Aleksi Holappa

# WEB FRONTEND DEVELOPMENT WITH ELM

# WEB FRONTEND DEVELOPMENT WITH ELM

Aleksi Holappa
Bachelor's Thesis
Spring 2018
Degree program in Information and Communication Technologies, software development
Oulu University of Applied Sciences

## ABSTRACT

Oulu University of Applied Sciences
Information and Communication Technologies, Software Development

Author: Aleksi Holappa
Title of thesis: Web Frontend Development with Elm
Supervisor: Veikko Tapaninen
Term and year of completion: Spring 2018 Pages: 37

The primary purpose of this Bachelor's thesis was to implement a real-world example application written in the programming language called Elm, go through the basics of web development and Elm language, and go through a little bit of the Elm maintainability analysis.

Modern web development tools were used in this thesis work. Such as modern web browsers, open source tools, and Elm language-specific tools. The most used one was Visual Studio Code text editor which were used to write all the Elm code. The example application uses an external interface to get data which is displayed in the browser.

As a result, Elm is a great alternative for JavaScript and its libraries. The Elm language is still under the heavy development work, thus there are parts which are not included in this thesis work. Elm could be very interesting for web developers, who are seeking alternative options to create reliable frontend applications. The next thing to research would be to find out how Elm scales up in large single page applications, and how the Elm platform behaves.

Keywords: frontend, Elm, web development

# PREFACE

I want to thank Codemate Ltd. For the opportunity to take part in the Elm project and for giving me the inspiration to make this thesis work, it has been extremely valuable for me. Also, I would like to thank Veikko Tapaninen for supervising me through this project.

I am looking forward to face new challenges on web development and especially with Elm.

Oulu,

18.4.2018

Aleksi Holappa

## CONTENTS

ABSTRACT 3
PREFACE 4
CONTENTS 5
VOCABULARY 7
1 INTRODUCTION 8
2 WEB TECHNOLOGIES 9
2.1 Basic components of frontend 9
2.1.1 HTML 9
2.1.2 CSS 11
2.1.3 JavaScript 12
2.2 Web browser 14
2.3 Elm 15
2.3.1 Language 15
2.3.2 Type system 16
2.3.3 Data structures 19
2.3.4 Architecture 23
2.3.5 JavaScript interop 26
3 XKCD APPLICATION IN ELM 27
3.1 Tools 27
3.1.1 Yarn 27
3.1.2 Create-elm-app 27
3.1.3 Webpack 28
3.2 Analysis 28
3.2.1 Use cases 28
3.2.2 Activity diagram 29

3.3 Application architecture 29
3.4 Development and implementation 30
3.5 Building and bundling 30
3.6 Deployment 32
4 ELM APPLICATION MAINTAINABILITY ANALYSIS 33
4.1 API and type changes 33
CONCLUSIONS 35
REFERENCES 36

## VOCABULARY

| | |
|---|---|
| API | Application Programming Interface |
| Create-elm-app | Package used to start developing Elm applications with zero configuration |
| DOM | Document Object Model |
| Msg | Elm variable type |
| Server | Computer program which provides services |
| Tooltip | Small popup box containing text |
| UI | User interface |
| Web | A complex system of interconnected elements |
| Web application | Application type which runs in browser |

# 1 INTRODUCTION

Website count has been evolving since the first website was published, and it has become the most popular platform for interactive content and applications. In 28 years, the web development has been modernized so much that the development work is faster and more efficient nowadays. Reactive websites and applications are created by using different frameworks and languages. This thesis work compares very basic web development languages and markup languages with a functional language called Elm. The main problems in web development are maintainability, scalability and performance. (Maruti techlabs, 2018.). This thesis contains information about basic web technologies used today, information about Elm and its features compared to other languages, about an application example developed with Elm and about an analysis of the maintainability case.

# 2 WEB TECHNOLOGIES

The modern web sites and applications are dependent on few basic structural parts which have been there since user interactive web sites and applications started to appear. Besides of web sites and applications, web browsers have been evolving along with them.

## 2.1 Basic components of frontend

The web frontend itself as a term means the visible content to the user in a web browser. This specific content runs in the browser, has styling to make it look good, and functionality to make it usable by humans.

### 2.1.1 HTML

HTML stands for Hypertext Markup Language (Duckett 2011, 28). An HTML file always contain elements, which are the building blocks of the current user interface in the browser. These elements describe the structure of the web site or application in a semantic way. The HTML language structure style is usually referred to as a tree of elements. The HTML file contains a root element, which is usually a body element on modern websites and applications.

*“HTML Uses Elements to Describe the Structure of Pages.”* (Duckett 2011, 21).

As the above quotation declares, elements can describe any kind of structure of the webpage or application. This structure can be built by using, for example headings, paragraphs, lists, links, quotes, images, input fields, buttons and many other items. These elements can also be used to create forms which can be used to collect information.

```html
<body>
    <h1>Heading for article</h1>
    <p>
        Lorem ipsum dolor sit amet,
        consectetur adipiscing elit.
        Aliquam at nisl dolor.
    </p>
</body>
```

```html
<body>
    <p>Heading for article</p>
    <h1>
        Lorem ipsum dolor sit amet,
        consectetur adipiscing elit.
        Aliquam at nisl dolor.
    </h1>
</body>
```

*FIGURE 1. Comparison between correct and incorrect semantic elements*

In figure 1 there is an example of how semantic rules apply to html code design. The picture shows two cases; the left side of the picture is correct and the right side is incorrect. The H1 element stands for a Heading 1 and the P element for a paragraph. An element should always describe the content of element in a semantic way, as seen in figure 1 in the left side case. The header text “Heading for article” clearly indicates that it should be treated and marked as a header but on the right side of the figure it is treated as a paragraph, which is incorrect in a semantic way of thinking. The semantic way of thinking becomes important when, for example a script element is changing the DOMs elements, which are built from HTML in the browser. This can lead to an incorrect behavior of the end-user on the website or application.

**HTML Attributes**

Attributes provide additional information about the contents of an element. They appear on the opening tag of the element and consist of two parts: a name and a value, separated by an equal sign (Duckett 2011, 25). These attributes give an ability to identify a specific element by defining id, giving a tooltip for the user, applying styling to the element and various other actions.

*“Attributes provide additional information about the contents of an element. They appear on the opening tag of the element and are made up of two parts: a name and a value, separated by an equals sign.”* (Duckett 2011, 26).

```
<body class="Article">
    <h1 id="articleheading">Heading for article</h1>
    <p>
        Lorem ipsum dolor sit amet,
        consectetur adipiscing elit.
        Quisque et felis varius tellus interdum laoreet.
    </p>
    <button title="Press for more text">Press me for more text</button>
</body>
```

*FIGURE 2. Usage of different element attributes*

Figure 2 shows the usage of these earlier mentioned attributes. The body element now has a class named “Article”, which means that all styling declarations included in that class are applied to this specific element. The H1 element has an id attribute, which declares a unique id for this element. Now there is a possibility to refer to this element from JavaScript (1.1.3 JavaScript) code and to manipulate the element. The button element has title attribute and the attribute itself has a text value. This text value is displayed in a small popup, when moving a cursor over the button and holding it still. This functionality is meant to give more information about that element and what it does.

### 2.1.2 CSS

Cascading Style Sheets (CSS) is used to style documents which are written in a markup language. The used markup language is usually HTML.

*“CSS works by associating rules with HTML elements. These rules govern how the content of specified elements should be displayed.”* (Duckett 2011, 231).

When writing CSS, the purpose is to provide styling and layout for specific elements in the view. Pointing to that specific element or layout is done by using a selector. This selector consists of an element type (e.g. button) and optional pattern, class or an element id.

```css
button{
    background-color: yellow;
    width: 5rem;
}

#submitbutton{
    background-color: blue;
    height: 2rem;
}

.submitbutton{
    background-color: red;
    position: relative;
}
```

*FIGURE 3. Example of different selector definitions in CSS-file*

Figure 3 above presents three most common ways to define a selector with the CSS code. This CSS file must be imported to an HTML file in order to use these selectors for styling. The first selector will apply the styling defined inside the selector braces to all button elements where this CSS file is imported. The second selector applies only to the element which has a "submitbutton" id defined by an attribute. When referring to the element by an id from the CSS file, the selector must have a pound sign prefix in the beginning of the selector. The bottom example is a so called CSS class. The class selector does not differ much from the id selector, only the prefix is different, and in the class, it is a dot. The example of using this kind of selector and CSS definition is shown in figure 2. In figure 2 the body element has a defined class-attribute which takes a string type of value. This value is the exact CSS class selector without the dot prefix.

### 2.1.3 JavaScript

The JavaScript language is used to create the functionality for websites and applications. All modern web browsers have support for JavaScript nowadays. This gives an ability to create modern web applications efficiently and with functionalities, which was not possible before JavaScript. (Haverbeke 2018, 6).

*“a standard document was written to describe the way the JavaScript language should work, so that the various pieces of software that claimed to support JavaScript were actually talking about the same language. This is called the ECMAScript standard, after the Ecma International organization that did the standardization.”* (Haverbeke 2018, 6).

This means that JavaScript always has a standardization and it is well documented. This helps web developing a lot, since every browser follows this standard, which means that JavaScript will behave and work similarly for almost all browsers, except in Internet Explorer. (Zaytsev, 2018).

### Usage along with HTML

The following example will contain the basic usage of JavaScript with an HTML file, a basic use-case and functionality, without refreshing the page. It is one of the main benefits of JavaScript, to have a dynamic content on the page.

```
function changeText() {
  let para = document.getElementById("first-paragraph");
  para.innerHTML = "Nice to meet you!";
}
```

*FIGURE 4. Function named “changeText”*

Good coding standards always begin with a proper naming. The “changeText” function in figure 4 is named that way because it is representational what the function does when executed. The two code lines between brackets in figure 4 are the code lines which will be executed.

```html
<script src="./script.js"></script>

<body>
    <p id="first-paragraph">
        Hi, It's me, Aleksi!
    </p>
    <button onclick="changeText()">Press here</button>
</body>
```

*FIGURE 5. HTML-file with simple UI and button with functionality*

In figure 5 it can be seen how the “changeText” function is called. The button element takes an onclick attribute. This attribute is a browser’s event. Basically, whenever a user clicks this button element, it will trigger the “changeText” function and execute the code in that function. As seen in figure 5, the paragraph element has an id attribute with a value “first-paragraph”. In figure 4 the first code line inside the “changeText” function inserts this paragraph element to a “para”-variable. This specific variable now contains the whole element with all its properties. On the second line the innerHTML property is accessed and the paragraph value/text to “Nice to meet you!” is changed. To be able to apply this functionality, the script file must be imported to the HTML file with script tags, as seen in figure 5, on the first line.

This value change is made in real-time and it does not require refreshing the site. This is a very basic example of the dynamic content changing in the user interface.

### 2.2 Web browser

Commonly used as a browser there is a computer application presenting webpages or applications. It can also retrieve and transfer information in or out through the world wide web from a resource. These resources are servers in most cases. (Web browser, 2018).

These servers return files to be displayed in a browser’s viewport. There are a few most common browsers which handle these files similarly. These files are commonly HTML, CSS and JavaScript files. These files are static, but can display dynamic content, which is fetched from API, for example.

Modern browsers feature different privacy and security solutions. The most used basic security option is HTTPS, which encrypts the incoming and outgoing data in the browser. Browsers also provide quick and easy ways to delete cached information, history of browsing and cookies. (Web browser, 2018).

## 2.3 Elm

The Elm language is a functional programming language for creating web applications and websites. Its main benefits compared to other languages, frameworks or UI libraries, are that it always compiles to JavaScript and theresore it can be used as a part of an existing website or application, or it can be used to build the whole site or application in Elm. This section contains information about the Elm application architecture, type system, data structure and other language specific features. (Elm guide, 2018).

### 2.3.1 Language

The Elm language syntax is similar to Haskell's, and Haskell is used to program Elm's compiler. When compared to other languages, for example JavaScript, there is no curly brackets for a function definition. Instead of curly brackets, Elm uses a 4-space indentation, which makes the syntax very readable and easy to understand. (Pandy, 2014). Also, Elm does not require a semicolon use for statements. When programming with a functional language, every time a function is defined it always returns a value. This can be confusing for programmers who have a JavaScript background. Below in figure 6 there is an example of function definition in Elm.

```
isEven : Int -> Int -> Bool
isEven a b =
    if a == b then
        True
    else
        False
```

*FIGURE 6. Elm language syntax example*

### 2.3.2 Type system

Elm's type system is built to be static and a strong type system. This means that if the developer is trying to pass, for example, a string to function as a parameter, but the function is annotated to take only an array type parameter, Elm's compiler will give an error and complain about this. This makes Elm's type system very handy and reliable. (Waselnuk, 2016).

*"Static typing means that the actual source code (the text file) of your Elm program is verified by a compiler."* (Waselnuk, 2016).

#### Type aliases

*"Type aliases allow you to attach human readable names to existing types." (Waselnuk, 2016).*

With this ability, developers can write more understandable code and this affects straight to the code quality. These type aliases can be used just as any other type, such as a string or an integer.

```
type alias Book =
    { name : String
    , writer : WriterName
    , year : Int
    }


type alias WriterName =
    { firstName : String
    , lastName : String
    }


initBook : Int -> String -> WriterName -> Book
initBook year name writer =
    { name = name
    , writer = writer
    , year = year
    }
```

*FIGURE 7. Example of using type aliases in records*

In the above example, the type alias called as “Book” is defined. This type alias contains a record, which defines three different keys and value types to them. Name and year keys are defined to take only a string and integer type of values. There is one exception. The writer key has a type of “WriterName”, which is another type alias as seen in figure 7. The writerName type alias has two keys and both keys take string type values.

Defining type aliases like this, it is possible to achieve more modularity in this application. In case the purpose is just to manipulate the first and last name of the writer, it can be handled with the WriterName type alias instead of using the whole Book type alias.

### Union types

*“With tagged unions, you can define a type that represents the possibility of multiple types. This is one of the key features that supports the Elm architecture.”* (Waselnuk, 2016).

This comes very useful when some action can produce a different result depending on other data inside the module or user action. For example, if there is a union type called

“Door”, the door has only two possible types which are “Open” or “Closed”. Every time when using the “Door” union type, it has only two available options, thus the application does not need to cover any other values than the two possible values, “Open or “Closed” in this case. This creates reliability in type checking.

```
type Request
    = NotExecuted
    | Loading
    | Success
    | Error


getStateOfRequest : Request -> String
getStateOfRequest request =
    case request of
        NotExecuted ->
            "No request has been made"

        Loading ->
            "Data is still loading..."

        Success ->
            "Data requested successfully"

        Error ->
            "Error occurred"
```

*FIGURE 8. Example of using union types in Elm*

In the above example, a “Request” union type with four possible types is denifed. This union type can only be some of those four types, nothing else. Under this union type defining is a function, which returns a string depending on, which type the “request” parameter contains. Inside this particular function is a case-expression which has all the four possible types covered. If not, the Elm’s compiler will give an error and the application will not be compiled.

### Type annotations

*“Type annotations describe the input and output types of functions.”* (Reimann, 2016).

This is not required when writing Elm functions but it is highly recommended, since it makes the Elm code more readable.

```
getStateOfRequest : Request -> String
```

*FIGURE 9. Elm type annotation of function*

In figure 9 there is a type annotation from a union type example. This annotation tells the name of the function, the first parameter type, which is “Request”, and the type of value which has been returned when calling this function. Also, there can be multiple parameters and all these parameters are separated with an arrow, but the rule is the same, the last type is the returned type, all others are types of the parameters.

```
getStateOfRequest request =
    case request of
        NotExecuted ->
            "No request has been made"

        Loading ->
            "Data is still loading..."

        Success ->
            "Data requested successfully"

        Error ->
            "Error occurred"
```

*FIGURE 10. Function definition without type annotation*

Above in the figure 10 it is possible see the same function, without the type annotation. This is not readable and it may slow down the development work since it is not possible to quickly see which is the returned type of this function. When the type annotation is in place and defined, it speeds up the development work since the annotation itself tells the type of returned value.

### 2.3.3 Data structures

Elm has its own set of data structures, which are record, tuple, list, set, dict and array. Almost all of these are Elm’s language specific data structures, except array, which is used in other programming languages, for example, in JavaScript. (Hanhinen, 2017).

**Record**

*“A record is a lightweight labeled data structure.”* (Records, 2018).

```
type alias Book =
    { name : String
    , writer : WriterName
    , year : Int
    }


type alias WriterName =
    { firstName : String
    , lastName : String
    }
```

*FIGURE 11. example of record data structure*

As seen in figure 11, there are two types of aliases that are holding different record structures. Records remind very much objects, which are used in JavaScript. The main difference between objects and records are that in Elm it is not possible to ask for a field that does not exist. (Records, 2018).

Records are always structured with at least one field. The syntax of constructing records is simple. First, a name is given to the field, after that a colon and a last type. This can be translated “create a field with this given name which has this kind of type”.

All these field values can be accessed through a variable where the record is. If a variable called “elmBook”, has the “Book” type alias and data in it, the name field can be called just simply typing “elmBook.name”, and it will return the value of this name-field.

**Tuple**

The tuple data structure is a set of values. And these values can be string, integer, record or any other available data type by Elm. (Core Language, 2018).

```
isUserValid : Int -> ( Bool, String )
isUserValid valid =
    if valid == 0 then
        ( False, "User not valid" )
    else
        ( True, "User is valid" )
```

*FIGURE 12. Usage of tuple-data structure*

Figure 12 shows an example of tuple-data structure usage in a function type annotation. This kind of data structure comes useful when the purpose is to return more than one value from the function. If the data structure comes more complex, a better solution would be the earlier mentioned record type of structure, instead of the tuple. Values of the tuple are always wrapped in parentheses and separated with commas.

**List**

Lists are very similar to JavaScript arrays. Lists contain values, and all those values must have the same type. (Core language, 2018).

Lists are very useful when it comes to populating dropdown elements with a list of values, showing the list of values in the view as a bullet list or other ways to utilize the list, where items indexes are not mandatory.

```
type alias ListOfNames =
    List String


initNames : ListOfNames
initNames =
    [ "Aleksi", "Antti", "Jesse", "Lauri" ]
```

*FIGURE 13. Definition and usage of list data structure*

As seen in figure 13, a “ListOfNames” type alias is defined to have a type list, and all list items must have a type string. Under this type alias, there is a function which returns a list of names, and the returned value matches the function type annotation, since all items are strings, items are wrapped to square brackets and separated with commas.

**Set**

*"A set is a collection of unique items; in Elm, these items must be comparables (ints, floats, chars, strings, lists, or tuples) and of the same type."* (Kelly, 2016).

When speaking of comparables in Elm, these values can be compared to each other. A set data structure creates it a little bit easier, with helper functions. If there is a record, with a field myCar, this field can contain a value of Set.empty, which constructs an empty set to this field. Afterwards this field in the record can be a filled with a list, int, float or string. After that, the comparison can be made between two different sets and they can be used.

**Dict**

*"A dict maps unique keys to values. There's a fair amount of overlap in the use cases for dictionaries and sets, but dictionaries allow for storing a bit more information. Since values are indexed by unique keys, we can try to Dict.get 'some comparable' Dict.empty, which may give us Nothing."* (Kelly, 2016).

This is very much similar to sets, as mentioned above. Dicts are useful when it comes to storing and manipulating the dict items. This is a great way to store data from the user input and handle it inside the module since the dict data structure offers various use cases.

**Array**

*"Arrays are ordered collections, where each item in the collection must be of the same type. In Elm, [] doesn't correspond to arrays ([] is an empty list, discussed above). To create an empty array, use Array.empty."* (Kelly, 2016).

Arrays in Elm are quite similar to lists, but with the difference mentioned above, arrays are always ordered and the values can be accessed in various ways.

### 2.3.4 Architecture

The basic elm application architecture is divided into four different parts. The parts are model, view, update and runtime. This structure is the most recommended way to construct modules with an internal state and functionality. When there is an application, which is built with multiple different modules, the runtime is only included in the main module since it wraps the whole application to it. Nevertheless, all modules, which are included and used in the application, will communicate with the runtime. (Reimann, 2016).

#### Model

```
---- MODEL ----


type alias Model =
    { age : Int
    , name : String
    }


init : ( Model, Cmd Msg )
init =
    ( { age = 22
      , name = "Aleksi"
      }
    , Cmd.none
    )
```

*FIGURE 14. Elm language syntax example*

In figure 7, the Model can represent the user's current module data as well as the state. In most cases, the Model is usually referenced as a record that describes the structure of data. The model can also be only string, int or boolean, but this is a very rare case in modern application structures. This is the only place where the user's module specific

data is held. All the data is immutable. The init function in figure 7 is for initializing the Model data. Since Elm is functionally written, there cannot be record keys without a value. This init function is executed by the runtime.

### View

```
view : Model -> Html Msg
view model =
    div []
        [ h1 [] [ text "Welcome to elm application" ]
        , p [] [ "This is a very basic application which is written in Elm" ]
        ]
```

*FIGURE 15. Example of view*

This function contains all the user interface elements, such as headers, images, buttons, forms and other UI related elements. This function itself just presents a view to be displayed on the web browser. Another function which returns user interface elements, can be called an inside view function. As seen in Figure 15, it contains only one header and paragraph. In figure 15 the example model is passed to this view function, this means that data or state in the view can be accessed and the current values of it can be rendered. In this example, the model is not utilized in the function, therefore it is useless.

### Update

This specific part of module is responsible for all the updates for model's data or state. It is called when the user interaction happens in a view since the view returns Html and Msg.

```
type Msg
    = NoOp
    | ChangeName


update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        NoOp ->
            ( model, Cmd.none )

        ChangeName ->
            ( { model | name = "Niko" }, Cmd.none )
```

*FIGURE 16. Example of Msg type and update function*

The above example in figure 16 represents a new union type called Msg, which is short for message. It has two different values, NoOp, which is the standard “do nothing” message and “ChangeName” message. The update function takes Msg and Model as a parameter. Inside the function Elm compiler forces to go through every possible value of Msg type which is defined in the Msg union type. The “ChangeName” message simply returns a model with a new value “Niko” to name a field and a command, which does not trigger any other message, since it calls none.

**Runtime**

```
---- PROGRAM ----


main : Program Never Model Msg
main =
    Html.program
        { view = view
        , init = init
        , update = update
        }
```

*FIGURE 17. Main function wrapping the application to program*

In figure 17 the main function wires up all the three critical parts of the program. Those parts are earlier mentioned update, view and model/init. The runtime is initialized with these parts and it creates an endless loop to run the program in a browser. Runtime will now handle user interactions and manage the application state or data. (Reimann, 2016).

### 2.3.5 JavaScript interop

Since Elm is a programming language which compiles to JavaScript, there is a special way to communicate with the existing JavaScript code through so called Elm ports.

*"**any communication with JavaScript goes through a port**. Think of it like a hole in the side of your Elm program where you can send values in and out."* (JavaScript interop, 2018).

What this means, is that if there are earlier developed features or tools with JavaScript, they can be utilized through Elm ports instead of coming out with pure Elm based solution. This helps a lot, since if there is no native Elm solution available, it is possible to use the JavaScript solution and still maintain the Elm codebase for module state handling and other purposes.

# 3 XKCD APPLICATION IN ELM

The example application would be an XKCD reader. XKCD is an already existing website at https://xkcd.com/, but in this thesis work, the aim is to make a more modernized solution for reading XKCD comics. The XKCD website offers an open API for requesting data, which can be used to show e.g. comic, transcript or title.

## 3.1 Tools

Modern web development is made easy to start with proper tools. In this example project a few popular tools, which are comfortable and easy to use and offer a different kind of functionality, were selected.

### 3.1.1 Yarn

Yarn is a package manager for different packages needed in the development work. With Yarn, all needed tools and packages will be installed during the development. Its benefits are that after using Yarn in the project, everybody else who decides to do the development work, will always have the same environment, which includes installed packages and tools. This is achieved by a so called lockfile format. The lockfile format is a kind of file, which has listed all the dependencies used in the project. (Yarn, 2018).

### 3.1.2 Create-elm-app

This package is for creating and starting the Elm development with zero configuration. It is installed through Yarn but has its own package specific commands. Elm’s own tools do not provide any tools for optimization, supporting for ports or hot reloading. Hot reloading is used for real time compilation and the developer can see the changes immediately in the browser where the site or application is open. (Create-elm-app, 2018). In this project, this package is used to achieve the project initialization without manual configuration.

### 3.1.3 Webpack

*“Webpack is a static module bundler for modern JavaScript applications. When webpack processes your application, it recursively builds a dependency graph that includes every module your application needs, then packages all of those modules into one or more bundles.”* (Concepts, 2018).

Although the above quotation states that Webpack is for JavaScript applications, it can also be used with Elm. Achieve the use of Webpack, a command “elm-app eject”, which is included in the create-elm-app Yarn package, needs to be run. This command ejects the Webpack configuration files. By doing this, the user has a full control of the building process. Custom stylesheet libraries can be added and configured, also the output, different development modes and various other things can be configured.

## 3.2 Analysis

The XKCD reader application provides a very simple graphical interface for reading and browsing comics.

### 3.2.1 Use cases

- **Use-case 1:** Browse comics forward
- **Use-case 2:** Browse comics backward
- **Use-case 3:** Go to first comic
- **Use-case 4:** Go to latest comic
- **Use-case 5:** Get a random comic

There is only one actor in this case, the user. This user is associated with all use cases.

### 3.2.2 Activity diagram

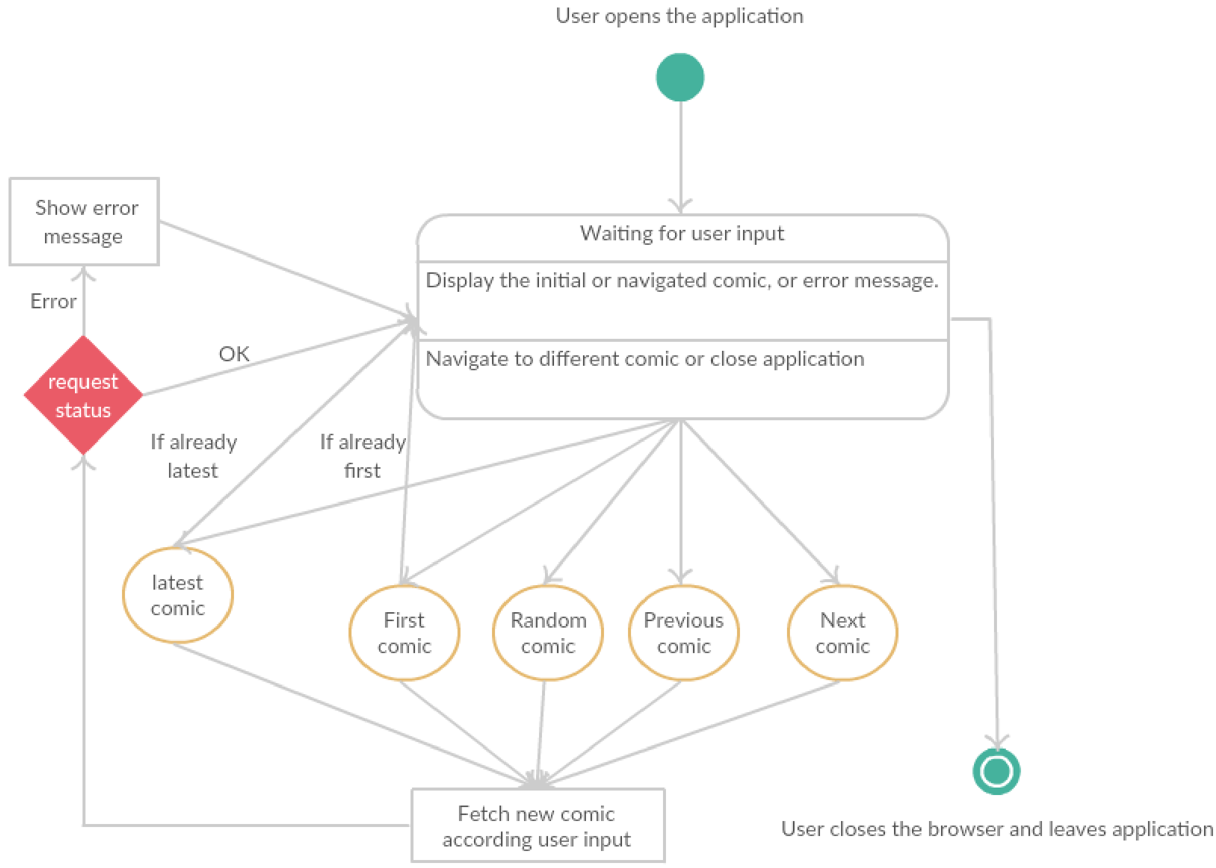


*FIGURE 18. Activity diagram of the application flow*

### 3.3 Application architecture

This application uses the same module architecture that was mentioned in section 2.3.4.

The module architecture is divided into three different modules. These modules are page, data and request. The page in this case is the main module, which is responsible for wiring everything up, updating the view and handling the internal state of the application. The data module describes the incoming comic data and what fields it should have. The request module is responsible for making the requests to the API to get data for comic.

### 3.4 Development and implementation

This XKCD application was implemented using the Elm version 0.18. Webpack was responsible for building and bundling the application for deployment. Yarn was responsible for the package and dependency managing. The version control was done with GitHub. GitHub is a popular web service, which provides an ability to save and manage code.



*FIGURE 19. Screenshot of the ready application*

### 3.5 Building and bundling

The application was built and bundled by Webpack. In this project, there were two different configurations which have different purposes. There are development and production configurations. These two configurations output a bundle, which contains static files that can be run in the browser.

The development configuration is used when developing the application on a local working environment. It sets up a local development server which runs the application and supports hot reloading. Hot-reloading is a technique which scans the project files for changes. When change occurs, the local development server builds a new bundle and the changes can be seen in the browser. The local development server and bundle can be set up with the “yarn start”-command in the command line.

The production configuration is used to create a bundle, which can be deployed and hosted in cloud services or web server. This configuration minifies all the code files which are generated to the bundle. Minifying means that all the whitespaces are removed in order to decrease the file size, which results to a smaller bundle. The production bundle can be generated with the “yarn build” command.

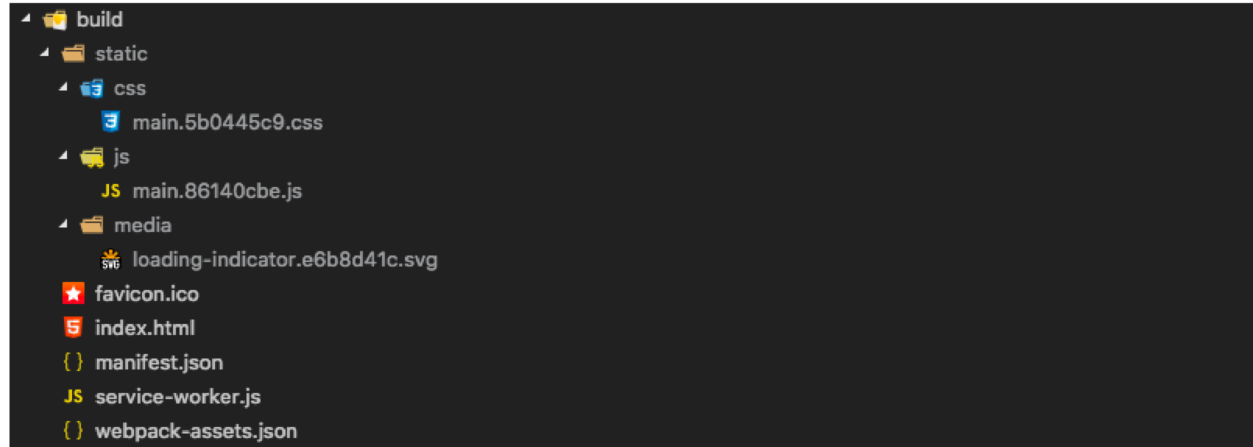


*FIGURE 20. Production bundle structure*

As seen in figure 20, the build folder contains all the static files. In the root of the folder there is index.html, which is used as an entry point for the JavaScript file in the js folder, favicon, which is displayed in the browser tab, and other needed configuration files. The visible and functional content is bundled to the JavaScript file inside the js folder. This JavaScript file contains the Elm runtime, view generation and functionality to the view, which is compiled from the Elm files. There are no separate html template files since the JavaScript file also generates the view. The CSS folder contains all the styles that are applied to the view and the media folder contains the loading indicator which is displayed when loading a new comic.

### 3.6 Deployment

GitHub also offers ability to host the static content, which is easy to use and suits the purpose of this application, since this XKCD reader application does not have any backend service (data storage or API) on the bundle. For every GitHub user, GitHub offers their own URL where the user can host this application. In this case the URL is https://hlappa.github.io/, where “hlappa” is the author’s personal GitHub account. The content in the previous URL is fetched from the GitHub repository automatically by GitHub’s own services, thus there is no need for external build or deployment scripts. The only requirement for host the static content is to name the GitHub repository as {username}.github.io. GitHub services automatically search for the index.html file from the repository and use this file as an entry point of the application.

# 4 ELM APPLICATION MAINTAINABILITY ANALYSIS

This section will concentrate on to managing the application of the section 2. How the API changes affects the application and how the changes should be dealt with.

## 4.1 API and type changes

The common problem, when building web applications, where data is fetched from the 3rd party services, is that the incoming data shape changes. In the Elm application the incoming JSON data field values must be converted to Elm values and types. This is done by Elm language specific decoding. By decoding, the incoming JSON values are converted to values that Elm can understand.

```
comicDecoder : Decoder Comic
comicDecoder =
    decode Comic
        |> required "num" int
        |> optional "alt" string ""
        |> optional "title" string ""
        |> required "safe_title" string
        |> optional "day" string ""
        |> optional "month" string ""
        |> optional "year" string ""
        |> required "img" string
        |> optional "link" string ""
        |> optional "news" string ""
        |> optional "transcript" string ""
```

*FIGURE 21. JSON-decoder of XKCD-reader application*

In figure 21 it is defined a decoder for incoming JSON data. The decoder must handle all the possible fields in the JSON. However, if the API changes, and the 3rd party API service provider adds one field, this application will not crash, since Elm does not handle this additional field at all. The developer must monitor the API, in case it changes.

If there is a new additional field, it is easy to expand this current decoder and data model by just adding this missing field to it and possibly rendering it to the view, if it is necessary.

In case, where the existing JSON data value changes its type to something else, for example, a string changes to integer, it is fast and efficient to change the decoder and data model to correspond this API change. If the changed value is used somewhere else inside the application, all the other usages of this specific value must be refactored to handle integer instead of the string.

# CONCLUSIONS

The main focus was to consider Elm as a main programming language for web development. The results were good when it comes to maintainability and learning. The only problem was a steep learning curve when the developer comes with a JavaScript background and starts to develop with Elm but it is manageable. Also, there is no native Elm solutions for everything, but those obstacles are surmountable.

I think the purpose and focus of this thesis work went pretty well one to one. I would consider using Elm in the future web projects as a main language and framework, of course bearing in mind that Elm does not provide a solution for every problem.

# REFERENCES


1. Maruti techlabs. 2018. 5 challenges in web application development. Date of retrieval: 23.4.2018. https://www.marutitech.com/5-challenges-in-web-application-development/.
2. Concepts. 2018. Webpack. Date of retrieval: 10.4.2018. https://webpack.js.org/concepts/.
3. Core Language. 2018. Elm-lang. Date of retrieval: 5.4.2018. https://guide.elm-lang.org/core_language.html.
4. Create-elm-app. 2018. Yarnpkg. Date of retrieval: 10.4.2018. https://yarnpkg.com/en/package/create-elm-app.
5. Duckett, J. 2011. HTML&CSS. Indianapolis, Indiana: John Wiley & Sons, Inc.
6. Elm guide. 2018. Elm-lang. Date of retrieval: 27.3.2018. https://guide.elm-lang.org/.
7. Haverbeke, M. 2018. Eloquent Javascript 3rd edition. Date of retrieval: 10.3.2018. https://eloquentjavascript.net/Eloquent_JavaScript.pdf.
8. Hanhinen, O. 2017. Introduction to data structures. Date of retrieval: 5.4.2018. http://ohanhi.com/master-elm-2-data-structures.html.
9. JavaScript interop. 2018. Elm-lang. Date of retrieval: 5.4.2018. https://guide.elm-lang.org/interop/javascript.html.
10. Kelly, T. 2016. Data structures in Elm. Date of retrieval: 5.4.2018. http://blog.noredink.com/post/140646140878/data-structures-in-elm.
11. Pandy, L. 2014. Elm style guide. Date of retrieval: 23.4.2018. https://gist.github.com/laszlopandy/c3bf56b6f87f71303c9f.
12. Records. 2018. Elm-lang. Date of retrieval: 5.4.2018. http://elm-lang.org/docs/records.

13. Reimann, D. 2016a. The Elm Architecture. Date of retrieval: 27.3.2018. https://dennisreimann.de/articles/elm-architecture-overview.html.

14. Reimann, D. 2016b. Elm Type Annotations. Date of retrieval: 5.4.2018. https://dennisreimann.de/articles/elm-type-annotations.html.

15. Web browser. 2018. Wikipedia. Date of retrieval: 27.3.2018. https://en.wikipedia.org/wiki/Web_browser.

16. Waselnuk, A. 2016. Understanding the Elm type system. Date of retrieval: 5.4.2018. http://www.adamwaselnuk.com/elm/2016/05/27/understanding-the-elm-type-system.html.

17. Yarn, 2018. Yarnpkg. Date of retrieval: 10.4.2018. https://yarnpkg.com/.

18. Zaytsev, J. 2018. ECMAScript compatibility table. Date of retrieval: 19.3.2018. http://kangax.github.io/compat-table/es6/.