save a considerable amount of migration time.

When AngularJS UI was depending on external libraries such as JQuery based controls, the migration became hard since React's way of doing things is different from the direct DOM manipulations used in those libraries. The same issue was there with JS event systems. The new libraries which are compatible with React had to be introduced manually without the help of Ng-React Copilot which took considerable amount of time in here. But currently there was no obvious improvement to the Ng-React Copilot tool to avoid this problem.

Accuracy of the refactoring detection: The accuracy of the refactoring detection was quite good but there were some false positives detected during the scanning. Following are the examples.

1. “Refactoring #6: Extract controller as a directive” was highlighting all the occurrences of the ng-controller directive usages but some of them are needed at the root level of AngularJS component tree.
2. “Refactoring #7: Extract directive from large templates” was introduced to force developers to extract child components from larger parent components. But directives which were below to the limits were passing the limits after migrating to React components. So the extraction was more prominent after the migration.
3. “Refactoring #15: Migrate the directive bindings as component props” was assisted by Ng-React Copilot tool but when the variables were inside inline functions in JSX, they were detected as to be converted to props and scope variables but were false positives. The algorithm needs some improvements to avoid these false positives.

There should be improvements done in the algorithms to avoid these false positives in Ng-React Copilot tool.

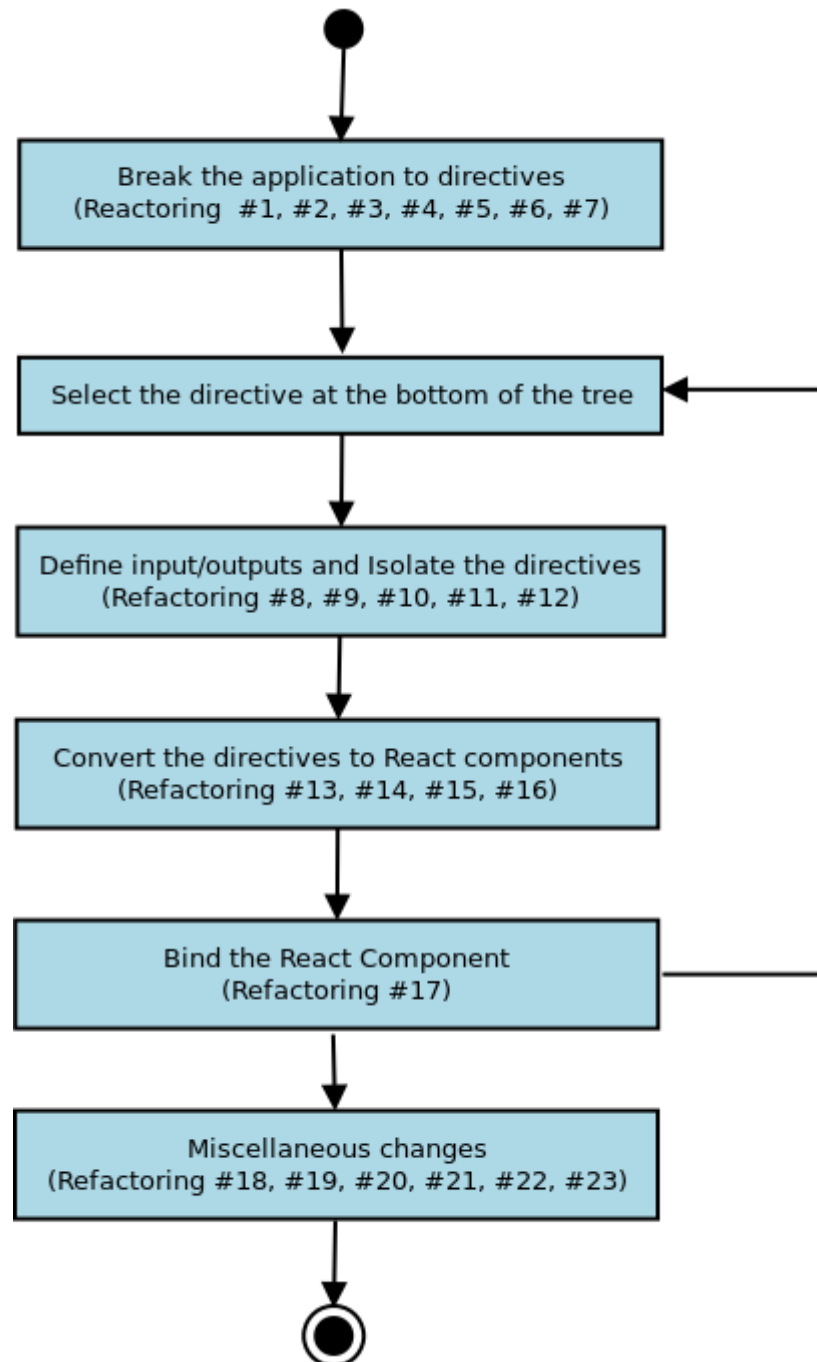
## 6. CONCLUSION

This research was carried out to identify pain points in migrating legacy AngularJS applications to newer component based React architecture and to come up with a clear set of guidelines and tools to assist the migration process to save hundreds of hours which will be wasted initially. During the research, the similar efforts and approaches which were taken before were referred to gather the information and then that information were converted to a framework which can be universally referred to in this migration process. The bottom up approach of migrating the application small part by part was selected as the recommended approach where both AngularJS code and React code will coexist during the migration lifetime. The proposed framework contains a set of identified refactoring and their order to follow in the migration so it assists the developer to migrate a small part of the application at a given time. Following is the proposed migration framework and the workflow.

| Refactoring                                                                       | Purpose                                            |
|-----------------------------------------------------------------------------------|----------------------------------------------------|
| Refactoring #1: Place all the code of a directive together                        | Break the application to directives                |
| Refactoring #2: Favor directives as elements over attributes, comment and classes |                                                    |
| Refactoring #3: Eliminate transclusion                                            |                                                    |
| Refactoring #4: Replace ng-include with a directive                               |                                                    |
| Refactoring #5: Extract root controller of a directive                            |                                                    |
| Refactoring #6: Extract controller as a directive                                 |                                                    |
| Refactoring #7: Extract directive from large templates                            |                                                    |
| Refactoring #8: Replace external reference with bound inputs                      | Define the input/outputs and isolate the directive |
| Refactoring #9: Replace external effect with bound outputs                        |                                                    |

|                                                                                                                                                       |                                            |
|-------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------|
| Refactoring #10: Isolate directive scope                                                                                                              |                                            |
| Refactoring #11: Replace state mutation with bound output                                                                                             |                                            |
| Refactoring #12: Replace two-way data binding with one-way data binding                                                                               |                                            |
| Refactoring #13: Convert the directive template as JSX in react component                                                                             | Convert the directives to React components |
| Refactoring #14: Replace the AngularJS specific features used in the converted component                                                              |                                            |
| Refactoring #15: Migrate the directive bindings as component props                                                                                    |                                            |
| Refactoring #16: Migrate the directive logic into the component                                                                                       |                                            |
| Refactoring #17: Bind the React component to AngularJS using React2Angular middleware                                                                 | Bind the React component to AngularJS      |
| Refactoring #18: Extract container components from presentational components when too many properties passed down through the intermediate components | Miscellaneous changes                      |
| Refactoring #19: Convert the angular service as a javascript module and import it in the container component                                          |                                            |
| Refactoring #20: Wrap the angular service as a javascript module and import it in the container component                                             |                                            |
| Refactoring #21: Introduce a UI library which provide React favored form widgets                                                                      |                                            |
| Refactoring #22: Migrate widgets with custom directives to components                                                                                 |                                            |
| Refactoring #23: Migrate formatting, sanitizing and validations to services                                                                           |                                            |

*Table 36: Migration Framework*



*Figure 31: Workflow of the migration framework*

In order to make the migration process more effective, Ng-React Copilot tool was developed to automate 9 steps in the framework. The tool was capable of analyzing a given codebase, detecting the possible refactoring for the migration, highlighting them to the developers and even providing them the option to perform some automatic refactoring. Coming up with the tools and algorithms for the refactoring

detection and building the tool in a way to support popular IDEs and command lines was also done as part of this research. The different coding styles used in the AngularJS application were analyzed when defining algorithms to detect refactorings and except for few false positives, the accuracy of the refactoring detection was at an acceptable level.

Once the implementation is completed, the proposed framework and the tool was evaluated by applying it to migrate small, medium and enterprise level applications one in each category and each of these application migrations, Ng-React Copilot was able to detect all 9 refactoring which were automated with it. With the help of the migration framework and the Ng-React Copilot, the small application was fully migrated to React while the medium and enterprise applications were migrated partially with the interest of the time. The results show that the proposed framework and the tool is useful when migrating the AngularJS applications in all these three scales. The framework and the tool fulfills the needs of migrating small and medium size applications and gives clear guidance for developers to migrate. But migration becomes complex and time consuming when the complexity of the application increases. The framework satisfies most of the needs of enterprise level application migrations but the automation tool needs to be improved to cover uncovered manual refactoring proposed in the framework to actually make it very useful in enterprise level application.

### **6.1. Limitations**

As discussed in the evaluation of results section, in order to make the framework and the tools effective most the uncovered automatic refactoring needs to be automated and added to Ng-React Copilot tool. Following are a list of currently identified limitations of the framework Ng-React Copilot tool.

1. The Ng-React Copilot tool doesn't have an option to configure what are the files to exclude and what are the files to include. This limitation leads to manual verification of false positives which consumes valuable developer time.

2. Even though the framework provides a workflow for the migration, it's not reflected in Ng-React Copilot tool which shows all the refactoring at the same time which doesn't give a clear idea for the user where to start the migration process. If the tool can suggest places to start the migration, it can save some more development time.
3. There is no automated way to identify the leaf nodes of the AngularJS application to start the migration so a thorough understanding of the application and conscious decision making is required to pick leaf nodes to start the migration. The impact of this limitation exponentially grows when migrating enterprise level applications.
4. Not having automation steps to most time consuming steps in the enterprise application migration process such as covering controllers to directives, avoid scope soups in the code and making directives isolated, migrating logics of the directives. These refactoring are harder to detect programmatically so they were not implemented in the initial phase of this research.
5. The framework doesn't clearly give solutions to avoid the coupling between external libraries so that burden is passed to the developer to decide which decisions to make then. This requires a research on possible such commonly used libraries and to identify the levels of coupling between them. It may require even to find out the possible alternatives for such libraries in React.
6. There were false positives detected in Ng-React Copilot tool which require the developer to go through and decide whether it's false positives or not. This makes the migration process inefficient and misguides the developers so they cannot run the tool and achieve 0 pending refactoring as results.

## **6.2. Future work**

The future expansions of this project can immediately focus on improving the limitations mentioned in the limitations section. Proposed solutions to overcome these limitations are as follows.

1. Improve the Ng-React tool to accept configurations through a config file so it can be customized based on the application nature.

2. The tool needs to be improved to cover the most time consuming steps which are manually done now. At the same time those steps should be given as a workflow without letting the user spend time to figure out where to start the migration process.
3. It's better to improve the tool to identify the AngularJS directive usage in the code and create a component map and highlight the leaf nodes to start the migration process.
4. Further research should be done on what are the most common couplings AngularJS has with external libraries and provide some guidelines to refactor them in a standard way. It should be incorporated into the Ng-React tool.
5. The logics of the detection algorithms should be improved in Ng-React Copilot tool to avoid identified false positives.
6. It's good if the tool could provide some estimation for the identified refactoring based on the number of lines in the code so the developer may get a good understanding of the effort.
7. The evaluation should be done by completing a full flow of enterprise application from top to bottom so more limitations and improvements can be detected. As an example, there is no guide on the proposed framework on how to migrate routing of an AngularJS application to React.
8. Steps and guidelines to setting up React and other required libraries in an AngularJS application has not been covered in this framework and currently it's allowed to do based on developer's preference. But when it comes to a React application, there are industry standards to follow in setting up the libraries and recommendations on how to do operations such as state management. This aspect of how to properly architect an enterprise level React application should be covered and included into the Framework to make it complete.

Apart from improving the tool and the framework toward the identified improvement, the framework and the tool can be applied to migrate a full enterprise application which will allow us to find out more limitations and future work. Making the framework and the tool available for others to try out on actual migrations will help us to evaluate the tool further.

## 7. REFERENCES

- [1] M. Wanyoike, “History of front-end frameworks,” LogRocket Blog, Oct. 16, 2018. <https://blog.logrocket.com/history-of-frontend-frameworks/> (accessed May 17, 2020).
- [2] “AngularJS Architecture | Conceptual Overview & MVC pattern in AngularJS.” <https://www.educba.com/angularjs-architecture/> (accessed Jan. 17, 2021).
- [3] M. Ramos, M. T. Valente, R. Terra, and G. Santos, “AngularJS in the wild: A survey with 460 developers,” in PLATEAU 2016 - Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools, co-located with SPLASH 2016, Nov. 2016, pp. 9–16, doi: 10.1145/3001878.3001881.
- [4] J. Ravenscroft, “Migrating an Angular 1 Application to React,” Medium, May 19, 2016. <https://medium.com/@jrwebdev/migrating-an-angular-1-application-to-react-8891ec73d462> (accessed May 17, 2020).
- [5] T. Parviainen, “Refactoring Angular Apps to Component Style,” Teropa, Oct. 18, 2015. <https://teropa.info/blog/2015/10/18/refactoring-angular-apps-to-components.html> (accessed May 17, 2020).
- [6] “5 Guidelines For Avoiding Scope Soup in Angular - Techno Fattie.” <http://www.technofattie.com/2014/03/21/five-guidelines-for-avoiding-scope-soup-in-angular.html> (accessed Jan. 17, 2021).
- [7] “AngularJS: Developer Guide: Components.” <https://docs.angularjs.org/guide/component> (accessed Jan. 17, 2021).
- [8] A. Kras, “11 Tips to Improve AngularJS Performance,” ALEXKRAS.COM, Apr. 28, 2015. <https://www.alexkras.com/11-tips-to-improve-angularjs-performance/> (accessed May 17, 2020).
- [9] M. Meyer, “10 Top Mistakes AngularJS Developers Make,” airpair. <https://www.airpair.com/angularjs/posts/top-10-mistakes-angularjs-developers-make> (accessed May 17, 2020).
- [10] “5 AngularJS Antipatterns & Pitfalls | I care, I share, I’m Nathan LeClaire.” <https://nathanleclaire.com/blog/2014/04/19/5-angularjs-antipatterns-and-pitfalls/> (accessed Jan. 23, 2021).
- [11] “9 Ways to Improve AngularJS Performance - DZone Performance.” <https://dzone.com/articles/9-ways-to-improve-angularjs-performance> (accessed Jan. 17, 2021).

- [12] M. Niemelä, “Component-Based Directives in AngularJS,” airpair. <https://www.airpair.com/angularjs/posts/component-based-angularjs-directives> (accessed May 17, 2020).
- [13] S. Guan, W. Hu, and H. Zhou, “Front-end and Back-end Separation - React Based Framework for Networked Remote Control Laboratory,” in Chinese Control Conference, CCC, Oct. 2018, vol. 2018-July, pp. 6314–6319, doi: 10.23919/ChiCC.2018.8483988.
- [14] TechMagic, “React vs Angular vs Vue.js — What to choose in 2020? (updated in 2020),” TechMagic, Mar. 16, 2018. <https://medium.com/techmagic/reactjs-vs-angular5-vs-vue-js-what-to-choose-in-2018-b91e028fa91d> (accessed May 17, 2020).
- [15] E. Baer, “Using React is a Business Decision, not a Technology Choice,” formidable.com, Dec. 04, 2015. <https://formidable.com/blog/2015/12/04/using-react-is-a-business-decision-not-a-technology-choice/> (accessed May 17, 2020).
- [16] “Rethinking Separation of Concerns with React – Dotted Squirrel.” <https://dottedsquirrel.com/react/rethinking-soc-with-react/> (accessed Jan. 17, 2021).
- [17] “Angular vs ReactJS : Which one is Most In-Demand Frontend Development Framework in 2019? - GeeksforGeeks.” <https://www.geeksforgeeks.org/angular-vs-reactjs-which-one-is-most-in-demand-frontend-development-framework-in-2019/> (accessed Jan. 17, 2021).
- [18] P. Maniraho, “The Deep Dive — Migration From Angular to React,” Simple, Medium, Oct. 02, 2018. <https://medium.com/hoopeeze/the-deep-dive-migration-from-angular-to-react-ea5a807e95eb> (accessed May 17, 2020).
- [19] D. Dughila, “The 4 Layers of Single Page Applications You Need to Know ,” Hacker Noon, Feb. 28, 2018. <https://hackernoon.com/architecting-single-page-applications-b842ea633c2e> (accessed May 17, 2020).
- [20] “Understanding Transclusion in AngularJS (Part 1 of 2) - Accelebrate.” <https://www.accelebrate.com/blog/angularjs-transclusion-part-1> (accessed Jan. 17, 2021).
- [21] “Replacing ngInclude With Component Directives In AngularJS.” <https://www.bennadel.com/blog/2740-replacing-nginclude-with-component-directives-in-angularjs.htm> (accessed Jan. 23, 2021).
- [22] “AngularJS - Understanding Isolate Scope - Thinkster.” <https://thinkster.io/egghead/understanding-isolate-scope> (accessed Jan. 23, 2021).

- [23] “Passing Functions to AngularJS Directives · Europace - behind the scenes.” <https://tech.europace.de/post/passing-functions-to-angularjs-directives/> (accessed Jan. 23, 2021).
- [24] “Get to get all child scopes in Angularjs given the parent scope.” <https://www.xspdf.com/resolution/31038711.html> (accessed Jan. 23, 2021).
- [25] “One-way binding in AngularJS | Ninja Squad.” <https://blog.ninja-squad.com/2016/04/12/ng1-one-way-binding/> (accessed Jan. 23, 2021).
- [26] “JSX vs HTML: Overview + Answers to FAQs — Soshace • Soshace.” <https://soshace.com/jsx-vs-html-overview-answers-to-faqs/> (accessed Jan. 23, 2021).
- [27] “Mastering Props And PropTypes In React — Smashing Magazine.” <https://www.smashingmagazine.com/2020/08/mastering-props-proptypes-react/> (accessed Jan. 23, 2021).
- [28] D. Abramov, “Presentational and Container Components,” Medium, Mar. 23, 2015. [https://medium.com/@dan\\_abramov/smart-and-dumb-components-7ca2f9a7c7d0](https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0) (accessed May 17, 2020).
- [29] “It has begun! Moving from Angular 1.6 to ReactJS.” <http://coreysnyder.me/it-has-begun-moving-from-angular-1-6-to-reactjs/> (accessed Jan. 23, 2021).
- [30] M. Fowler, “Catalog of Refactorings,” [martinfowler.com](http://martinfowler.com/catalog/). <https://refactoring.com/catalog/> (accessed May 17, 2020).
- [31] “The easiest way to embed React components in Angular 1 apps,” [react2angular](https://github.com/coatue-oss/react2angular). <https://github.com/coatue-oss/react2angular> (accessed May 17, 2020).
- [32] “Documentation - Materialize,” [materializecss](https://materializecss.com/). <https://materializecss.com/> (accessed May 17, 2020).
- [33] “Angular Material UI component library,” Angular Material. <https://material.angular.io/> (accessed May 17, 2020).
- [34] “Material-UI: A popular React UI framework,” [material-ui](https://material-ui.com/). <https://material-ui.com/> (accessed May 17, 2020).
- [35] “ESLint - Pluggable JavaScript linter.” <https://eslint.org/> (accessed Jan. 10, 2021).
- [36] “Hello from HTMLHint | HTMLHint.” <https://htmlhint.com/> (accessed Oct. 30, 2020).
- [37] “eslint/espre: An Esprima-compatible JavaScript parser.” <https://github.com/eslint/espre> (accessed Oct. 30, 2020).

- [38] “reactjs/react-magic: Automatically AJAXify plain HTML with the power of React. It’s magic!” <https://github.com/reactjs/react-magic> (accessed Oct. 30, 2020).
- [39] “AST explorer.” <https://astexplorer.net/> (accessed Jan. 10, 2021).
- [40] johnpapa, “Angular 1 Style Guide,” [github.com. https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md](https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md) (accessed May 17, 2020).
- [41] “EmmanuelDemey/eslint-plugin-angular: ESLint plugin for AngularJS applications.” <https://github.com/EmmanuelDemey/eslint-plugin-angular> (accessed Jan. 10, 2021).
- [42] “Announcing Angular Copilot v1.0. We are pleased to announce the... | by Hanyu Xiao | AngularDoc | Medium.” <https://medium.com/angulardoc/announcing-angular-copilot-v1-0-aae80ec2083f> (accessed Jan. 10, 2021).
- [43] “Projects using AngularJS · angular/angular.js Wiki.” <https://github.com/angular/angular.js/wiki/Projects-using-AngularJS> (accessed Jan. 10, 2021).
- [44] “angular/angular-phonecat: Tutorial on building an angular application.” <https://github.com/angular/angular-phonecat> (accessed Jan. 10, 2021).
- [45] “angular-app/angular-app: Reference application for AngularJS.” <https://github.com/angular-app/angular-app> (accessed Jan. 10, 2021).
- [46] K. B. Pedersen and D. Chang, “Use React Components in Angular,” ngReact. <https://github.com/ngReact/ngReact> (accessed May 17, 2020).
- [47] “HR Management System | HR Management Software | OrangeHRM.” <https://www.orangehrm.com/> (accessed Jan. 10, 2021).