
Framework change for modernization of webservice

Master of Science Thesis
University of Turku
Department of Future Technologies
Software Engineering
2020
Alice Thomas

Supervisors:
Ville Leppänen
Jarko Papalitsas
Artem Goutsoul

Software companies that provide Software as a service, constantly seek to improve their product, by adding features according to the customer's needs. In an attempt to meet all the deadlines set by the customer, an important aspect of software development that gets neglected is code maintenance and code modernization, since this is not something that a customer demands of the software. But from a developer's stand point, a good software is one that is easy to work with and easy to maintain. In the long term ignoring code modernization will produce legacy code. Legacy code need not be code that is old, instead it can be code that is written in an outdated language, or has libraries that are no longer vendor supported. Legacy code then leads to different types of technical debt. This inadvertently affects the customer, because as the technical debt of the codebase increases, new feature development or maintenance will become more expensive and time consuming.

This was the situation in the chosen case study company. This thesis focuses on studying the different types of technical debt and the possible code modernization methods and strategies that could be applied to a legacy system and possibly reduce technical debt and make the code more maintainable and modern. For this thesis, the modernization process selected is the Chicken Little methodology. This technique allows both the legacy system and the target system to run in parallel by using gateways, and this is an important feature for this project. Especially during the client-testing phase or in the first few months after the new system is taken into production, if there are any issues with the system, customers can be directed to the old system without losing business. In each step of this technique minimal functionality is selected, there by reducing the chance of risk.

By following the steps of the chosen methodology to change the framework, the benefits identified were, easier code re-usability and thus code maintainability, reduced lines of code, more unit test cases and many more. Thereafter, concluding the thesis.

Keywords: Legacy code, technical debt, framework change, code modernization strategies, chicken little

Contents

| | | |
|----------|-------------------------------------|----------|
| 1 | Introduction | 1 |
| 1.1 | Goal | 2 |
| 1.2 | Problem Definition | 2 |
| 1.3 | Scope | 3 |
| 1.4 | Thesis structure | 3 |
| 2 | Web Development | 4 |
| 2.1 | Web Frontend technologies | 4 |
| 2.1.1 | HTML | 4 |
| 2.1.2 | CSS | 5 |
| 2.1.3 | DOM | 8 |
| 2.1.4 | JavaScript | 8 |
| 2.1.5 | TypeScript | 10 |
| 2.2 | Web Frontend frameworks | 11 |
| 2.2.1 | Backbone.js | 12 |
| 2.2.2 | Angular | 13 |
| 2.3 | Web Backend Technologies | 15 |
| 2.3.1 | Server-side languages | 15 |
| 2.3.2 | Servers | 16 |
| 2.3.3 | Databases | 16 |

| | | |
|----------|--|-----------|
| 2.4 | Web application structure | 16 |
| 2.4.1 | Traditional web application | 16 |
| 2.4.2 | Native application | 18 |
| 2.4.3 | Single page application | 19 |
| 3 | Case Study Company | 23 |
| 3.1 | Enkora Oy | 23 |
| 3.2 | The Software Product | 24 |
| 3.3 | Current state of code | 25 |
| 3.3.1 | Enkora’s codebase | 25 |
| 3.3.2 | Why the need for a change | 26 |
| 4 | Legacy Systems, Technical Debt and Modernization Techniques | 28 |
| 4.1 | Overview of Legacy systems | 28 |
| 4.2 | Technical Debt | 30 |
| 4.3 | Modernization strategies | 32 |
| 4.4 | Modernization methods | 34 |
| 4.4.1 | Chicken Little and Cold Turkey Methodologies | 34 |
| 4.4.2 | Renaissance | 38 |
| 4.4.3 | The Butterfly Methodology | 41 |
| 5 | Implementation Plan | 44 |
| 5.1 | Enkora’s Reservation System | 44 |
| 5.2 | Problems in the current state of code | 46 |
| 5.3 | Modernization plan | 49 |
| 6 | Evaluating the plan | 54 |
| 6.1 | The process | 54 |
| 6.2 | Benefits | 62 |

| | | |
|----------|--|-----------|
| 6.3 | Disadvantages | 64 |
| 6.4 | Problems faced during the implementation | 65 |
| 7 | Conclusion | 67 |
| | References | 69 |

List of Figures

| | | |
|-----|--|----|
| 2.1 | Traditional web application architecture [15] | 17 |
| 2.2 | Native applications [15] | 19 |
| 2.3 | Front-end and back-end architecture of a single page application [15] | 22 |
| 3.1 | Enkora Oy Solutions | 24 |
| 3.2 | Online Reservation System | 26 |
| 3.3 | Online Reservation System 2a | 27 |
| 3.4 | Online Reservation System 2b | 27 |
| 4.1 | An example of Chicken Little’s General Migration Architecture[42], [41] | 37 |
| 4.2 | Renaissance overview [44] | 39 |
| 4.3 | Finding candidates for evolution [44] | 40 |
| 4.4 | Butterfly methodology [41] | 43 |
| 5.1 | Intellisense in IDE | 48 |
| 5.2 | Intellisense errors | 48 |
| 6.1 | Pictorial representation of the components/modules in shop module | 55 |
| 6.2 | Calendar module | 57 |
| 6.3 | Example of specs file for a module | 60 |
| 6.4 | Github pull requests[49] | 66 |

1 Introduction

Any company that retails a product is always looking to improve the product and provide better versions of the same to their customers. This is true in every field.

When it comes to a software product, a new and improved version is identified through the added features that the product provides. Very often, in an attempt to keep up with the consumer needs, whether it be feature development, or bug fixes or just product maintenance, there is not enough time (if any at all) assigned for codebase modernization. This is not something that the customer usually concern themselves with, so, this is given a lower priority. The absence of periodic measures towards code maintenance and code modernization gives birth to the “legacy code”.

Legacy code and its problems are not a newly uncovered topic. In 1995, Bennett[1] discussed what legacy code was and what measures could be taken to handle it. A codebase is not a legacy system just because it is old (with respect to age). The term *legacy* can also be defined as a system’s inefficiency at meeting the organization’s needs and requirements, or if certain components (hardware or software) of the system lack of vendor support [2]. Bennett described a legacy system as a system using “state of the art” technology during its creation, but, due to the lack of timely “remedial actions”, the codebase has become outdated[1].

The case study company chosen is Enkora Oy. Enkora Oy is a company located in Helsinki, that provides software and hardware solutions to multiple client companies in Finland. Some of their software products include Booking or Reservation Systems,

Time tracking systems, Point of Sale systems, Information and marketing channels etc. Their hardware products are Lockers, self-service terminals, RFID Bracelets, Card or Bracelet collectors, etc.

They have a significant client base that includes health and fitness centers, swimming halls, and construction companies.

1.1 Goal

This thesis aims to study how code modernization can be done on a large codebase and understand its effects while providing the case-study company (Enkora Oy) with an updated web frontend framework that is Angular based. Currently, a large part of the code base is in CoffeeScript which implements the Backbone.js framework. This CoffeeScript and Backbone based code has to be transitioned to TypeScript to implement the Angular framework.

Also, a study on the technical debt acquired by the legacy code is done to further understand the benefits of updating and modernizing the code.

1.2 Problem Definition

At Enkora Oy, their code base is quite large. This company has been running for more than 10 years and providing software solutions during that period. Their code base uses CoffeeScript, TypeScript and regular JavaScript to provide frontend solutions and their backend solutions are written in PHP. Ultimately moving into TypeScript benefits the company, since this provides a more homogeneous frontend solution, and they get all the benefits of using TypeScript, like static type-checking.

1.3 Scope

Transitioning the whole of Enkora’s codebase is not in the scope of this thesis. In this thesis, the “reservation system”, a significant part of the Enkora product - which is a portal where customers can view the different lessons or courses provided by a client company and make reservations to these, or where products can be listed which can be purchased by customers, will be under study, to understand the effects of transitioning from one framework to another.

1.4 Thesis structure

This thesis is structured so that firstly in Chapter 1, a general idea regarding this Masters Thesis is described and the scope of the thesis is set. Next, Chapter 2, provides a basic theoretical study on the various frontend and backend technologies that Enkora uses in their codebase, also, the various frontend framework options are studied. Chapter 3, then progress to give a description about the Case study company - Enkora Oy and their software product. Also, a description of why this modernization or code change is needed by the company is provided in that chapter. In Chapter 4, topics such as Technical debt, modernization techniques and few important modernization methods are described. From this list of modernization methods, one is chosen to implement the framework transition for the case study company. Next in Chapter 5, the problems in the current legacy code are discussed, and an appropriate modernization plan is chosen, for the migration process. In Chapter 6, the process of implementing the plan for Enkora is explained, and the advantages and disadvantages of this plan are discussed. Lastly, in Chapter 7, the overall outcome of this thesis is given in brief, and thereby concluding the thesis.

2 Web Development

When creating Web based projects, choosing the right platform is very important. Prechelt[3] describes *platform* as “a combination of technological ecosystems consisting of a programming language (and possibly alternatives), one or more Web development frameworks, and a large set of reusable libraries and components and also a development culture consisting of styles, priorities, process preferences”[3]

2.1 Web Frontend technologies

For Web Frontend development there are many options on the combinations of technologies that can be used. Some of the most common technologies that are used in almost every frontend development discussed in detail below.

2.1.1 HTML

One of the basic components in the construction of a functioning web page is the Hypertext Markup Language or HTML. HTML is used to structure the different components of a web page.

In their book “The Essential guide to CSS and HTML Web Design”, Grannell and Craig define HTML documents as “text files that contain tags, which are used to mark up HTML elements”[4]. These documents are saved using the `.html` extension. An HTML tag is represented using angle brackets. A tag usually has a start tag and an end tag. The end tag is written essentially the same as the start tag except

for the forward slash which is inserted before the tag name[5]. The start tag is also called the opening tag and the end tag is called the closing tag.

There are few tags that are essential in every HTML document for it to be valid, like `<html>` (this represents the root element of the HTML page, which means that all the other elements and tags of the HTML document are within this HTML tag), `<head>` (contains the metadata of the page) and `<body>` (represents the content part of the page).

Web browsers interpret the HTML documents and display the content. Only the content within the body tags are displayed on the web page. The code is interpreted to a DOM tree, which is then rendered graphically. Some examples of Web browsers are Edge, Internet Explorer(IE), Mozilla Firefox, Google Chrome etc.

But in the latest version of HTML (HTML 5), even these essential elements can be omitted, since by default they will be added by the web browsers if they are absent in the document.

In a perfect world an HTML document would render the same on the different browsers, but that is not the case for now[6]. Cross-browser compatibility is an issue that most developers have to address every time they develop a web application[6]. This is because even though there are rules for defining and writing an HTML document, the same cannot be said for the engines or interpreters of the Web Browsers. For developers of browsers there are only suggestions which are made. For example, the HTML5 specification is well over 900 pages of which only around 300 pages are of relevance to web authors and the rest is for developers of browsers, telling them how to parse markup and even bad markup[7].

2.1.2 CSS

If only tags and elements were used to describe an HTML page, then it would look quite mundane and bland. Every website would look like a document with links

and they would not be as accessible as they are today. Adding style to an HTML document is required to give the page the intended look and feel, which is also an important factor when it comes to conveying a message through a web page.

Cascading Style Sheets or CSS is the language used for defining styles that can be applied to HTML[8]. The World Wide Web Consortium writes the CSS specifications and maintains them. CSS3 is the latest version of CSS.

There are three ways to add style elements to an HTML page. These are:

Inline style: Here the style is added within the tag of an element. This way of defining the style is used when the style needs to be applied to a limited section and when our requirements are small. The keyword 'style' has to be specified in the tag and within quotes the property, colon and value. Multiple property value pairs are separated by semicolon.

For example, `<p style='color: blue;'>`. Here the property text-color which is specified by "color" is set to "blue" for a specific "p" selector or that paragraph.

Internal style sheets: In this method the styles are specified in the same HTML document but not within the targeted HTML tag. The styles are clubbed together within the style tag which is placed in the "head" of the HTML document as shown in Listing 1. Here the HTML document's body background color is set to green and all the paragraphs have a text size of 13px.

External style sheets: Here the styles are written into a separate file. This file is saved with the .css extension. This external file is then referenced in the HTML document according to need as shown in Listing 2 using the "link" tag.

By mixing up structure (HTML) and presentation (CSS) in each document, it gets difficult to maintain the pages. Such is the case when using inline and internal

```
<html>
  <head>
    <title>Internal CSS</title>
    <style>
      body {
        background-color: green;
      }
      p {
        font-size: 13px;
      }
    </style>
  </head>
</html>
```

Listing 1: Internal css

```
<html>
  <head>
    <title>External CSS</title>
    <link type="text/css" href="../css/test.css"
      ↪ rel="stylesheet">
  </head>
</html>
```

Listing 2: External css

styles. By using Cascading Style Sheets as external style sheets there are multiple advantages.

- A CSS can be shared across multiple pages[8]
- The web pages become easier to maintain and are more flexible
- The styles applied can be customized to suit different environments and devices.

Even though CSS3 is the latest version of CSS, it builds on its predecessors. For

example, shortcomings found in CSS 2 were fixed in version 2.1. Later more features were added and CSS 3 was created. Although features may become deprecated, they would still work in existing browsers because new versions only add new functionality or refine existing definitions [8].

Nowadays web applications can be accessed through different media devices which have different screen sizes. And, a webpage which looks and works well on the laptop may look distorted on a mobile phone. In CSS3 there were media queries added which allowed to address this. Media queries can be used to identify the screen size of a user and an appropriate style sheet is loaded based on the screen size.

2.1.3 DOM

The DOM or Document Object Model is a standard interface for representing XML and HTML documents[9]. It is a language independent interface that grants programs and scripts rights to access and update the content and style of documents[10]. This means that the commands to access different nodes or elements of the DOM are the same in any language or platform. A programmer can easily navigate and manipulate the data, via the generated tree of objects, which is obtained from parsing the data[9]. The global variable *document* can be used to access the objects within the tree[11]. The first version of the DOM (level 1) was released in 1998 by the World Wide Web Consortium.

2.1.4 JavaScript

In 1995 Sun Microsystems and Netscape released a scripting language which they first called LiveScript and then later renamed it to JavaScript[12]. Even though it started off as a scripting language, it later evolved into a complete programming language. JavaScript is an interpreted object-oriented programming language which

works along with HTML to create interactive pages[13]. It is used to write client side applications, which means that the code written is sent to the user's computer when the page is loaded[13]. When dealing with JavaScript, it is also important to know about ECMAScript. It is part of ECMA, the institution that standardizes the JavaScript language under the ECMAScript specification[14]. The current version of ECMAScript is ES10 which was released in 2019. Since 2015, when the ES6 was released lots of new features were added to JavaScript. Some of these include functions, closures, loosely typed language, dynamic objects etc [15].

The interpreter within the user's web browser executes the code line by line and generates the intended interactions of the web page. JavaScript can be included in an HTML file using script tags. The script tags can be used in two ways.

Writing JS(JavaScript) code within the script tag: An example for this is shown below in Listing 3. An initialization function is added in the head of the HTML program within the script tag.

```
<html>
  <head>
    <title>Example with 'script' tag</title>
    <script>
      function init() {
        document.writeln('Hello World');
      }
    </script>
  </head>
</html>
```

Listing 3: JavaScript 'script' tag

Referencing the path to the .js file: This is another way to include JS in an HTML page. The JavaScript is written in a separate page and saved with the .js extension. This is then referenced in the script tag of the HTML page,

within which we mention the path to the js file using the tag “src” as seen in Listing 4.

```
<html>
  <head>
    <title>Example with 'src' within `script`</title>
    <script language="JavaScript" type="type/javascript"
      ↪ src="test.js">

    <script>
  </head>
</html>
```

Listing 4: JavaScript ‘script’ tag with ‘src’

2.1.5 TypeScript

TypeScript is an open source programming language, that was created to overcome the shortcomings that came when working with JavaScript on large projects. It was developed and is maintained by Microsoft[16]. It is a statically typed compiled language that generates JavaScript code that can be used in cross-platform scenarios[17]. TypeScript can also be thought of as a super set of JavaScript, so any JavaScript program would also be a valid TypeScript program.

TypeScript improves the JavaScript development model by adopting object-oriented concepts like Inheritance, Encapsulation and Abstraction and making it easier to implement. TypeScript introduces many concepts which are present in other object-oriented languages such as *static typing* (where the parameters of a function can specify their type like boolean, string etc, and this allow type checking at compile time), *classes*, *interfaces*, *generics*, *modules*. TypeScript adds a layer of static typing on top of JavaScript that is run through a compiler, which parses the TypeScript code and converts it to regular JavaScript [17]. The addition of type

safety and code compilation allows errors to be caught sooner and bugs to be eliminated without having to deploy a line of code unlike while using regular JavaScript, where the changes need to be deployed/ run before even syntax errors are caught [17]. *Classes* and *Modules* make the development of large scale applications much easier [17]. The usage of *generics* and *interfaces* in the type system allows easier creation of components and libraries which can be used with other objects as well [17].

2.2 Web Frontend frameworks

Developing applications in JavaScript has always been a challenge, this is mainly because of its malleable nature and lack of type checking[18]. There are many libraries in JavaScript that provide simple constructs which help in effectively reducing the number of lines written. An example of these libraries are jQuery, Underscore.js etc. But one thing that these different libraries lack is structural guidance, and this is especially important when the project code has grown too big[18]. This is what lead to the emergence of frameworks in JavaScript. Many of these frameworks use a *design pattern* called Model-View-Controller(MVC), which separates the elements of the application into more manageable pieces[18]. Rodzvilla[19] describes *design pattern* as “reusable solutions in software development for dealing with common problems or needs in the software design”[19] and Addy Osmani describes MVC as “an architectural design pattern that encourages improved application organization through a separation of concerns, this means that it enforces the isolation of business data(models) from user interfaces(views), with a third component (controllers)traditionally managing logic, user inputs and coordination of models and views”[20]. Based on the documentation provided by Backbone.js and Angular, we know that the Backbone framework and Angular framework are based on different implementations of the design pattern MVC.

With the emergence of technologies such as Web Components and newer versions of JavaScript(ES2015), a new design pattern was introduced, this was the component pattern[18]. In software development components are logical units that can be combined to form larger applications[18]. They have internal logic and properties that are shielded or hidden from the larger application[18]. The larger application can make use of these components through interfaces, which only exposes certain information/data that is needed to use this component. In this way, the component's internal logic can be altered without affecting the larger application, as long as the interface isn't changed[18]. Angular 4 and above make use of this Component pattern in their framework.

2.2.1 Backbone.js

Backbone is a framework, where data is represented as models. These models can be created, destroyed and saved to the server. When a user interacts with the interface and clicks or does any action that initiates an action/change, which causes a change in the data that was saved. A "change" event is triggered by the model, then the 'Views' get the state change information, and a corresponding response is generated[21]. The view is then re-rendered with this new response information. In a finished Backbone app, there is no need for code that reads from the DOM to find an element with a specific id, and update the HTML manually — when the model changes, the views simply update themselves[21].

Backbone provides structure to web applications using the following

Model: stores the data as key-value bindings[21].

View: The user interface that is generated based on a specific model.

Collection: Related Models can be grouped together to form a Collection. This is useful when saving new models to the server, as the collection acts as a

focal point to notice any change that might occur to any model within that collection[21].

2.2.2 Angular

Angular is a platform or framework for building single page web applications in HTML and TypeScript[22].

Angular has come out with many releases. Its first version was called Angular 1 or AngularJS and it was based on model-view controller whereas from Angular 4 onwards it was based on component based structures. Components as mentioned earlier are the basic building blocks of an Angular application[18]. These components are organized together in the Angular modules to form a complete application. Every Angular application will have one or more modules that contain its components.

In Angular we have *NgModules*, which collects related code into functional sets[22]. Every app has a root module, and conventionally its named *AppModule* and is saved in a file named 'app.module.ts'. This module provides the bootstrap mechanism that launches the application[22]. Like in JavaScript modules, NgModules can import functionalities from other NgModules and also export them. An example of a root NgModule definition is shown in Listing 5.

```
import { NgModule } from '@angular/core';

@NgModule({
  imports: [ ... ],
  declarations: [ AppComponent ],
  exports: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Listing 5: NgModules

Some important properties of NgModules are

Declaration: Shows the components, pipes and directives that belong to this module.

Exports: Contains the subset of the declarations that should be visible and usable in the components of other NgModules.

Imports: Lists out the modules whose classes are required by the components declared in this module.

Bootstrap: From the online Angular documentation we know that “The main application view, called the root component, which hosts all other app views. Only the root NgModule should set the bootstrap property”[23].

A *Component* defines and controls a certain portion of the view of the application, it allows modifications based on the program logic defined in it. The `@Component` decorator specifies its metadata within it [22]. The class below it becomes the component. Within the component, the class contains the application data and logic, and this is linked with an HTML *template* that describes the view which will be displayed in a target environment[22]. The Angular markup along with the HTML elements that it can modify are combined in the template[22]. Application data is connected to the DOM by using data binding. There are two types of data binding:

Event Binding: Based on the user input, the application data is updated.

Property Binding: Allows interpolation of values that are computed from the application data into the HTML.

Angular also supports *Two-way Binding* which means that changes in the DOM, generated by user actions or preferences are also reflected in the program data. The

[()] syntax combines the brackets of property binding, [], with the parentheses of event binding, ()[22].

2.3 Web Backend Technologies

The Backend of a website refers to the web-server and the database and their connections. Backend technologies refer to these and the languages used to communicate with the server and database. A website which is open on the web browser, communicates with web servers using the Hypertext Transfer Protocol (HTTP)[24]. When a link on a webpage is clicked, or a search is run, an HTTP request is sent from the browser to the target server[24]. These servers then receive these requests, process them and return an HTTP response message. The response contains a status line indicating if the request succeeded (e.g. "HTTP/1.1 200 OK" for success)[24].

2.3.1 Server-side languages

Some popular server-side languages used for writing server-side code are PHP, Python, Ruby and Node.js. In this thesis PHP is used for connecting to the database.

PHP: This is an open source, general purpose scripting language. This means that PHP can be used to write *scripts*, which are small pieces of code that tell the system to do something, like, display "Welcome" on the screen or add two numbers, and store the value in the database. PHP has wide popularity because of many reasons such as, a large technical community following that can help in providing support and guidance when needed. It is also secure, as long as the scripts are written correctly, the PHP code is never seen on the site.

2.3.2 Servers

The Dictionary of Computing defines a web server as “a server on a TCP/IP network that listens for HTTP requests addressed to it and performs the appropriate action. Basic web servers do this by transmitting a copy of a pre-prepared static web page. However, many modern web servers can create dynamic web pages and host web applications”[9].

2.3.3 Databases

Any systematised storage of data and facts, can be called a database. In Software context we have Database Management Systems to access the data within the database. The Dictionary of Computing defines the Database Management System as “a software system that provides comprehensive facilities for the organization and management of a body of information required for some particular application or group of related applications” [9]. Some popular relational database systems include ORACLE, INGRES, Sybase and Microsoft SQL Server, INFORMIX, MySQL, and PostgreSQL [9].

2.4 Web application structure

Web applications can be divided into 2 main types, they are traditional web applications and the modern single page applications. But, for better understanding of the single page applications, native applications are also discussed below.

2.4.1 Traditional web application

Server-centric web applications can also be called the traditional web application which are browser-based applications that do not need any client installation [15]. When an HTTP request is made from the browser to the server, the rendered pages

are sent back as an HTTP response. This response causes a whole web-page refresh, i.e. the whole web page is replaced by a new page [15]. Since this is server-centric, the weight on client side development is very small. As seen in Figure 2.1[15], most

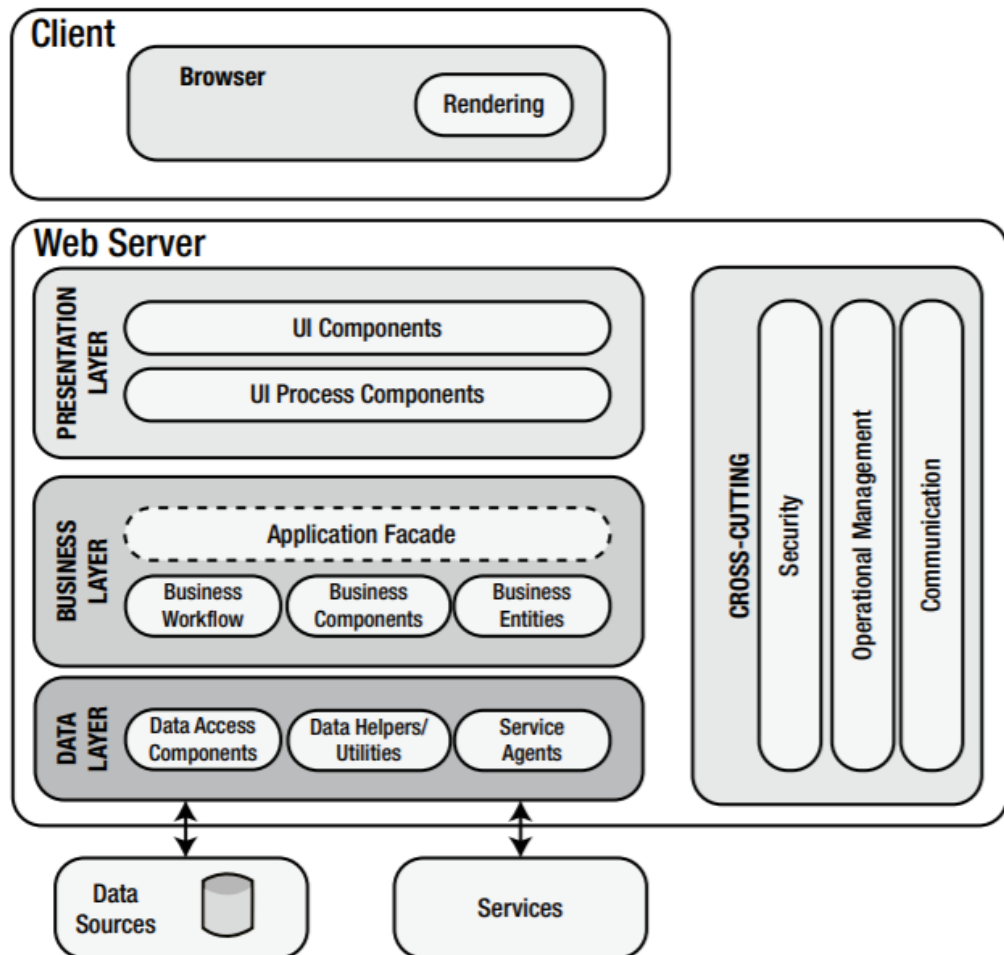


Figure 2.1: Traditional web application architecture [15]

of the application logic is stored in the web server and the client side is used only to render the web pages it receives.

There are a few problems that come with these kind of applications, the most pressing being the waiting time involved every time the web page is re-loaded. Handling events on the server can be very long and until the result comes, the web application will not be responsive. Another aspect to think of when creating a

server-centric application is, state persisting and data management [15]. This means that any time the user state or application state changes (which are handled on the server-side using sessions) or any new data is required, queries need to be sent to the server and these queries may take some time, which again means that the user is sitting and waiting for the application to refresh and show the new webpage.

Business data can be stored on the server using database software. The database software stores application data to the disk on the server it is running on [25]. The most common type of database used is a relational database which stores data in tables.

2.4.2 Native application

Native applications are those executable applications that need to be installed first[15]. They run on a specific platform or a device [26]. It can use device-specific hardware and operating system. As seen in the Figure 2.2[15], the application is created as a single piece of software that contains all the logic. One of the main problems of this is the dependency on the operating system. With all the different OS options for eg. Windows, UNIX, Android etc, to reach more clients, different versions of the same application would have to be created which would work on these different operating systems. But one advantage that native applications have over the traditional web application is that native applications can keep their state by using local databases, this allows for a richer user experience because accessing local resources are faster [15].

An example of a native application is the game 'Pokémon Go'. It first has to be installed on the device and then it efficiently accesses system functionalities like GPS for mapping locations, the camera for augmented reality, and the accelerometer to measure acceleration to give the user the best experience possible [26].

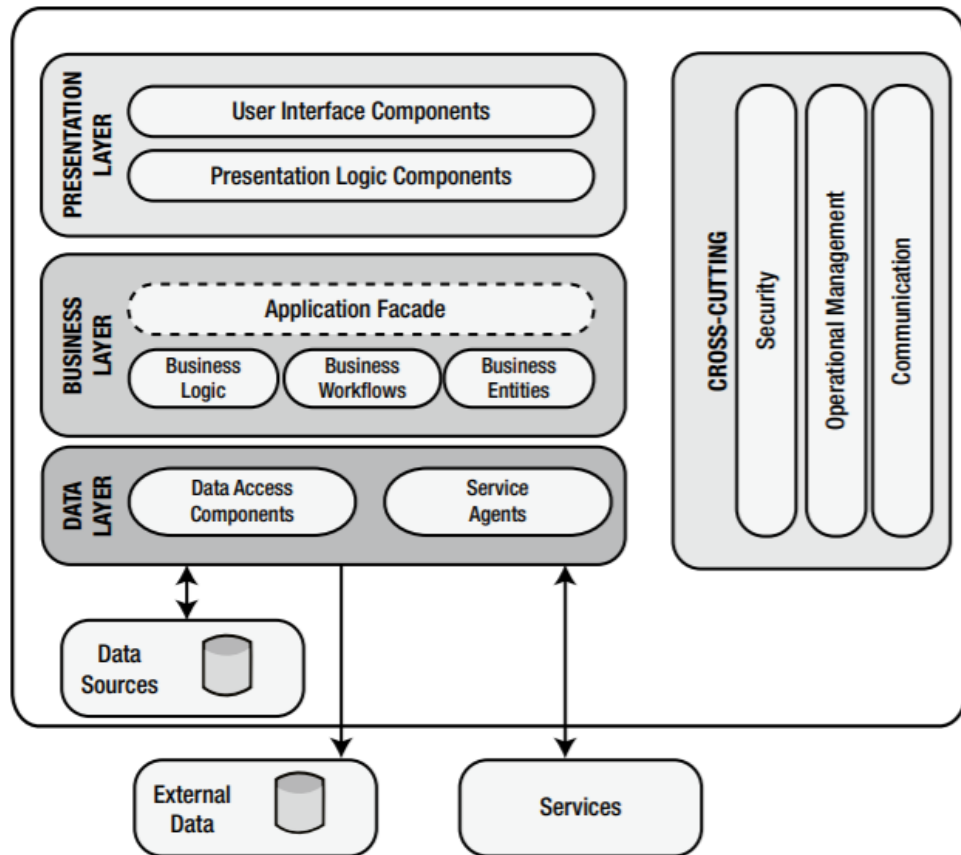


Figure 2.2: Native applications [15]

2.4.3 Single page application

A single page application (SPA) is a web application that does not require page reload during different clicks of the user. It uses a single HTML web page as a shell for all the application’s web pages [15]. *Gil Fink* and *Ido Flatow* explain that “SPAs resemble native applications in their behavior and development, but they run inside a browser process as opposed to native applications, which run in their own process” [15]. Most of the development happens on the front end, unlike the traditional web apps which are heavily dependent on the server.

The main SPA building blocks include

1. JavaScript Libraries and Frameworks: Choosing the right JavaScript library is

important to give the user the best possible experience while using the application. Using libraries that are used by well known sites, is a good indicator.

2. Routing: In an SPA, since there is only one page, steering from one view to another, where the different views/pages may have a different look and feel when compared to the starting page, would need to be handled differently than in the traditional web applications. Many questions like 'How do you do deep linking?' or 'How can search engine optimization be done?' may seem complicated to answer for SPAs[15]. But most of this can be handled by the HTML5 History API.
3. APIs: There are many APIs that HTML5 provides that can be leveraged to implement SPAs. For example, the regular communication between the SPA and servers can be accomplished using Ajax and `XMLHttpRequest` object. The HTML5 connectivity APIs, add new ways to enable better web server communications. This API includes WebSockets, Server-Sent Events and CORS(cross-origin resource sharing). Another example of an interesting API is Web Workers. They help in improving the performance of these applications. Web Workers are a simple means for web content to run scripts in background threads.[27] These background tasks run by the worker don't affect the user interface.
4. Client-Side Template Engine: In a Single Page Application there are certain portions of the page that will have to be re-rendered every-time due to user interaction[15]. Client side template engines can help here. They help in creating a more maintainable code by separating the view mark up from the view logic. Some examples of template engines are libraries like Underscore.js and Handlebars. Underscore.js gives you access to its APIs by using the underscore sign[15].

5. Server back end API and REST: Even though the SPAs rely heavily on the logic being on the client side, it doesn't mean that servers aren't required but instead the web server's role is changed[15]. Here, the web server helps by delivering the web page to the client and makes the relevant resources that the client needs, like templates available to it[15]. An added function of the web server is to make the web API available for the SPA, to do server functions like authentication, authorization, back-end database manipulation etc [15]. These web APIs expose endpoints to enable create, read update and delete operations and they mostly align to the REST architecture style. REST (Representational State Transfer) is an architectural style that describes the constraints that need to put in place while creating Web services.

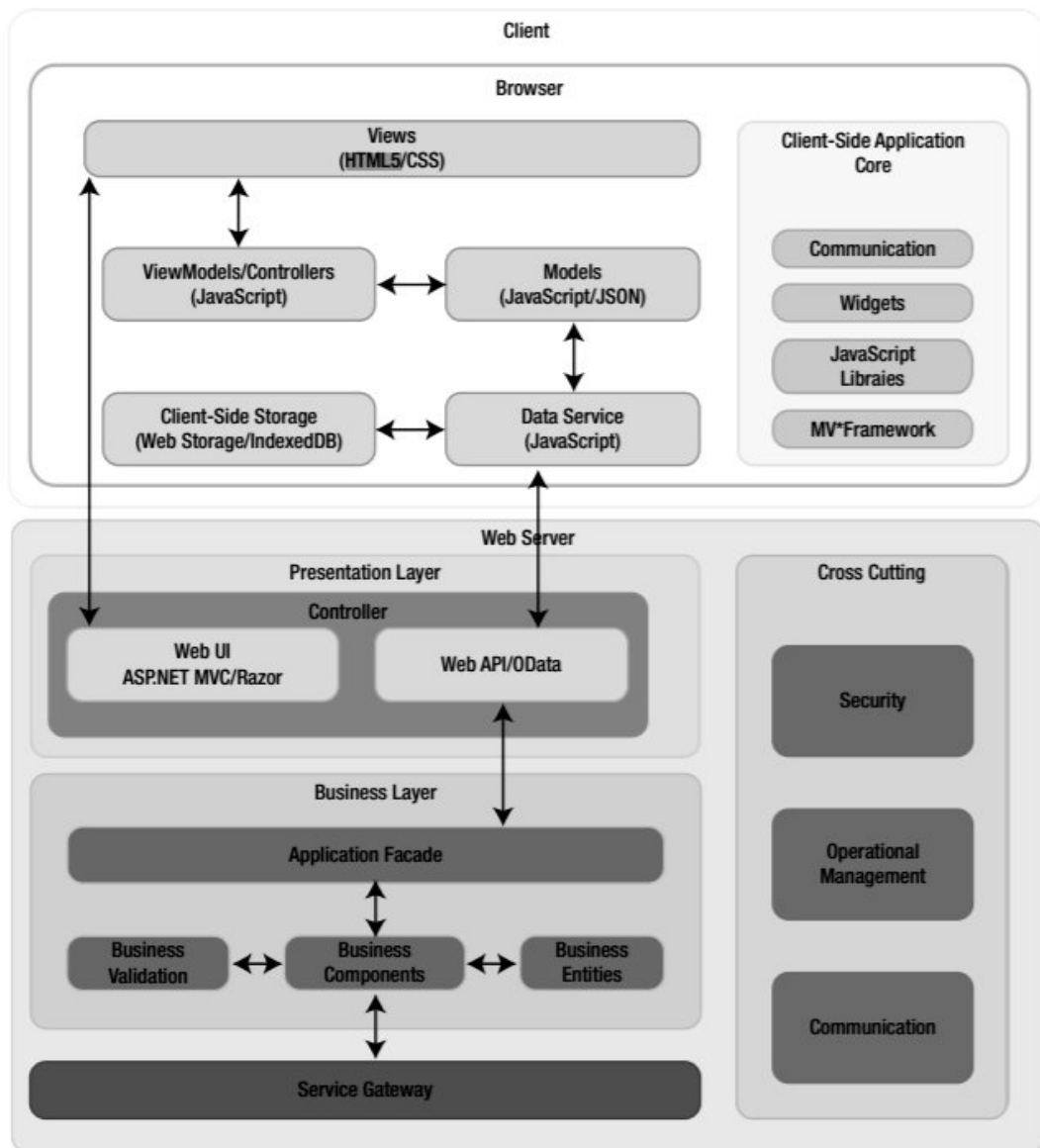


Figure 2.3: Front-end and back-end architecture of a single page application [15]

3 Case Study Company

3.1 Enkora Oy

Enkora Oy is a SaaS (Software as a Service) provider located in Helsinki. The company is a leading provider of modern customer service solutions in Finland. They operate by providing software and hardware solutions to their clients. The software and hardware products created, help their customers sell their services, accept payments, control access, view reports, manage reservations of resources, and allow self-service for end-users. Some of the hardware solutions provided by Enkora are smart lockers, RFID devices, turnstiles, self-service kiosks etc. The solutions they provide can be broadly split into Customer Flow Solutions and Market Flow Solutions as shown in Figure 3.1.

Few examples of Customer Flow Solutions can be seen in the Wellness/Fitness industry with the use of POS (point of sale) devices, access control devices, reservation systems etc. These systems are used everyday by customers and employees of the client company to purchase courses, add agreements, make subscriptions etc.

The workplace solution implementations can be seen in the construction industry, more specifically at the construction yards. Examples of services and resources used there are access control devices and software, time and attendance applications etc. They help to keep track of their employees who are working at a particular worksite and helps them keep track of the work hours.

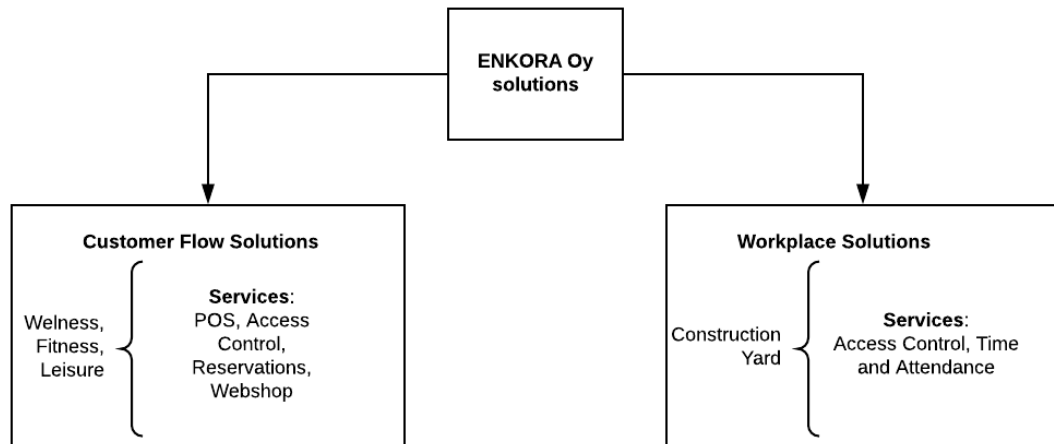


Figure 3.1: Enkora Oy Solutions

One of Enkora’s product concept is such that they have a basic product which can be used off the shelf to produce a working webshop and reservation system. This can also be tweaked/customized according to the needs of the client and their business model. More information about their reservation system is provided in the next chapter.

3.2 The Software Product

Enkora’s codebase is nearly 14 years old. One of Enkora’s more popular software products is the customizable online reservation system. The reservation system can be configured for reserving courses at a particular place, or with respect to resources like badminton courts or for reserving a one time session, like, a massage. Screenshots shown in Figures 3.2, 3.3 and 3.4 represent two different clients using the same Enkora product, but customized according to their needs.

The clients have the option of choosing either the default, which is Enkora’s general style template, for their version of the web reservation system, or else they may prefer having a customized look, which is saved in a custom CSS file.

Each service provided by the client company (as seen in Figure 3.3) may have a different business logic when it comes to creating a course/lesson and making a reservation for those lessons.

The online reservation system also allows the client’s customers to create their own accounts, which allows them to make reservations in their name and view these reservations later on. This is particularly useful when the clients want to offer discounts to their members or customers who have an account with them. But some clients also allow the usage of “guest” accounts, where the customer is not required to create an account to make a reservation.

There is another side to this reservation system and that is - the configuration portal. This is mainly used by the employees of the client company. This portal allows registered employees to create courses/lessons which will be visible for booking in the reservation portal. It can also be used (by the employees), for helping customers find information about their reservations, or to create reservations for the customer. The configuration portal provides the client with a wide range of tools to allow them to modify the reservation system to their needs. For example, they have a tool in the configuration portal which allows creating and editing translations, for a particular phrase or word, which is used in the reservation system. This allows the reservation system to be used in multiple languages. There is also a tool which allows the setting of parameters which can control, the visibility and functionality of different features in the reservation system.

3.3 Current state of code

3.3.1 Enkora’s codebase

Due to Enkora’s software-product providing a large number of features, the current code base is quite large. It uses languages like PHP and JavaScript and different

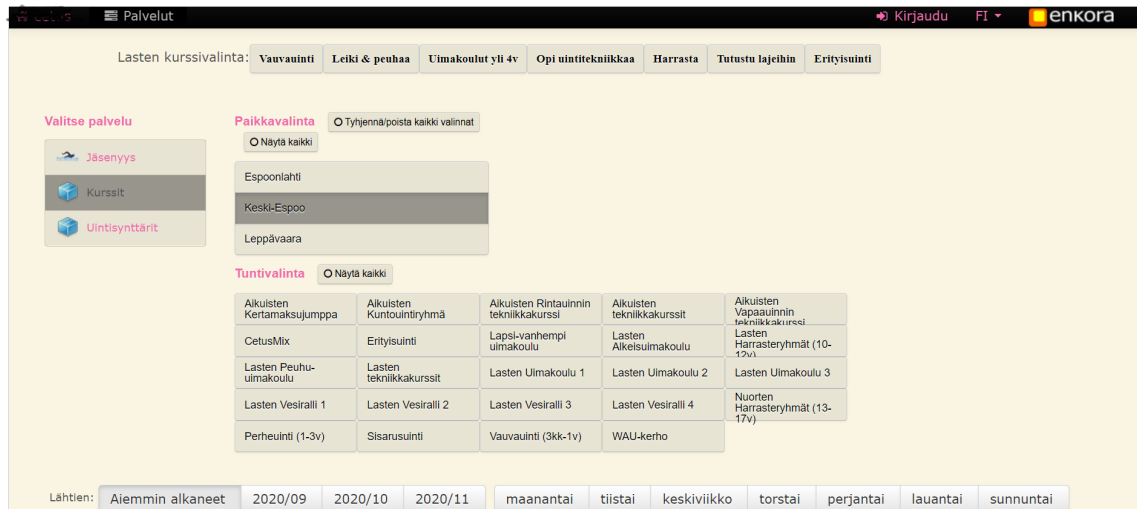


Figure 3.2: Online Reservation System

frameworks like Bootstrap, Backbone and Angular and also multiple libraries like jQuery, Underscore.js etc. to bring about a working final product. The frontend is implemented using TypeScript, CoffeeScript and regular JavaScript.

3.3.2 Why the need for a change

Challenges with the current code are mostly realized when customization is required or new features are to be implemented. In the current code base, some of the features are implemented using TypeScript and Angular, and some of the older features are implemented using CoffeeScript and Backbone. The aim of this thesis is to migrate code from CoffeeScript to TypeScript and study the advantages and disadvantages (if any) of converting some parts of the Backbone/CoffeeScript code to Angular/TypeScript.

Implementing this change would bring about a more uniform code base across features, making it easier to understand and customize code periodically. The other effects of this change are also to be studied. More details are provided in Chapter 5.

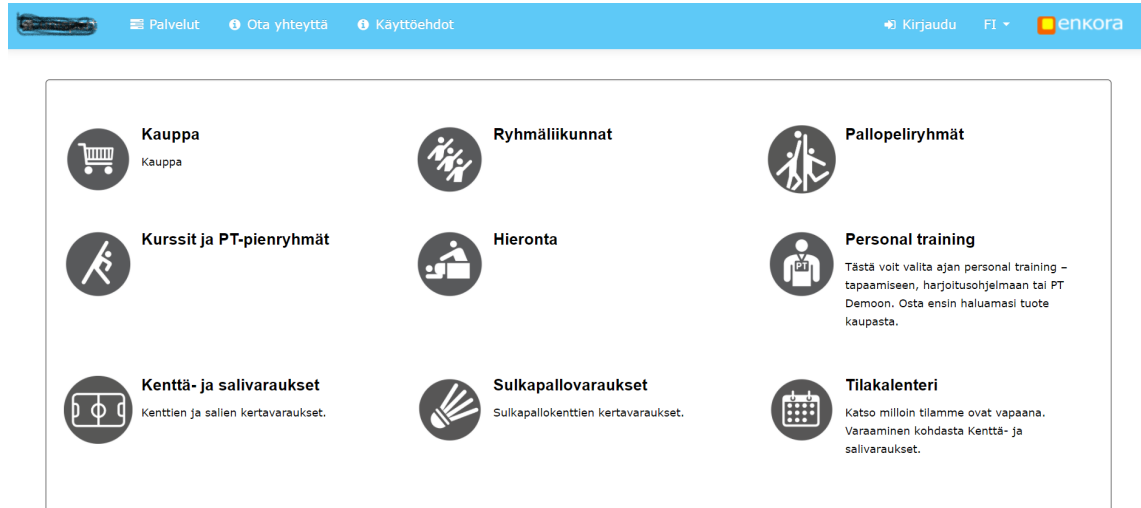


Figure 3.3: Online Reservation System 2a

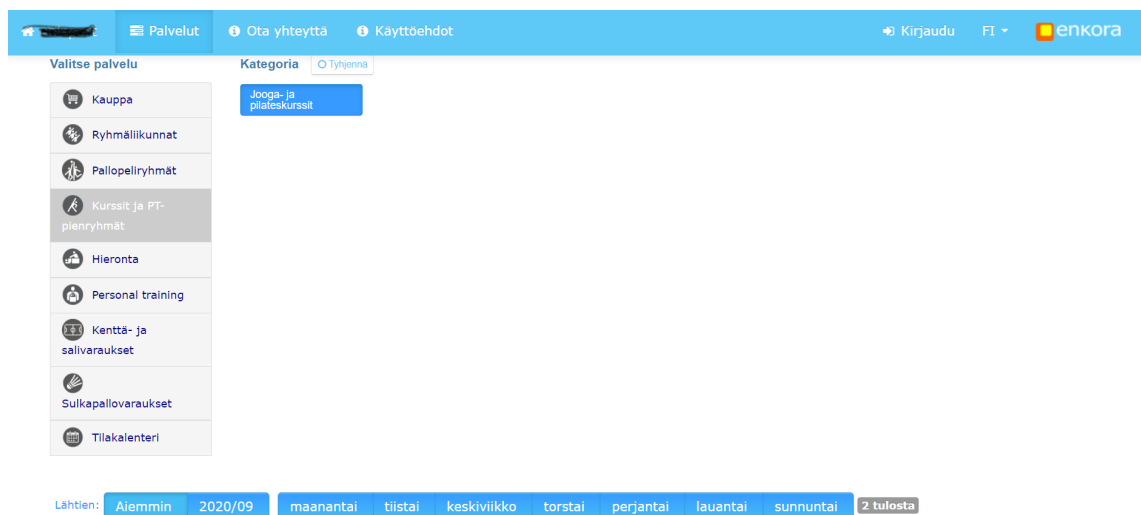


Figure 3.4: Online Reservation System 2b

4 Legacy Systems, Technical Debt and Modernization Techniques

In this chapter we discuss in detail about Legacy Systems and Technical debt. This gives us a background understanding of what kind of issues can be seen in the Enkora project. The modernization techniques discussed later, provide ideas on what kind of solution plan can be taken to tackle these problems.

4.1 Overview of Legacy systems

Different researchers have different descriptions of what a legacy system is. For example *Bakar and Razali*[28] describe a legacy information system as “an ‘old’ information system that remains in use in an organization. These systems have been developed in the past, and are critical to the business operations but are difficult and expensive to maintain”[28].

In their paper, *Khadka and Jansen*[29] describe a legacy system as “a business critical software system that significantly resists modification and whose failure can have a serious impact on the business”[29].

The definition of a legacy system that *Crotty and Horrocks* described in their paper, also applies to this paper. They described it as “a system that is business critical and demonstrates one or more of the following additional characteristics: old age, obsolete languages, poor if any documentation, inadequate data management, a

degraded structure, limited support capability and capacity, increasing maintenance costs, and lacking the necessary architecture to evolve”[30].

From the interviews conducted by *Khadka and Jansen*[29] we know, that the usual reasons the Legacy Information Systems are kept for as long as they are, are because they are usually business critical, i.e. they have been tested and have been working in production for so many years and hence proven to be reliable systems.

With the world moving towards the next phase of the industrial revolution, where “smart” is used to illustrate the intelligence in the smart products, smart facility etc[31]. Organizations will need to use state-of-the-art technology if they want to be part of the revolution and reap the benefits. This is one of the reasons that legacy systems may need to be modernized.

The other reasons for change are: [29]

To remain agile to change : As mentioned earlier customers would benefit from the system being flexible and using the latest technology.

High maintenance cost : High maintenance cost goes hand in hand with legacy systems. This is one of the most common motivations to modernize an old system.

Lack of knowledge : Sometimes it is the difficulty in finding personnel who know how to work with the legacy technology.

Prone to failures : Even though these systems have been running in production for ages, and are seen as reliable, they can also be prone to failures after a while, for example when the legacy system environment runs out of support.

To reduce technical debt : This can be defined as the technical compromises made during the software life cycle knowingly or unknowingly to yield short term benefits. We will discuss this further below[32].

4.2 Technical Debt

In most software companies, and in any software that is expected to run for a long time and cater to multiple clients, technical debt is an aspect that cannot be escaped. As mentioned earlier technical debt can be defined as the technical compromises made knowingly or unknowingly, that can yield short term benefits but cause bigger problems to the project/software in the long run[32]. In an IEEE software engineering article, author Sven Johnson describes Technical Debt as 'not-quite-right' code and that building on top of such a codebase would be expensive later on.

Technical Debt is analogous to Financial debt[33]. This can be explained with an example. It is common practise now for a person to take a loan from a bank, but when he does so, he incurs a debt. The incurred debt in itself is not a bad thing, as long as the installments are paid off regularly. But when installments are not paid off at the right time, then interest on the loan starts adding up, and later this amount becomes so large in comparison to the loan taken, that, it can lead a person to bankruptcy.

In a similar context, in software development, when a feature needs to be pushed into production soon or a critical bug is hampering the use of an application, a developer may choose to opt for a quick fix or a hack and later improve the code, so as to reduce the inconvenience to the customers. If the code is updated regularly, then the debt can be written off soon. But when the developer forgets about this incurred debt, this keeps increasing over time with each change to the software making it more expensive to pay off. In their book *Girish Suryanarayana et al.* describes a situation where “the acquired technical debt is so huge that it cannot be paid off anymore and the product has to be abandoned. Such a situation is called technical bankruptcy”[33].

In 1992, Ward Cunningham was the first to draw a comparison between technical

complexity and debt, when he said

“Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object- oriented or otherwise”

[34].

When developers are overburdened with development tasks, or they lack experience in applying design principles or awareness of design smells and refactoring methods, the outcome is poor quality code. Low quality code is an indicator of technical debt[33].

Technical Debt can be incurred from multiple sources. Below is a list of debts that are loosely based on the sources of debt.

Requirements TD - This technical debt develops when, requirement prioritization decisions are made, which in turn creates a product which is either not necessary or does not meet the need of the customer[35].

Architectural TD - Internal quality of a software is linked to its ability to be maintained and scaled. But when architectural decisions hamper these abilities we incur this debt.

Design TD - These are a result of shortcuts taken during the design phase

Code TD - Below par code that violates best coding practices or coding rules, like avoiding copy-paste of code which may lead to code duplication.

Test TD - Shortcuts during the testing phase, like not including acceptance tests or unit tests can lead to Test TD.

Documentation TD - Lack of relevant code comments and incomplete documentation of the code base, can leave developers struggling on similar issues and spending time on issues which have been solved by other developers and thus collecting debt.

Self Admitted Technical Debt [36] - Most of the above debts would require someone who can review the code and decipher what kind of debt is being created. Then there is a type of technical debt that does not need to be deciphered, since while writing the code, the developer has added comments such as 'todo', or 'fixme' or something more verbose like 'Modify this block, currently works, but it's a hack!'

When understanding Technical Debt (TD), it is also important to understand what things are not TD. Some examples of non-TD are defects, features that have not been implemented yet, lack of helpful processes, unfinished tasks in the development process, trivial code quality issues, and low external quality[32].

4.3 Modernization strategies

Even though Legacy systems cause technical difficulties, they are an important asset to the organization[31]. These systems cannot be fully eliminated because they contain essential business information and any failure caused by the systems will have serious consequences in running daily business tasks[31]. So the organization using such a legacy system, constantly finds itself battling the technical issues that arise, since the uninterrupted running of the system is more important.

Software Modernization can be defined as the process to re-develop existing legacy software by developing, or migrating software modules and libraries, when

new features can no longer be viably developed[37].

Modernization of a Legacy system can propose plenty of challenges, based on the size and complexity of the system. Some strategies for software modernization discussed by Khadka[37] are:

Replacement strategy : In this strategy a legacy system is retired or replaced with a more up to-date Commercial off the shelf package(COTS). In this case, either none or a very small part of the former system is continued to be in use. Two significant risks of the replacement strategy as mentioned by Almonaies et al.[38] are: the maintenance of the new system, which will not be as familiar as the old system; and the lack of a guarantee that the new system will be as functional as the original.

Wrapping is a popular modernization strategy that allows the possibility of encapsulating existing legacy software for reuse in a new target architecture[37]. Wrapping is a quick strategy that can be used when the legacy system has a high business value and is most suitable for smaller programs since identifying and exposing business functions can be time-consuming[38]. However, wrapping does not reduce maintenance cost, but rather increases it as the enterprise has to maintain the interface (wrapper) layer as well[37]. Almonaies et al.[38] mentions that the main problem with this strategy is that it does not change the fundamental characteristics of the legacy applications that are being integrated. Wrapping will not solve problems already present, such as problems in maintenance and upgrading [38].

Redevelopment proposes the redevelopment of legacy system functionalities. Almonaies et al.[38] described redevelopment as a reengineering approach instead, where the application is first studied and then adjusted, so that it can be represented in a new form. To reach this new form, activities like redesign-

ing, restructuring and re-implementing software can be done[38]. However, the possibility of failure is usually quite high for organizations to seriously consider this approach and it may also require a significant amount of investment[37]. In this strategy, the existing system/assets may not be reused or if reused, it will be very sparingly.

Migrations are usually done when a system needs to be run in a different environment during a system's life[39]. Almonaies et al. use the term migration when referring to any approach which moves the entire legacy system and its core framework to the new environment[38]. The migration strategy tends to be costly and time-consuming compared to other strategies. However, a migration strategy gradually allows to internally restructure, reuse and modify the legacy systems into a new target system. Thereby, potentially reducing maintenance costs associated with legacy systems in the long run[37].

4.4 Modernization methods

Based on the above strategies many researchers have devised modernization methods. A few are discussed below.

4.4.1 Chicken Little and Cold Turkey Methodologies

A modernization method discussed by *Brodie and Stonebaker*[40] during their pioneering *DARWIN* project, was based on the *Chicken Little strategy*, which concerns migrating the legacy software, by small incremental steps, until the final desired objective is reached. Every step necessitates a smaller investment, a shorter time, and produces a quantifiable result[40].

To understand why the Chicken Little strategy was adopted, they compared it to the *Cold Turkey* strategy and discussed its benefits. The Cold Turkey strat-

egy involves re-writing the entire legacy system from scratch to produce the target information system using modern software techniques and hardware of the target environment[40]. But there are many risks involved with the Cold Turkey strategy which *Brodie and Stonebaker*[40] cite as follows:

A better system must be promised : In a redevelopment process, if the only benefit to such a big expenditure, is the promise of lower maintenance cost in the future, then it may almost seem an unnecessary expense to the management paying for this endeavour, unless new features were also added during this process. Thereby, increasing the risk of failure[40].

Business conditions never stand still : Usually migration of large information systems take years to reach completion. While the legacy IS rewrite proceeds, the original legacy IS evolves in response to maintenance and urgent business requirements, and by midnight functions (i.e., features installed by programmers in their spare time). It is a significant problem to evolve the developing replacement system in step with the evolving legacy system[40].

Specifications rarely exist : Many times the only documentation for old legacy systems is the code itself, and in such cases the exact purpose of many aspects of the legacy system would have to be decrypted from the code, which adds to the complexity of the whole replacement process.

Undocumented dependencies frequently exist : As the legacy information system has been operational for a long time, other systems grow dependent on the legacy information system and unexpected dependencies cause additional complexity to the redevelopment process[41].

Legacy information systems can be too big to cut over : Most Legacy information system have so much data that it would need a long time to transfer

the data to the new environment. And business may not be able to survive the downtime that this transfer would require.

Lateness is seldom tolerated : All the problems cited above can cause the project to be delayed and this may in turn lead to its termination.

Large projects tend to bloat : There is a tendency for large projects to become bloated with nonessential groups or people[40]. These groups may be a part of an exploration strategy that the organization wants to try in their project. But this increases the cost of the project and may in turn lead to its termination.

[40]

The Chicken Little legacy system migration involves iterative selection and migration of parts of the legacy system to become new parts of the iterative target system[40]. During the migration, the legacy system and the target system form a composite system which run in parallel to collectively provide the mission-critical system function[40]. The iterative nature of the Chicken Little strategy provides two ways to reduce risk.

- First, a fallback position should be set, in case a step fails.
- Second, the functionalities selected for a step should be minimal, so that the effective risk is zero.

[40]

With the completion of the Darwin project, Brodie and Stonebraker proposed a 11 step generic migration strategy, where each step is an incremental one.

1. Analyze the legacy information system
2. Decompose the legacy information system structure
3. Design the target interfaces

In the beginning, the new system will be small, but when the migration process advances, the growth of the target system will continue progress until it has call the functionalities of the legacy system. This inter-operate-ability is provided by a module known, in general, as a *gateway*, “a software module introduced between operation software components to mediate between them”[42]. A gateway can be used at any level. If used between an application and the DBMS, then it is called a *database gateway*. If used between the interface and the rest of the system then it is called an *interface gateway*. Then a 3rd type of gateway is the *IS gateway*, this is placed at a higher level, it encapsulates the entire legacy system.

A pictorial representation of the Chicken little method using two alternatives of database gateway is shown in Figure 4.1. In the figure there are 2 gateways being used. The *forward gateway* is designed so that it constitutes a translator that receives and transforms database service calls from legacy applications into calls to the target DBMS on the server machine(s)[43]. The *reverse gateway* contains a decoder that receives and transforms calls to the modern DBMS from the new applications and maps them into calls to the legacy database service[43]. In the Figure 4.1 there is also a *coordinator* being used. The coordinator maps calls from the legacy and target applications to any of the following, legacy database, the target database, the reverse gateway or the forward gateway[43].

4.4.2 Renaissance

Battaglia, M. et al[44] discuss a step by step methodology which they called *RENAISSANCE*. The overall process can be seen in Figure 4.2.

There are 4 main phases to this methodology. They are:

1. Evolution Planning: Here the '*what to do*' is decided. In this method, it is important to reduce the cost of assessment, by reducing the scope of assessment to only the required components that can benefit from reengineering[44]. First

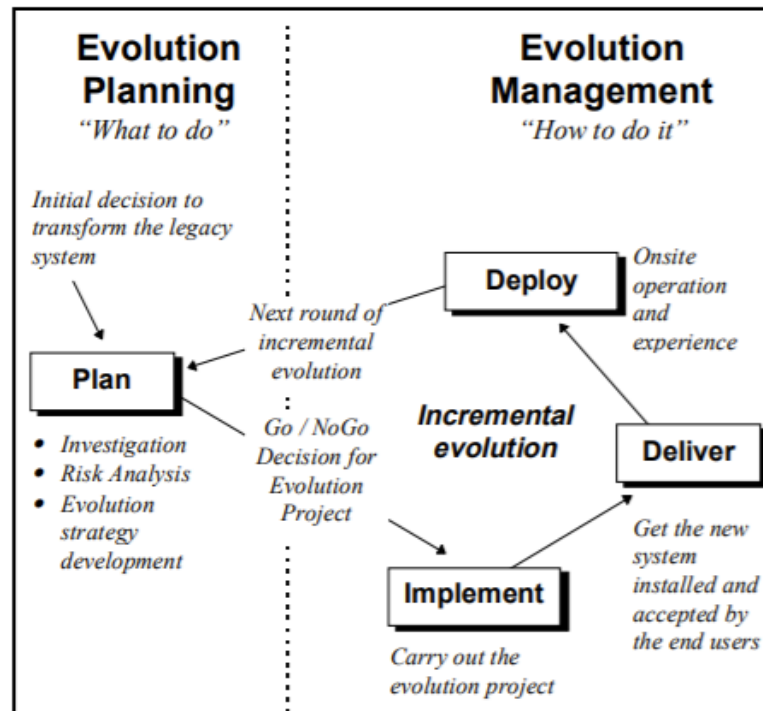


Figure 4.2: Renaissance overview [44]

decisions need to be taken at what level the legacy system will be assessed, e.g. will it be a quick top level assessment or will it be a detailed assessment of a certain component of the system[44]. Once it is decided which parts of the system will be assessed, the focus is limited further, to only those components that will benefit the evolution.

As seen in Figure 4.3 the candidates are assessed further based on their technical quality and their value to business. Components that have high business value, but contain lower technical quality, are good candidates for evolution[44].

2. Evolution Implementation. Implementation is different when compared to a traditional information system project because there is already a system in place(the legacy system). To begin the implementation, a detailed structure and behaviour of the system would have to be deciphered, and this may be

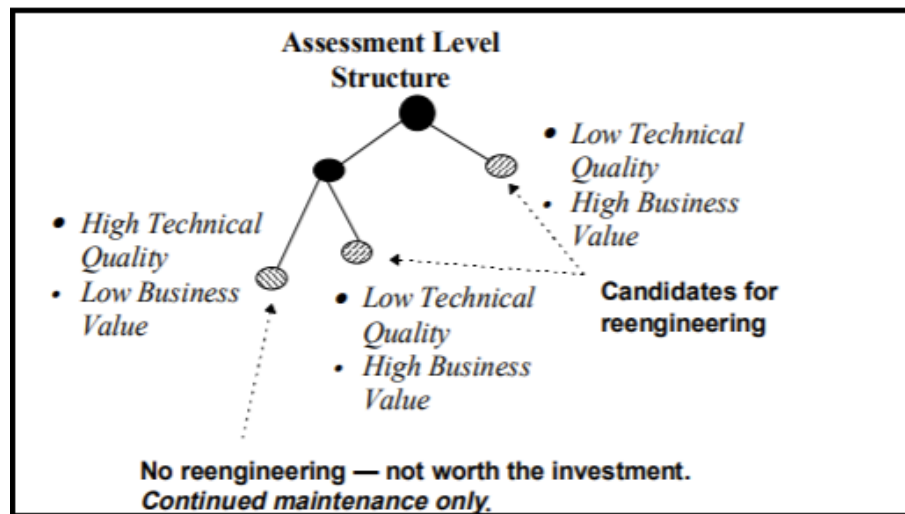


Figure 4.3: Finding candidates for evolution [44]

difficult in a legacy system, as the information may be scattered[44]. Renaissance provides an extensive group of procedures for technical modelling, thereby creating multiple views of the old and new system, using the Unified Modelling Language[44]. Special techniques are elaborated that can be used in evolution projects, for example “cohesive batches” of components are identified that can be migrated together to the new system in an orderly and incremental fashion[44]. Continuous testing would have to be done and accordingly the component batches are integrated[44]. The result of this controlled implementation process is a transformed and fully tested system, ready for final installation and acceptance testing[44].

3. Delivery and Deployment. These are the last two phases of this method. In the delivery phase, the new system is given to the final users, who migrate the enterprise data to the new system[44].

Firstly, install the new system on the actual hardware on which it will operate, with the actual software packages, etc[44]. Then, is the *changeover design* i.e. deciding between operations change over from the legacy to the new

system incrementally, or in one single step? This task is often ignored in other reengineering methods, but not in the Renaissance method[44].

4.4.3 The Butterfly Methodology

The Butterfly Methodology was developed as part of the MILESTONE project, involving Trinity College Dublin, Broadcom Eireann Research, Telecom Eireann, and Ericsson in 1996[42]. In this method, the premise is that, the information within the legacy system is the crucial part of the system and from the target system's development aspect, it is not the dynamic legacy information that is important, but instead its schema[42]. The Butterfly Methodology divides the target system development from the data migration phases, hence, removing the need for gateways. When comparing the Chicken Little approach and the Butterfly approach, one noticeable difference is that the latter is designed so that, at the same time both the legacy and target system cannot be accessed[41]. The data is the last aspect that is migrated, so as long as the migration has not been completed, the data will be stored in the legacy system[41].

The 6 phases of this methodology as described in the paper described by Bing Wu et al.[42] are:

Phase 0: Prepare for the migration

Phase 1: Understand the semantics of the legacy system and develop the target data schema(s).

Phase 2: Build up a Sample Datastore, based upon the Target Sample Data, in the target system.

Phase 3: Incrementally migrate all the components (except for data) of the legacy system to the target architecture.

Phase 4: Gradually migrate the legacy data into the target system and train users in target system.

Phase 5: Cut-over to the completed target system.

[42]

In Phases 0 to 2, Wu et al. starts by understanding the user requirements and what the benchmarks would be for determining if a migration is a success or not. Then they try to understand the legacy interfaces, legacy applications and the legacy data. In phase 2 a Sample dataStore is built, which will be used to develop and test the target system. [42]

In Phase 3, using the Sample dataStore built in phase 2, a 'design-develop-test' approach will be used. This phase is mainly for the target system development. The legacy interface and legacy application will be migrated or developed partially and then tested using the Sample dataStore and then Validated against the User's requirements[42].

In Phase 4, the migration of data is the most important aspect. To facilitate this gradual data migration, The Butterfly Methodology proposes the following concepts:

Data Access Allocator (DAA): The Data Access Allocator, redirects all manipulations on the legacy data[42].

Legacy Data Store: When the legacy system requests data that is unaltered from this point forward, the DAA will direct the request to the legacy data store[41].

TempStore :The results of the data manipulations are stored in the latest TempStore by the DAA[42] These are auxiliary dataStores.

Data-Transformer (DT): This is employed to migrate the legacy data to the target system[42]. It is responsible for transforming the data from the legacy format to the target system format. This will depend on the legacy and target schemas [42].

Termination Condition (TC): The Termination Condition helps to determine if the data migration has reached the final stage[42]. These iterations will continue until the TC is complete and the legacy system can be closed[41]. After this, the final cut-over can be done. Based on *Sami Peräsaari's* paper we know “A *Threshold Value* (TV) is the maximum admissible amount of data of the final TS. TV is derived from the maximum time that the legacy information system can be shut down without a significant impact to the business. Thus, if $\text{size}(\text{TS}_n) \leq \text{TV}$, then the n 'th iteration is the final iteration of the data migration”. [41].

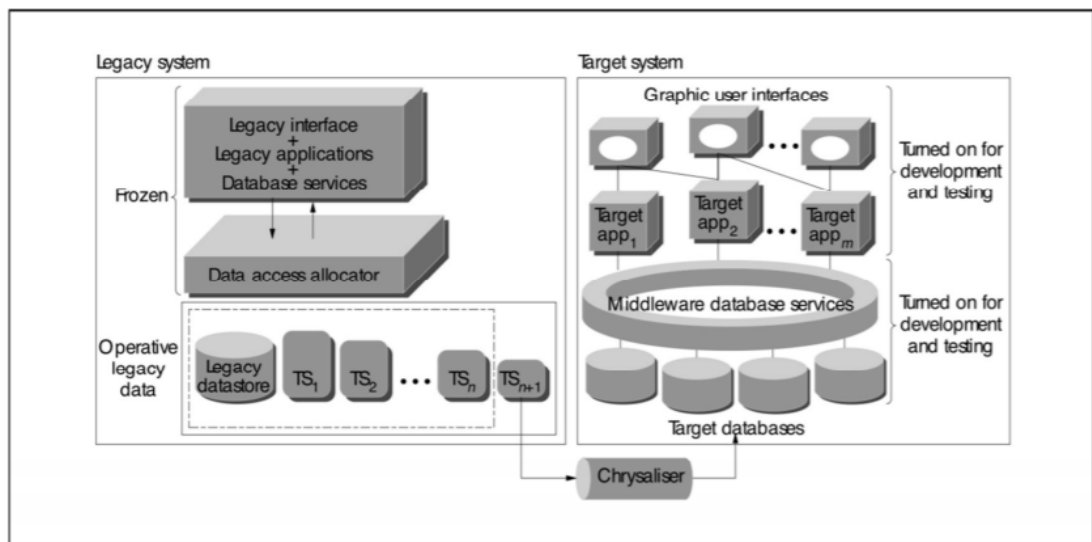


Figure 4.4: Butterfly methodology [41]

In Phase 5, since the target system is now ready and the interfaces, applications and data has been transferred, the new system is ready to run. Based on his research, in his own thesis, *Sami Peräsaari* states that “the original data and its modifications would still be found from the legacy datastore and the TSs”[41]. This is helpful if a rollback is necessary. Figure 4.4 shows an overview of the concepts and components of the Butterfly methodology.

5 Implementation Plan

Any software company that hopes to keep up with the demands of its customers and have their product labeled *smart* needs to be agile to change. Moreover, every company desires to have easily maintainable and well documented software product. This is what Enkora is striving towards.

Both frameworks (Backbone and Angular) have their own advantages and this need for change is not to showcase any one framework's advantage over the other. It is the project/software that dictates what kind of framework would suit it.

In this chapter we cover the modernization plan that is suggested for Enkora's reservation system. So in Section 5.1 Enkora's reservation system is discussed further. In Section 5.2 the problems with the current code base are explained and in Section 5.3 a step by step plan is suggested which is used to implement the code change.

5.1 Enkora's Reservation System

14 years ago when Enkora started their codebase, they used a combination of technologies and around 7 years later they decided to continue the frontend development using CoffeeScript. This was because CoffeeScript was quite a popular language at the time and its syntax style which is similar to Python and Ruby also provided certain features which were not available in JavaScript at the time. This made it more attractive for development purposes. CoffeeScript and its advantages served

its purpose for a long time. Later, with ES6/ECMAScript6 providing many of the same coding features as CoffeeScript, like arrow functions, classes, inheritance, using “let” to define variables etc, there was nothing special being brought to the table anymore, by using CoffeeScript. And now with the project growing larger, the combination of CoffeeScript and Backbone did not seem to work for the project.

First a brief examination is done of how Enkora’s codebase looks. Currently Enkora’s reservation system uses the Backbone framework in most of its implementation. The data which is to be used in the views are defined as models. Each type of data is defined as a unique model and is saved as a separate CoffeeScript file. For example every reservation is defined as a reservation model which is identified by a unique reservation-id and this structure is saved in a file named *reservation.coffee*. Similarly, each event, product etc. has a separate model defining it. In the codebase, these models are put together in a *models* folder.

A *collection* folder contains structures which can accommodate multiple models within itself. For example a collection *cart* is written in a file named *cart.coffee* and models of structure *cartitem* can be inserted into it.

Another folder named *views* contains all the files which will help in creating each view. They link the model and the HTML template into which the model’s data is inserted. Together they display the necessary view + information on the screen.

One problem that I noticed when trying to understand the system, is trying to pinpoint the model which provides the data to a particular view. Many models are initialized inside a view, since the data that needs to be displayed may be more spread out.

As mentioned previously in Chapter 3, Enkora’s reservation system has two parts to it. One is the portal used by the customers, where they can log into their own account and do one of the following actions:

- Edit their info

- View their reservations
- View their unpaid receipts
- Browse through the different services
- Reserve a course or reserve a timeslot for a service like massage or reserve a court/hall for an activity etc

The other is a portal that is used by the employees. Employees include anyone who has a valid username and password for this portal. This also includes the staff at Enkora, who can log into this portal and help when necessary. This portal is more of a tool where the following actions can be done. View and edit customer Info, create products like gift cards, multi-tickets etc. which can be purchased from the shop portal, create and modify lessons and courses, reserve a user to any of these lessons and courses, assign instructors to lessons and courses, assemble and view reports such as sales reports, receipt reports, event reports etc. More than 25 different subheadings are present in this portal, each of which can perform multiple different functions. Some of these subheadings include - Users, User Parameters, Fare Products, Reservations, Manage Reservations, Translations, Fields, Cache Reset, etc.

5.2 Problems in the current state of code

There are a few things that could be improved in the current codebase. This list was aggregated by interviewing the Senior Software Developer and the Chief Technical Officer at Enkora. My experience of working with this code for the past 2 years has also allowed in identifying some of the issues mentioned. The interviewed developer was the one who initially instigated the need for a framework change and had already started moving some of the modules to the new framework 2 years ago.

Some of the problems noticed are listed below,

1. Use of Templating engines

In a project when HTML files need to be linked in multiple places, Backbone uses a templating engine. In Enkora, for this purpose the Underscore[45] library's `_.template` method was used as a templating engine. It compiles JavaScript templates into functions that can be evaluated for rendering[45].

An example from the current code base is shown here. A piece of code is saved in an html file, named as `__participant__select.html`. This is required by 4 other files `__eventgroup__buy.html`, `__product__buy.html`, `event-page.html`, `resource-page.html`. In the Listing 6, the example of how this template is linked is shown. Linking complex html pages using the Backbone templates causes the loading of the page to get slower.

```
_.template(jQuery('#template__participant__select').html(), {  
  ↪ data: data })
```

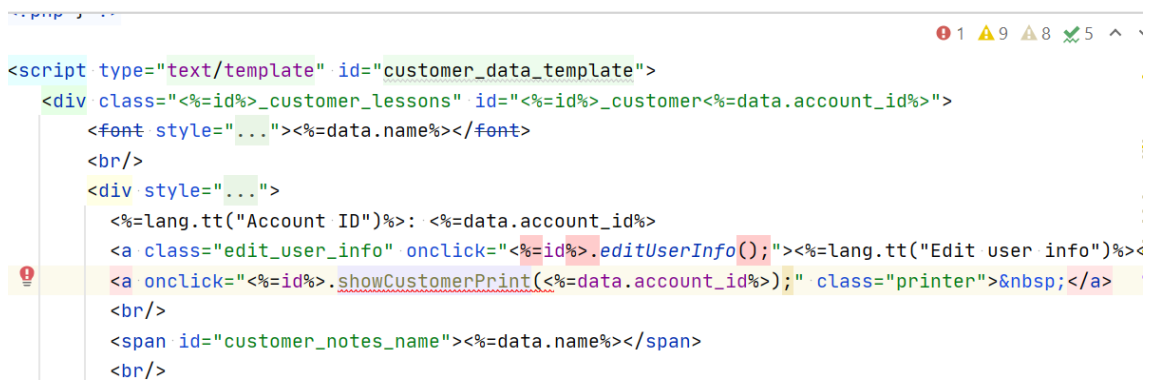
Listing 6: Template

2. Bugs and regressions

One problem that I have noticed quite often with the old code is, with every new feature created, there are regressions occurring, which are not caught during the testing phase. It is not easily detected which modules will reference the changed piece of code, so at times only one out of the multiple use cases are tested. Also, it is not easily visible from the code what kind of data is provided by the model. The old code has a mixture of PHP + HTML and Backbone + CoffeeScript.

3. Intellisense

Intellisense is a general term that references to a collection of features that allow the editing of code, in an IDE like parameter info, code completion and quick info[46]. Enkora has a lot of code that uses a combination of PHP, jQuery, JS and HTML in the same file. In such cases the IDE's intellisense cannot help in suggesting the right code to be written or in pointing out the errors in code. For example in Figure 5.1 (this snippet of code is taken from Enkora's codebase), the code runs without any errors, but the intellisense of the IDE highlights certain lines as errors which can be confusing and misleading for developers. The errors and warnings are shown in Figure 5.2.

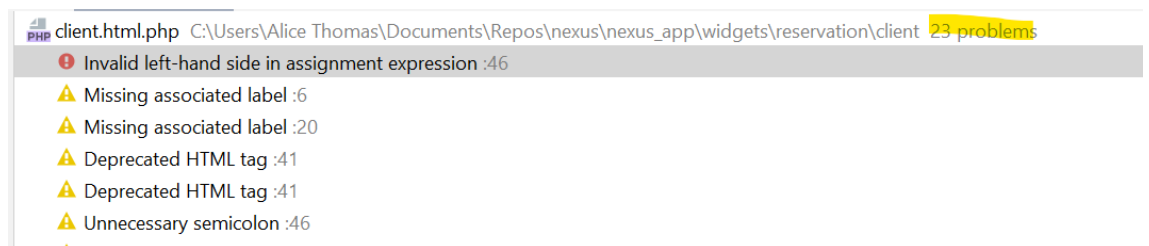


```

<script type="text/template" id="customer_data_template">
  <div class="<%=id%>_customer_lessons" id="<%=id%>_customer<%=data.account_id%>">
    <font style="..."><%=data.name%></font>
    <br/>
    <div style="...">
      <%=lang.tt("Account ID")%>: <%=data.account_id%>
      <a class="edit_user_info" onclick="<%=id%>.editUserInfo();"><%=lang.tt("Edit user info")%><
      <a onclick="<%=id%>.showCustomerPrint(<%=data.account_id%>);" class="printer">&nbsp;</a>
    <br/>
    <span id="customer_notes_name"><%=data.name%></span>
    <br/>
  </div>
</script>

```

Figure 5.1: Intellisense in IDE



```

client.html.php C:\Users\Alice Thomas\Documents\Repos\nexus\nexus_app\widgets\reservation\client 23 problems
  Invalid left-hand side in assignment expression :46
  Missing associated label :6
  Missing associated label :20
  Deprecated HTML tag :41
  Deprecated HTML tag :41
  Unnecessary semicolon :46

```

Figure 5.2: Intellisense errors

4. Use of outdated php library as CoffeeScript transpiler

CoffeeScript is not natively browser supported, so a transpiler is required. The current code uses a php library for transpiling its CoffeeScript code to JavaScript. This php library that is used as a transpiler is no longer supported

and is now being maintained by the developers at Enkora. This is quite a hassle, as they would rather be spending the time creating and maintaining their features.

5. Code Technical Debt

In the current code, in the combination of Backbone and CoffeeScript, there are many custom jQuery AJAX calls which go against the spirit of Backbone, which follows a REST based approach and handles POSTS in its REST interface.

6. Lines of code

In the current code there are a few files that when compiled give generated JavaScript files which contain up to 30,000 lines of code. This is too large a number and is not the best practise. This can also contribute to the Code Technical debt.

7. Test Technical Debt

The current bits of front-end code implemented in CoffeeScript do not have any unit tests associated with it. Initially, in an attempt to get a viable working product, few tests are only in place for the backend/server side code. The project would benefit from more frontend tests, so that errors can be isolated earlier before being visible to the users.

5.3 Modernization plan

Transitioning the code is done based on the modernization strategies and methods discussed in Chapter 4. A plan is created that best suits Enkora. The plan chosen is the *Chicken Little method*. This plan was chosen because, one necessity was that both the old system and the new target system should be able to run in parallel,

atleast for a while until it is sure that the new system can run without any problems and meets all the client requirements. This is made possible because the Chicken Little method uses gateways either at the database level or at higher levels to do the switching. This feature was not available in the Butterfly method, where the possibility of using gateways to switch between the old and new system is completely eliminated. The Renaissance method provided information at a very high level and so it did not provide enough information for practical implementation.

Details of the Chicken Little method are discussed in the Section 4.4.1. The Chicken Little method is not followed completely since certain steps are not required in the Enkora project. For example, in Section 4.4.1 you find steps that are dedicated for designing the target database(step 5) and migrating the legacy database(step 8) which is not required in this project, since no database change is required.

The steps to the plan are as follows.

1. *Step1: Iteratively analyze the legacy information system*

In this step, first capture all the functionalities that are currently in the module that is going to be changed. This is important during the redevelopment or wrapping process so no feature / functionality is lost in the transition.

2. *Step2: Iteratively decompose the legacy information system structure*

The dependencies between the modules must be studied, so that it is determined by changing the module, what and how other modules will be affected. It should be ensured that well-defined interfaces(if required) are present between the modules and between this module and the database service.

3. *Step 3: Iteratively design the target interfaces*

Writing each new module as a component in Angular also allows for creation of useful interfaces to the objects being used in these components. The interface changes will be made based on what features can be put into individual

components. This makes the code more readable and maintainable, even later on. These components encapsulate a small portion of the user interface, which can be reused through-out the project. This helps in fixing the maintainability issue mentioned previously.

4. *Step4: Iteratively design the target application*

The target application will be designed based on the Angular framework rules, but also the services previously provided by the module should be retained by the target system. The services should be structured into smaller components, so that they can be reused. These components should be named appropriately so that it is understood what function it does /service is provided by the component.

5. *Step5: Iteratively design the target database*

For Enkora, no database change will be done. So the target database will be the same as the current database.

6. *Step6: Iteratively install the target environment*

The target environment is one where the Angular framework is installed. In Enkora's case, the target environment was already set up since the code transition was started some time ago.

7. *Step7: Iteratively create and install the necessary gateways*

The gateway created here is more of an application/interface gateway (since no change is happening in the database level, there is no need for a database gateway). A parameter is created and set to *true*, if the target module is ready for use by the clients. This parameter will direct the clients to the new environment for use. In case any issues are noticed, the clients can easily be directed back to the legacy module usage. Initially both the legacy system

and target system will have to run side by side until a few iterations are run and it is clear that no service was lost during the transition and all interfaces work as they should.

8. *Step8: Iteratively migrate the legacy databases*

There is no need for this step to be done at Enkora, since no database changes are made.

9. *Step9: Iteratively migrate the legacy applications*

The module of the legacy application which is to be migrated is selected and then re-written in TypeScript, following the rules of the Angular framework. At this point an extra step taken is to make sure that unit tests are written for each new Angular component created. This will help in clearing some of the Test based technical debt that the legacy code had acquired.

10. *Step10: Iteratively migrate the legacy interfaces*

By doing the previous step, all the user interfaces of the legacy module will be migrated. But care has to be taken to also make sure that the interface between modules is also handled. Since no more CoffeeScript is being used in these modules, the dependency on the outdated php library which transpiles code from CoffeeScript to JavaScript is reduced.

11. *Step11: Iteratively cut over to the target information system*

If the target system works with no issues, then after a set period of time, the legacy module can be retired and the parameter which served as a gateway to switch between the two can be deleted as well.

The above process is then repeated for every module/feature that needs to be modernized until the whole legacy system is moved to the target system. Initially

the modules which have interfaces with the chosen module will be chosen for the transition.

6 Evaluating the plan

By implementing the modernization plan suggested in Section 5.3, some of the issues noticed in the legacy code, (mentioned in Section 5.2), were handled and thus solved.

In this chapter, an attempt is made to apply the different steps of the proposed modernization plan on one module of the reservation system. The description of the process is explained in Section 6.1. All the steps were not applicable to the module I worked on, so I have tried to find examples of pieces of code that were worked on by other developers at Enkora, reflecting the changes required/suggested in the plan. The benefits of the plan are discussed in Section 6.2. The disadvantages of the code change (if any) are discussed in Section 6.3.

6.1 The process

Efforts to modernize the legacy code were streamlined according to the modernization steps discussed by the Chicken Little method in Section 5.3. For this purpose the title of each step is kept the same as that of the Chicken Little Method and the explanation below it describes what actions I have taken to implement the step.

The system that will be modernized is part of Enkora's reservation system. Enkora's reservation system was described in detail in Section 3 and Section 5.1. The reservation system/ shop (as referenced in the codebase) uses a calendar through which events can be clicked and reserved. This calendar module is going to be migrated for the purpose of this thesis. The steps followed are described below.

1. *Iteratively analyze the legacy information system*

Enkora is currently in the process of migrating the *shop* component of the reservation software. Migrating the whole module is quite a large undertaking as it has many functionalities like showing different services, login module, viewing the user's details, viewing the user's reservations, receipts etc. These can be divided into smaller working components. Few of the modules are shown in Figure 6.1. The Chicken Little method suggests migrating smaller modules to reduce the risk in case the migration fails.

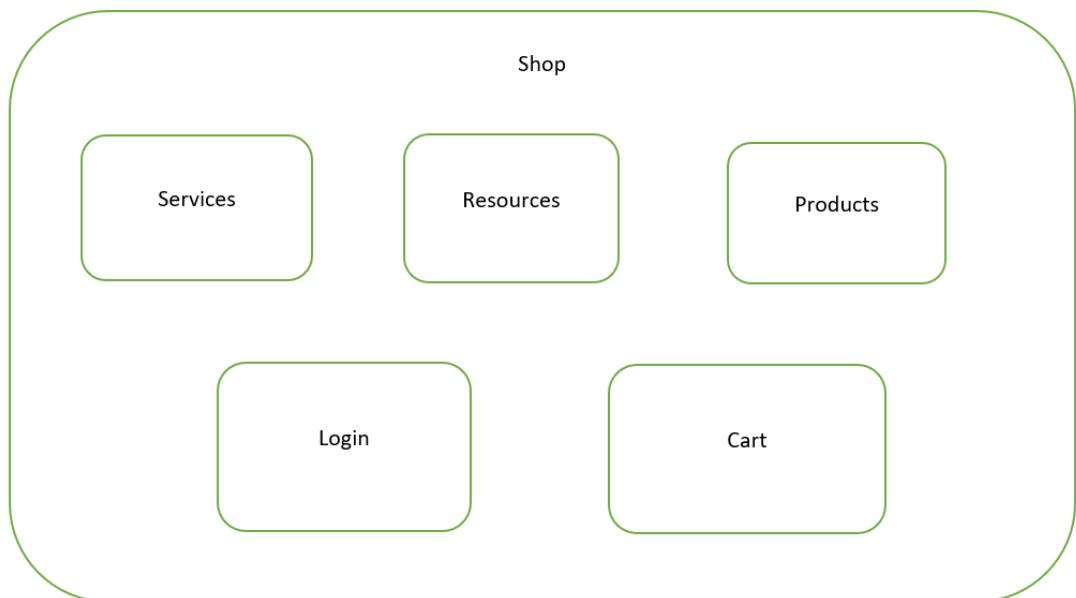


Figure 6.1: Pictorial representation of the components/modules in shop module

I begin by choosing a smaller module and understanding all the features/services provided by that module. To understand what a certain module does, discussions were held with the senior software engineer who started this process and I was assigned a module for implementation. I start by clicking around in the legacy module's user interface, to see what it does.

For the purpose of this thesis, the chosen module is the *calendar view in the webshop*. Here, customers can navigate to any week of the year and the

calendar will display all the events at a specified location for a chosen service. It also has the feature of providing Week, Month or Day views. The displayed event for each date and time can have events for multiple resources at the location. And they would be coloured according to availability. If the event is green/available, then by clicking the event, we get the options of “Reserve” or “Cancel”. On hovering over an event, it should open a pop-up modal which gives information about the price for different resources for that particular date and time.

2. *Iteratively decompose the legacy information system structure*

For this step, I first tried figuring out how to define what the boundary of this module would be. Since Angular uses components to encapsulate views and its function, it is important to decide what features could be grouped together as part of the module. I decided to have the calendar as a component, which should accept data in a certain format and display the events based on the data provided.

In Figure 6.2 the hierarchy of the different modules are shown. The *Shop* module contains the *Resource* module which contains information about the location selected and the resource availability for the location. The *Basic* module modifies the events received from the resource module and makes it suitable to be used by the calendar module.

3. *Iteratively design the target interfaces*

Here, I was trying to design the inter-module interface and the user interface for the component.

The calendar module would be referenced/called in a parent module. The information that would be required by the calendar module, which will be passed from the parent to this module, was designed and will be referenced in

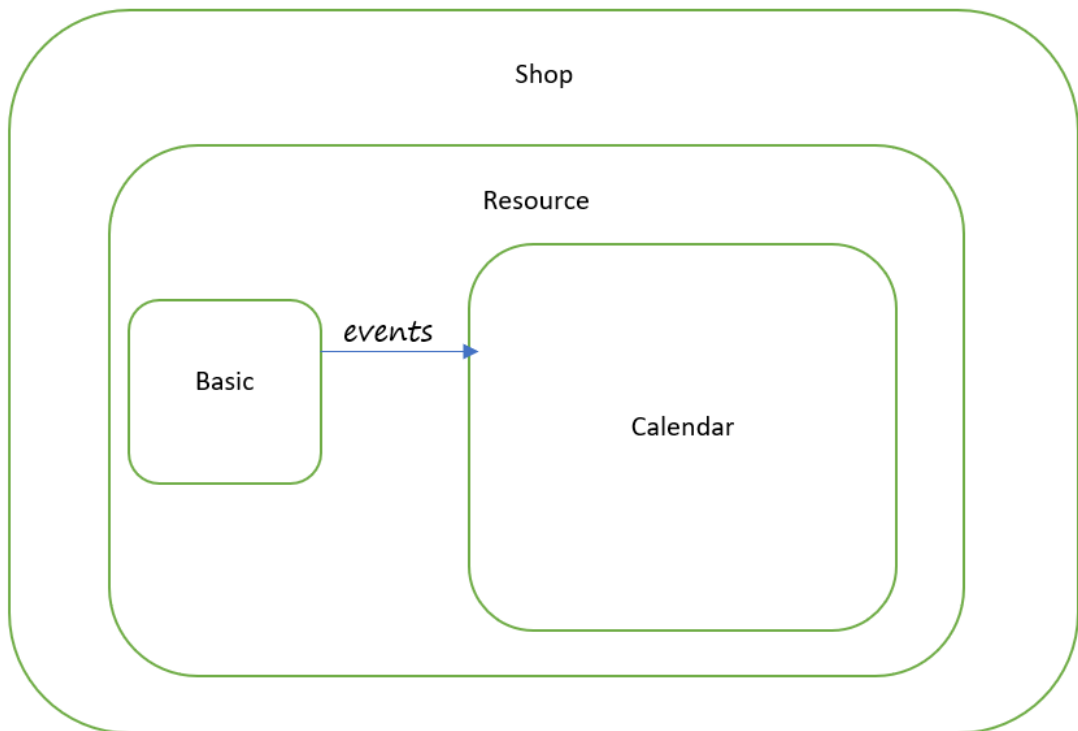


Figure 6.2: Calendar module

the inter-module interface. For this purpose the documentation of the chosen 3rd party calendar component will need to be referenced. The resource module contains information about the availability of the resources and time for which they are available. The resource-availability information needs to be formatted to be visible in a chosen calendar.

One of the requirements was to get a calendar widget/library that easily integrates with Angular and provides TypeScript support. This was not as straightforward, since initially my assumption was that any calendar library would integrate with any version of Angular. But after few tries with 2 different calendar libraries, I realized that my previous assumption was wrong. It was important to know what version of Angular Enkora was already using and then based on that version, a calendar library needed to be found and

taken into use.

Since Enkora was trying to use the latest version of Angular, there was also the added doubt if the chosen calendar will be able to support all the features that the current calendar provides.

Each time-slot in the calendar shows availability of multiple resources for a location for the specified time. Making that possible for the chosen calendar was a little difficult. The calendar documentation and StackOverflow were referenced when attempting to implement this. Understanding how this 3rd party component should integrate with the in-house components took some time.

4. *Iteratively design the target application*

This *calendar* module is part of a bigger module which is the *shop* module. But within this calendar module I did not feel the need to break it further into smaller components. So work was continued in developing this component. The events received from the *Resource* module had to be redesigned to suit the calendar. This would be done in the Basic component. Having smaller components makes it more re-usable. So the calendar module could be referenced in any other module when needed.

5. *Iteratively design the target database*

There was no work done towards this step. The APIs to the database were not changed since no database changes were required. The components are expected to access and receive the data same as the legacy code did.

6. *Iteratively install the target environment*

The target environment was already set up since this project of transitioning the framework was started 2 years ago. So I have not had to make any new

changes here. But in brief the target environment involves installing the Angular CLI so that components can be created. The following information was gathered on interviewing the Senior Software Engineer. In order to start moving code, non-REST backend functions were created that produce the same data as the standard call API. An Angular service was created that sits over the CallService (the above non-REST backend functions) in the legacy code, so if there is a cart or a product list, we can get it from the backend and update, reload and access it from any part of the new code. Also routing code was written so that when the change from the legacy code to the new code happens, in the website the url paths would not be changed drastically.

7. *Iteratively create and install the necessary gateways*

In this project, access is restricted to the target system by use of parameters. This serves the same purpose as the gateways described in the Chicken Little method. The parameter created here is “*turn on new ng shop*”. If set to true, then the old URL will be redirected to the new shop and if false it will be to the legacy shop. And if set to “-1” then both the legacy shop and angular shop will run in parallel. This was set to -1 during the development process.

8. *Iteratively migrate the legacy database*

This step was not required for this project since no database changes were made as described previously in Step 5.

9. *Iteratively migrate the legacy application*

In this step the component is created using the Angular CLI. Angular documentation defines the angular CLI as a command-line interface tool that can be used to initialize, develop, scaffold, and maintain Angular applications directly from a command shell [47]. The application module which was designed in Step 4 is now created. When the component is created it also

creates a *.specs.ts* file in the folder which contains some basic unit tests for the component. An example of the specs file that gets created for the calendar component is shown in Figure 6.3. It checks whether the CalendarComponent gets created. This is a basic test created for each new component created. Further tests can be added when this component is completed and more features are implemented. Adding simple unit tests to these components makes it possible to catch bugs as the complexity increases.

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { ShopResourceCalendarComponent } from './calendar.component';

xdescribe( 'ShopResourceCalendarComponent', { specDefinitions: () => {
  — let component : ShopResourceCalendarComponent;
  — let fixture : ComponentFixture<ShopResourceCalendarComponent>;

  — beforeEach(async( fn: () => {
    — TestBed.configureTestingModule( moduleDef: {
      — declarations : [ShopResourceCalendarComponent]
    — })
    — .compileComponents();
  — }));

  — beforeEach( action: () => {
    — fixture = TestBed.createComponent(ShopResourceCalendarComponent);
    — component = fixture.componentInstance;
    — fixture.detectChanges();
  — });

  — it( expectation: 'should create', assertion: () => {
    — expect(component).toBeTruthy();
  — });
});
```

Figure 6.3: Example of specs file for a module

10. Iteratively migrate the legacy interfaces

Since this module is only one small part of a bigger module (the shop module) the calendar module interacts with other modules within the shop module.

The parent module (named as `basic.component`) calls the calendar module. The events for the selected location and the date are passed to the calendar component from `basic.component`. Thus the inter-module interface designed previously is implemented.

When the legacy application is migrated, the created interfaces are also migrated.

At this time we also try to create interfaces for the arguments that the functions within the applications use. This will make the code more readable and thus more maintainable.

11. *Iteratively cut over to the target information system*

As each smaller module becomes ready, they are given to the product team for testing.

But once the shop module is completed, more extensive testing will be required. The cutover will only happen after the *shop* module is reengineered. The migration of the shop module was not possible during the thesis period, due to the time constraint. But in theory, the parameter which acts as an application gateway switching between the legacy and target version would be removed once the target version is well tested and the target system does everything that the legacy application did, and thus completing the modernization process for the module.

To understand all the benefits of this change, the effects of this change would have to be studied for a longer time. By using Angular and TypeScript, some immediate benefits are already noticed which were problems in the legacy code like no templating engine is required now to serve the html pages and more test cases are present. More details about the benefits are explained in the Section 6.2.

6.2 Benefits

By choosing to use Angular, a Component-based software engineering (CBSE) approach is employed, where each data structure which does a specific action can be implemented as a unique component and these components can be reused in different parts of the code when necessary. So by changing to the framework-language combination to Angular and TypeScript, the project benefits in the following ways:

1. *Code reuse is easier*

Once a component is created, it can be used and reused throughout the project, reducing code repetition and development time. The associated data and data handling for a template is stored within the component, making it easier to identify how the data is obtained and manipulated.

2. *Data Binding*

Angular provides a Data Binding feature. Data Binding is a technique of linking the data to the view layer. This could only be reproduced in Backbone with a lot of written code.

3. *Lines of code*

Angular uses components to define the data and structure that is to be displayed on each view of the screen. Each component generates fewer lines of JS code, which is less than 100 lines of generated code.

4. *No template engine used*

Angular does not need any templating engine to display the html pages. This takes care of displaying even the most complex html pages. Angular's documentation describes the functionality of the Angular ahead-of-time (AOT) compiler as "converting the Angular HTML and TypeScript code into efficient JavaScript code during the build phase before the browser downloads

and runs that code”[48]. It is possible to render the pages to the browser faster because the application is compiled during the build process.[48]. The client browser that opens the web application, gets the pre-compiled version of the application.[48].

5. *Better security*

The Ahead-of-time (AOT) compiler also gives the added benefit of providing fewer opportunities for injection attacks since HTML templates and components are converted to JavaScript files and there are no requirements for client-side HTML or JS evaluations[48].

Another security feature that comes along with Angular are “route-guards”. Angular’s route-guards are interfaces which can tell the router whether or not it should allow navigation to a requested route.

6. *Easier Maintenance and Testing*

Currently, the codebase has multiple languages and frameworks which together make up the frontend code. Some of the code is already implemented in TypeScript and Angular, so if all the frontend code is implemented in the same way, it provides uniformity in the code base, making it easier to read and maintain code. Code reuse also allows for easier maintenance. A maintainable code base helps in reducing the future possibilities of technical debt. That is something that this new code base can help achieve.

Since the code is more modular in smaller components, it is easier to create unit tests for each component. Reducing the scope of the test and therefore the complexity of the tests.

7. *Better tool support*

TypeScript is best when static type checking and better tool support is re-

quired. Since CoffeeScript is not well maintained anymore, the tools are getting old and out of date. For the backend code Enkora uses PHP and the CoffeeScript is compiled using a PHP library that is not maintained anymore. So if the compiler has any issues, there will not be any further updates made to it and then for the compiler to work, debugging and fixing would have to be done in-house.

8. *Larger online community*

Based on Github's programming language-based pull requests for 2020 shown in Figure 6.4, we notice that TypeScript has an increase of 7.4% in the number of pull requests, whereas CoffeeScript has an increase of only 0.246%. CoffeeScript has a dwindling online community now, whereas TypeScript has a larger following and a larger company backing it (Microsoft). This is especially useful during the development phase, help is required in figuring out how to get something implemented or to find out how certain things work. During such times, it's beneficial to have a wider community to discuss these with.

9. *Lower costs*

Having to maintain fewer language- framework combinations will lower development costs. By removing the use of languages that have a lower backing/ following, we increase the chance of finding more developers with the necessary skill level.

6.3 Disadvantages

On interviewing the Senior Software Engineer, I was able to learn that one disadvantage for developers was the requirement to compile the Angular code by the developers which requires some waiting time and having to commit the generated

build files. This does have an advantage for the end user, since the website will be loaded faster as the compiled files are served.

Another disadvantage is that if some error occurs in this new code, it points to the compiled (minimized) code which is difficult to decipher.

6.4 Problems faced during the implementation

Since this project was started some time ago, certain parts of the shop module were already done by different developers. So while trying to understand which component within the shop module to start with for this thesis, it was important to figure out how much of the shop module was already done and not waste efforts towards a module that was already implemented in some part.

Understanding the features of a module just by clicking around on the user interface partially worked for the chosen module. But to further understand some of the reasons behind having certain features implemented a certain way, time was required from the product team who work with this software more frequently. This was not always possible, owing to their busy schedules.

Also another aspect was understanding what functions should be grouped together in a component. This was quite confusing, since we also needed to think from the point of making them reusable.

Another hindering factor during this implementation was my lack of experience working with Angular. More time was spent on figuring out how components work and understanding the different ways of including events and data to these components.

PULL REQUESTS

Year Quarter

2020 3

| # Ranking | Programming Language | Percentage (Change) | Trend |
|-----------|----------------------|---------------------|-------|
| 1 | JavaScript | 20.147% (-0.161%) | |
| 2 | Python | 15.873% (-1.185%) | |
| 3 | Java | 11.404% (+1.038%) | |
| 4 | Go | 8.814% (+0.005%) | |
| 5 | TypeScript | 7.488% (+1.479%) | ^ |
| 6 | C++ | 6.934% (-0.117%) | v |
| 7 | Ruby | 6.185% (+0.316%) | |
| 8 | PHP | 5.030% (-0.034%) | |
| 9 | C# | 3.716% (-0.082%) | |
| 10 | C | 2.890% (-0.305%) | |
| 11 | Shell | 1.863% (-0.198%) | |
| 12 | Scala | 1.544% (-0.255%) | |
| 13 | Dart | 1.113% (+0.346%) | ^ |
| 14 | Rust | 0.919% (-0.080%) | v |
| 15 | Kotlin | 0.712% (+0.002%) | ^ |
| 16 | Swift | 0.694% (-0.149%) | v |
| 17 | Groovy | 0.440% (+0.030%) | ^ |
| 18 | DM | 0.414% (+0.019%) | ^ |
| 19 | Objective-C | 0.387% (-0.164%) | v |
| 20 | Elixir | 0.354% (-0.111%) | v |
| 21 | Perl | 0.258% (-0.049%) | |
| 22 | CoffeeScript | 0.246% (+0.021%) | ^ |
| 23 | PowerShell | 0.243% (+0.036%) | ^ |

Figure 6.4: Github pull requests[49]

7 Conclusion

The aim of this thesis was to study how code modernization can be done on a legacy code base, while also providing the case study company - Enkora Oy - with an updated front-end framework which is Angular based.

Since the code base was nearly 14 years old and it had a 3rd party CoffeeScript compiler library that was no longer supported by the vendors, it was important to update the codebase by moving away from using CoffeeScript towards a modern language that had more benefits. Material was gathered on the latest web development technologies that are available. The technologies which would be used for implementing the code changes were studied, like HTML, TypeScript and the Angular Framework.

Since legacy code modernization was the requirement, popular modernization strategies and methodologies were also discussed. The modernization steps of the Chicken Little Method were chosen to be implemented with Enkora's code base. As Enkora's code base is quite large, and implementing these steps on all the modules was outside the scope of this thesis, these steps were only used on one module of the system. The process of implementing these steps were recorded.

All in all, this code and framework change did bring some benefits to the module and also helped in reversing some of the problems seen in the legacy system. Some of the advantages were, reduced code repetition and therefore increased maintainability, no need of external template engine and better tool support. But for

understanding all the benefits and disadvantages of this language and framework change, it would have to be implemented throughout the system and then studied while being taken into use for a while.

References

- [1] K. Bennett, "Legacy systems: Coping with success," vol. 12, 1995.
- [2] "Legacy system," [Online]. Available: <https://www.techopedia.com/definition/635/legacy-system>.
- [3] L. Prechelt, "A web development platform comparison by an exploratory experiment searching for emergent platform properties," vol. 37, 2011.
- [4] C. Grannell, *The Essential guide to CSS and HTML Web Design*. 2007.
- [5] *Html introduction*. [Online]. Available: https://www.w3schools.com/html/html_intro.asp.
- [6] M. West, *HTML5 foundations*. 2012.
- [7] R. S. Bruce Lawson, *Introducing HTML 5*. 2011.
- [8] I. Lunn, *CSS3 Foundations*. 2012.
- [9] E. W. John Daintith, *A Dictionary of Computing*. 2008.
- [10] *Javascript html dom*. [Online]. Available: https://www.w3schools.com/js/js_htmlDOM.asp#:~:text=%22The%20W3C%20Document%20Object%20Model,and%20style%20of%20a%20document.%22.
- [11] M. Haverbeke, *Eloquent JavaScript: A modern introduction to programming*. No Starch Press, Incorporated, 2014.
- [12] D. Goodman, *Javascript Bible*. John Wiley and Sons, 2010.

-
- [13] D. R. Brooks, *An Introduction to HTML and JavaScript for Scientists and Engineers*. Springer, London, 2007.
- [14] P. N. Mohan M, *Learn ECMAScript - Second Edition*. Packt Publishing, 2018.
- [15] I. F. Gil Fink, *Pro Single Page Application Development*. Apress.
- [16] [Online]. Available: <https://en.wikipedia.org/wiki/TypeScript>.
- [17] C. Nance, *TypeScript Essentials*. Packt Publishing, Limited, 2014.
- [18] K. H. Chandermani Arora, *Angular 6 by Example: Get up and Running with Angular by Building Modern Real-World Web Apps*. Packt Publishing, Limited, 2018.
- [19] J. Rodzvilla, “A review of ”learning javascript design patterns”,” 2012.
- [20] A. Osmani, *Developing Backbone.js Applications: Building Better JavaScript Applications*. O’Reilly Media, 2013.
- [21] *Backbone.js*. [Online]. Available: <https://backbonejs.org/>.
- [22] *Introduction to angular concepts*. [Online]. Available: <https://angular.io/guide/architecture>.
- [23] *Introduction to modules*. [Online]. Available: <https://angular.io/guide/architecture-modules>.
- [24] *Introduction to the server side*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction.
- [25] A. Sinisalo, “Web frontend component quality model,” M.S. thesis, 2017.
- [26] M. Rouse, *Native app*. [Online]. Available: <https://searchsoftwarequality.techtarget.com/definition/native-application-native-app>.
- [27] *Using web workers*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers.

- [28] H. K. A. Bakar and R. Razali, "A preliminary review of legacy information systems evaluation models," in *2013 International Conference on Research and Innovation in Information Systems (ICRIIS)*, 2013, pp. 314–318.
- [29] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage, "How do professionals perceive legacy systems and software modernization?" In *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: Association for Computing Machinery, 2014, pp. 36–47, ISBN: 9781450327565. DOI: 10.1145/2568225.2568318. [Online]. Available: <https://doi.org/10.1145/2568225.2568318>.
- [30] I. Crotty James ; Horrocks, "Managing legacy system costs: A case study of a meta-assessment model to identify solutions in a large financial services company," 2017.
- [31] H. K. A. Bakar, R. Razali, and D. I. Jambari, "A guidance to legacy systems modernization," *International Journal on Advanced Science, Engineering and Information Technology*, vol. 10, no. 3, pp. 1042–1050, 2020, ISSN: 2088-5334. DOI: 10.18517/ijaseit.10.3.10265. [Online]. Available: http://ijaseit.insightsociety.org/index.php?option=com_content&view=article&id=9&Itemid=1&article_id=10265.
- [32] P. L. Zengyang Li Paris Avgerioua, "A systematic mapping study on technical debt and its management," 2015.
- [33] T. S. Girish Suryanarayana Ganesh Samarthiyam, *Refactoring for Software Design Smells : Managing Technical Debt*. Elsevier Science & Technology, 2014.
- [34] W. Cunningham, *The wycash portfolio management system*, 1992.
- [35] E. N.A, "On the role of requirements in understanding and managing technical debt," IEEE, 2012.

- [36] Z. Yu, F. M. Fahid, H. Tu, and T. Menzies, “Identifying self-admitted technical debts with jitterbug: A two-step approach,” *IEEE Transactions on Software Engineering*, pp. 1–1, 2020. DOI: 10.1109/TSE.2020.3031401.
- [37] R. Khadka, *Revisiting legacy software system modernization*, 2016.
- [38] J. R. C. Asil A. Almonaies and T. R. Dean, *Legacy system evolution towards service-oriented architecture*, 2010.
- [39] “Iso/iec/ieee international standard for software engineering : Software life cycle processes - maintenance,” Tech. Rep., 2006.
- [40] M. S. Michael L. Brodie, *Darwin: On the incremental migration of legacy information systems*, 1993.
- [41] S. Peräsaari, “An agile approach to system migration,” M.S. thesis, UNIVERSITY OF TURKU, 2013.
- [42] Bing Wu, D. Lawless, J. Bisbal, R. Richardson, J. Grimson, V. Wade, and D. O’Sullivan, “The butterfly methodology: A gateway-free approach for migrating legacy information systems,” in *Proceedings. Third IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.97TB100168)*, 1997, pp. 200–205.
- [43] M. S. Michael L. Brodie, *Legacy Information Systems Migration: Gateways, Interfaces, and the Incremental Approach*. 1995.
- [44] M. Battaglia, G. Savoia, and J. Favaro, “Renaissance: A method to migrate from legacy to immortal software systems,” in *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*, 1998, pp. 197–200.
- [45] *Underscore.js*. [Online]. Available: <http://underscorejs.org/#template>.

-
- [46] *Intellisense*. [Online]. Available: <https://code.visualstudio.com/docs/editor/intellisense#:~:text=IntelliSense%20is%20a%20general%20term,%2C%20and%20%22code%20hinting.%22>.
- [47] *Cli overview and command reference*. [Online]. Available: <https://angular.io/cli>.
- [48] *Ahead-of-time (aot) compilation*. [Online]. Available: <https://angular.io/guide/aot-compiler>.
- [49] *Githut 2.0*. [Online]. Available: https://madnight.github.io/githut/#/pull_requests/2020/3.