



**BRNO UNIVERSITY OF TECHNOLOGY**

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

**FACULTY OF INFORMATION TECHNOLOGY**

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

**DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA**

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

**LIBRARY FOR MULTIPLATFORM DEVELOPMENT  
OF MOBILE APPS**

KNIHOVNA PRO MULTIPLATFORMNÍ VÝVOJ MOBILNÍCH APLIKACÍ

**BACHELOR'S THESIS**

BAKALÁŘSKÁ PRÁCE

**AUTHOR**

AUTOR PRÁCE

**MICHAL KOVAŘÍK**

**SUPERVISOR**

VEDOUCÍ PRÁCE

**prof. Ing. ADAM HEROUT, Ph.D.**

BRNO 2017

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav počítačové grafiky a multimédií

Akademický rok 2016/2017

**Zadání bakalářské práce**

Řešitel: **Kovařík Michal**

Obor: Informační technologie

Téma: **Knihovna pro multiplatformní vývoj mobilních aplikací**

**Library for Multiplatform Development of Mobile Apps**

Kategorie: Uživatelská rozhraní

**Pokyny:**

1. Prostudujte a popište problematiku vytváření multiplatformních mobilních aplikací, včetně tvorby tenkých klientů.
2. Definiujte požadavky na nástroj/knihovnu/framework pro tvorbu multiplatformních mobilních aplikací.
3. Implementujte důležité části navržené funkčnosti.
4. Vytvořte demonstrační aplikace demonstrující vytvořené řešení a umožňující vyhodnocení jeho vlastností.
5. Vyhodnoťte vlastnosti vytvořené knihovny.
6. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte plakátek a krátké video pro prezentování projektu.

**Literatura:**

- dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 a 2, značné rozpracování bodů 3 a 4.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese

<http://www.fit.vutbr.cz/info/szz/>

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

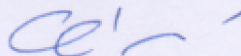
Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Herout Adam, prof. Ing., Ph.D., UPGM FIT VUT**

Datum zadání: 1. listopadu 2016

Datum odevzdání: 17. května 2017

**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**  
Fakulta informačních technologií  
Ústav počítačové grafiky a multimédií  
602 00 Brno, Bc Zetěchova 2



---

doc. Dr. Ing. Jan Černocký  
vedoucí ústavu

## Abstract

This thesis addresses the issues when developing mobile applications for multiple operating systems and development environments, with the target being to create the ideal user interface library. A framework for HTML app development has been designed and implemented that is built on top of modern web standards, allowing the developer to create applications with a single codebase that will, when deployed, intelligently adapt to the device and operating system that they are being run on. Released as an open-source project and currently supporting Windows 10, Android, Chrome OS and Web. Flexus, a framework for building user interfaces, is in live use and active development.

## Abstrakt

Tato práce se zabývá potížemi s vývojem mobilních aplikací pro vícero operačních systémů a vývojových prostředí, s cílem vytvořit ideální knihovnu pro tvorbu uživatelských rozhraní. Na základě moderních webových standardů byl navržen a implementován framework pro vývoj HTML aplikací, umožňující vývojářům snadné vytváření aplikací s jednotným zdrojovým kódem, které se samy inteligentně přizpůsobí zařízením a operačním systémům, na nichž jsou spuštěny. Zřejmě coby open-source projekt, současně podporující Windows 10, Android, Chrome OS a Web, je Flexus frameworkem návrhu uživatelských rozhraní v aktivním užívání a nadále vývoji.

## Keywords

Material Design, Universal Windows Platform (UWP), Microsoft Design Language (MDL), Neon Design, Android, Windows, HTML, CSS, JavaScript, Framework, Library.

## Klíčová slova

Material Design, Univerzální Platforma Windows (UWP), Designový Jazyk Microsoft (MDL), Neon Design, Android, Windows, HTML, CSS, JavaScript, Framework, Knihovna.

## Reference

KOVAŘÍK, Michal. *Library for Multiplatform Development of Mobile Apps*. Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Adam Herout, Ph.D.

# Library for Multiplatform Development of Mobile Apps

## Declaration

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. Ing. Adama Herouta, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....  
Michal Kovařík  
May 18, 2017

## Acknowledgements

Děkuji panu prof. Ing. Adamu Heroutovi, Ph.D. za jeho odborné vedení této práce, dále děkuji všem, kteří mi poskytli pomoc a korekci k tomuto dokumentu, jmenovitě David Smith, Emma Olivia Pedersen, Keifer Lucchi.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Current State of Apps and User Interfaces</b>	<b>4</b>
2.1	Platforms and runtimes . . . . .	4
2.1.1	Android . . . . .	4
2.1.2	iOS . . . . .	5
2.1.3	Windows – UWP . . . . .	5
2.1.4	Web . . . . .	5
2.1.5	Chrome OS . . . . .	5
2.1.6	Other platforms . . . . .	6
2.2	Anatomy of an Application . . . . .	6
2.3	Design Languages . . . . .	8
2.3.1	Material Design . . . . .	8
2.3.2	Neon Design Language (Microsoft Design Language) . . . . .	9
2.3.3	iOS Human Interface . . . . .	10
<b>3</b>	<b>Goals of the Framework</b>	<b>11</b>
3.1	Flexus Modules . . . . .	12
3.2	Code Simplicity . . . . .	13
3.2.1	Custom Attributes . . . . .	14
3.2.2	Comparison to Other Existing Libraries and Frameworks . . . . .	15
3.2.3	Customization . . . . .	18
3.3	Bridging Design Languages . . . . .	20
<b>4</b>	<b>Implementation</b>	<b>24</b>
4.1	Optimizations for Variety of Screens and Devices . . . . .	24
4.1.1	Scaling . . . . .	24
4.1.2	Responsivity . . . . .	24
4.1.3	Sizing and Spacing . . . . .	25
4.1.4	Touch vs. Mouse . . . . .	25
4.1.5	Composition . . . . .	27
4.2	Pixel Perfect Implementation of Design Languages . . . . .	27
4.3	Optimization Compromises . . . . .	30
4.4	Experimental Standards . . . . .	30
4.4.1	Shadow DOM . . . . .	31
4.4.2	CSS Custom Properties . . . . .	32
4.5	Modularity . . . . .	33
4.6	Performance . . . . .	33

4.6.1	DOM Manipulation . . . . .	34
4.6.2	GPU Acceleration . . . . .	34
4.6.3	Caching . . . . .	35
4.6.4	Scrolling with Passive Listeners . . . . .	35
4.6.5	Scheduling the Browser's Animation Frame . . . . .	36
<b>5</b>	<b>Evaluation for Real World Usage</b>	<b>37</b>
<b>6</b>	<b>Conclusion</b>	<b>40</b>
	<b>Bibliography</b>	<b>42</b>
<b>A</b>	<b>Content of the DVD</b>	<b>44</b>
<b>B</b>	<b>Demo applications</b>	<b>45</b>

# Chapter 1

## Introduction

Any software development with the intention of reaching a wide audience spanning multiple platforms and form factors has always been difficult due to the restriction in programming languages and APIs available for each of the platforms. And of course, the user interface has to be adjusted for various screen sizes and, preferably, even each operating system's specific look and rules. In addition to that, the internet has become increasingly important and, in order to stay relevant and broadly available, services and companies now have to provide users with not only a mobile application or a program for desktop computers but also a website or rather a full web application providing the same functionality.

Development efforts are being scattered across multiple separate apps that are concurrently being built from the ground up to fit each platform's needs. However the landscape of app development has undergone immense changes in the past couple of years and, thanks to the popularity of web platforms, languages that have previously been only used to create a website can now be used to build mobile applications as well. Given that every major operating system today has a built-in web browser, the first prerequisite is met. However, when it comes to creating user interfaces that are responsive to numerous screen sizes and input types, things start to get complicated. There are many open source UI libraries available to use, which are mainly designed for creating websites, not applications, and if so they only support a single design language for specific operating systems.

This thesis provides insight into the current state of application development and aims to remedy the situation by creating a framework for HTML development, called Flexus, that aims to solve the aforementioned problems by providing developers with a set of building blocks for designing a user interface that automatically scales and adjusts, to various screens and platforms with minimal effort. A tool, implemented in JavaScript, on top of modern web standards, designed with simplicity in mind and that takes away the hard and repetitive work from a developer's hands, resulting in clear, customizable and easily maintainable application source code.

## Chapter 2

# Current State of Apps and User Interfaces

The landscape of software development has shifted dramatically over the past couple of years, towards mobile devices such as phones and tablets, due to their increasing popularity and availability. These devices are usually used without any peripherals and are generally controlled using a touchscreen interface. As such they are primarily optimized for touch input causing text, buttons, and the whole user interface in general to be large enough for fingertips to control easily. Often programs, also called applications or apps for short, written for these platforms lack some functionality as the main focus is to be simple and clear. Their development, however is not as trivial as one would hope. There are currently four major platforms, each of which vastly differs both visually and in runtime, supporting only certain programming languages for the apps to be coded in. Beyond that developers are presented with user interface libraries that provide a basic set of components for building a GUI. These libraries are, of course, based on each platform's programming language of choice and coding style that they enforce varies. This leaves developers with app code that is non-reusable across other platforms and results in writing and testing the app multiple times, from scratch, for each of the platforms.

## 2.1 Platforms and runtimes

### 2.1.1 Android

Android (by Google) is currently the world's leading mobile operating system, holding a majority market share. The primary programming language is Java, aided by XML for configuration files and is confusingly bloated, repetitive and has extensive layout definitions. Although web technologies are not supported for native application development, it is possible to create an empty Java app covered with a Webview component, which then hosts the HTML and JS files, effectively creating a native web application. This Webview shares its engine with Google's web browser (Chrome) and is regularly updated. Making it an attractive development target, mainly with the help of the Cordova project, simplifying this process.



### 2.1.2 iOS

iOs is Apple’s smartphone and tablet powering operating system which, much like Android, prefers custom languages (Objective-C or Swift) over web technologies for native application development. However, the same Webview approach, as with Android, can be used.

### 2.1.3 Windows – UWP

Windows has traditionally been a desktop only system that eventually branched out into the smartphone sector with an overhauled interface. Version 8 of the system started to bridge differences by bringing touch enabled apps to its desktop counterpart. With Windows 10, this plan for a converged OS came to fruition thanks to a unified design language for interfaces that works well across all screen types. Along with two separate runtimes capable of hosting apps powered by variety of programming languages. It is known as Universal Windows Platform or UWP for short.

Firstly there is a runtime centered around Microsoft’s traditional C# and XAML languages. Secondly, and for my thesis more importantly, there is a second runtime (based on the EdgeHTML and Chakra engines) which is essentially the Edge web browser enhanced with the same system APIs provided to the C# runtime as well. This allows the creation of native applications with web technologies. Edge (both runtime and browser), which is updated twice a year alongside the system, supports most of the modern ECMAScript and HTML5 web standards and can be just as high performing as the C# app runtime since both underlying web engines are greatly optimized. Microsoft also provides the WinJS library for building UIs. However, it did not become very popular among developers due to its strange and complicated code style, making it challenging to understand and code.

### 2.1.4 Web

The Internet, once a text document sharing tool, grew in popularity with both users and developers. Websites slowly turned into web applications capable of what previously could have only been done in a desktop program thanks to new standards regularly extending HTML, JavaScript and CSS languages.

Most of the mobile apps today come with a web counterpart offering that has the same features through web browsers, usually powered by some MVC framework taking care of data rendering and navigation without actually reloading the whole web page. This paradigm is called a Single Page Application and, with new standards, under the name Progressive Web Apps [7], enables offline access and “pinning to the home screen” by storing all of the application’s files and placing icons beside native applications [5]. It then operates without connection to the internet and suppresses the browser’s URL bar, leaving the web app looking and behaving like an actual application without the need of installation. The requirement for this is the creation of a manifest.json file [8].

### 2.1.5 Chrome OS

Google’s other operating system is an extension of the existing Chrome browser, aimed at laptop users who spend most of their time browsing the internet, and as such is solely based on web technologies. The only available languages for development are HTML, CSS, and JavaScript, currently, applications must be published to the Chrome Web Store using a

separate manifest.json file, although it is expected that the PWA standard will be favored in the future being as it is already implemented in Android.

### 2.1.6 Other platforms

As previously mentioned, Android and iOS do not currently support web languages as their primary means for application development. However, Webview components can be utilised for hosting HTML, CSS, and Javascript. This has been popularized by the Cordova project <sup>1</sup>, formerly known as PhoneGap, which automates the creation of such Webview shells on mobile operating systems: Android, iOS, and Windows.

And, much like Cordova, two open source projects NW.JS <sup>2</sup> and Electron <sup>3</sup> combine Chrome's rendering engine (Blink) with deeper access to the system and removing sandbox limitations of typical websites. Bringing web programs for desktop OS's (MacOS, Linux, Windows).

## 2.2 Anatomy of an Application

Despite visual differences, the essential structure stays similar on most platforms. The basis for every screen in the application is what I, for the lack of a better term, call View. It is a container for a single state, or unit of content, with a unique set of functionality, taking up whole screen of a phone. Every View has content in the middle, taking up the majority of the space. This could be anything from plain text to photos, or an interactive component like buttons and checkboxes. The other important part of a View is usually, but not necessarily always, a heading, navigational icon button leading to a previous View, and an additional actionable button(s). These are all grouped into a single component called the Toolbar, also known as the AppBar or the NavBar due to naming diversity.

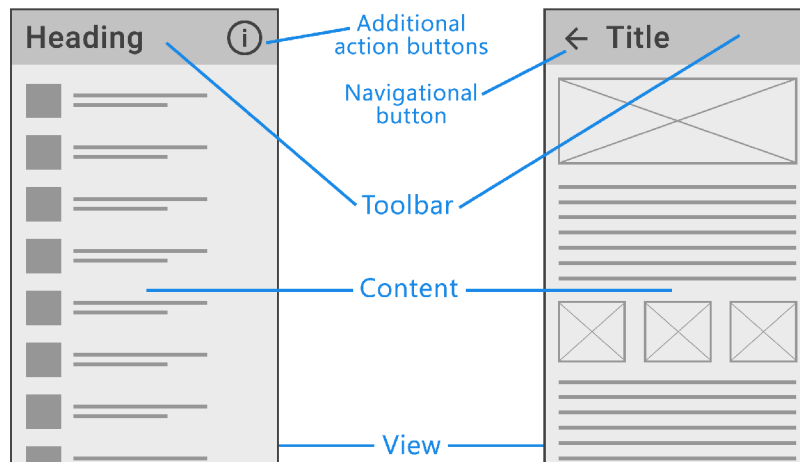


Figure 2.1: Typical View structure.

An app is essentially a collection of mutually interconnected Views and the user can navigate from one to the other. They do this by either going hierarchically deeper to access more information in the current scope, for example from list of contacts into a detail page

<sup>1</sup><https://cordova.apache.org/>

<sup>2</sup><https://nwjs.io/>

<sup>3</sup><https://electron.atom.io/>

of a single contact. Or by going to another View on the same level of flat hierarchy, for example from the home screen to a settings page.

For the latter type of navigation both the Neon and Material design languages use a side panel, known as the Drawer or the Navigational Drawer, which slides into the screen from the left edge upon clicking a so-called “Hamburger menu”, an icon of three horizontal lines that is located on the left side in Toolbar of top level View. The same space is then occupied by a left facing arrow or an X to close and return to the previous View, up in the hierarchy.



Figure 2.2: Navigational Drawer and its appearance on small and large displays.

In addition to primary navigation, a View’s content can also be split into sub sections that are navigable using a Tabs component that is either directly part of the Toolbar or immediately after it, visually blending into the rest of the content.

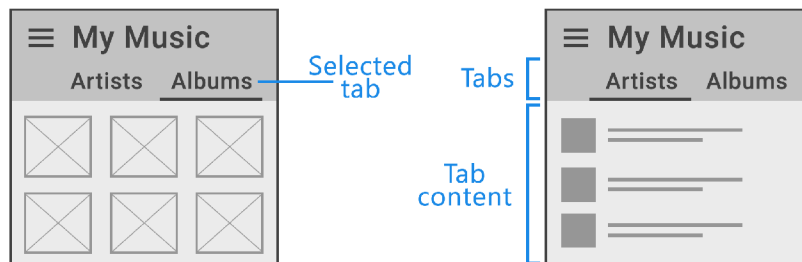


Figure 2.3: Single View with content separated using Tabs component.

Views are primarily designed to allow small screens to display all core functionality but it can and should expand from there to account for larger screens by showing, not only more content, but possibly changing the application’s layout. Most used is the master-detail pattern which is applied to two Views in a hierarchical parent-child relation where a single master View leads to many detail Views. Take, for example, a contact list, where the list is the master and always visible on the left half of the display, the right side is then occupied by details of individual contacts. Or a list of folder contents with additional actions hidden in a side panel, sliding from under the right edge on phones, but permanently visible on tablets where surface area is large enough for both.

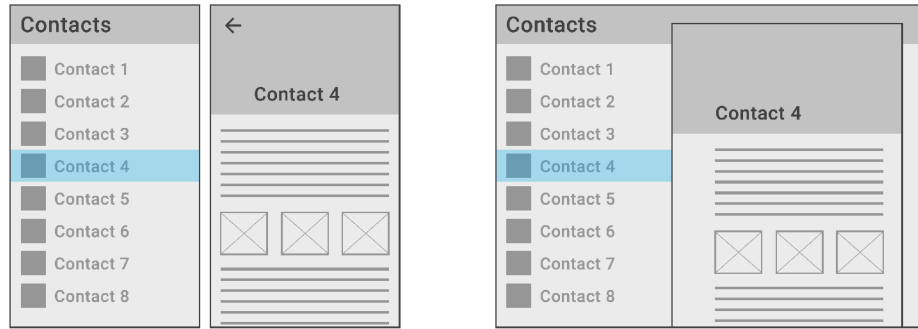


Figure 2.4: Two Views in Master-Detail relation as displayed on small and large screens.

Despite structural similarities, there are still a lot of differences between platforms and their design languages. Vague oversimplification often leads some developers to believe that they could simply port their existing application 1:1 to another platform without any design changes. Such approach to multiplatform development is frowned upon since platforms actually vary not only visually but also behaviorally. This leaves users confused because they come to expect every app to look similar and be controlled in the same way. A prime example is navigation. The design of iOS places navigational Tabs on bottom of the screen whereas Material Design (Google) tackles this with the Drawer, a vertical panel sliding from the left side of the screen by dragging the finger from the edge of the screen inwards or by clicking the “Hamburger” button in the Toolbar. These memorized steps are crucial for less experienced users that rely on the presence of these interactions in every app. Porting an iOS application to Android without first updating the interface to match the Material Design specifications can lead to confusion due to the absence of the aforementioned drag-in gesture for opening the navigational drawer.

## 2.3 Design Languages

Each platform devised a distinctive visual look to differentiate itself from the others. These looks are called Design Languages because it, just like natural human languages, follows syntax and rules where words are replaced with interface components. These languages are described in extensive documents known as Design Guidelines which define everything from fonts and colors, sizing, spacing, and placement of components, up to wireframes and examples of common mistakes to be avoided.

### 2.3.1 Material Design

Developed by Google for Android OS and web services, it is a bold new step that places emphasis on color, imagery, transitions, simple shadow effects in three dimensional space and rounded edges of boxes called Cards encapsulating the content. The layout is very spacious since it was primarily designed for touch devices, but extends beyond to Chrome OS laptops without touchscreen where sizing of a few components is reduced.

The visuals are an analogy to real world where all elements are made out of material, hence the name, and that is paper. Just like in real world, these sheets of paper can be placed next to each other creating a seam, stacked over each other casting a shadow, moved around, stretch or collapse and change color. Unlike other design languages it capitalizes on animations and transitions and bold visuals. Great example is the Toolbar component

that, unlike in other conventional design languages, goes beyond just a thin strip on top of the screen. It can hold images, stretch height, react to scroll and become more prominent part of the application's layout on larger screens.

Material Design Guidelines is a document extensive in both small details, like exact measurements of elements and their sizing on different screens, application layout, composition, responsiveness, terminology as well as guidance on customization or forbidden patterns to avoid. This combined with the flexibility and customization is the reason I chose it as a basis for Flexus.

### 2.3.2 Neon Design Language (Microsoft Design Language)

With Windows 10, Microsoft has focused on creating a balanced design that could work across various form factors and input methods. This poses a problem since desktop programs are usually delivered with lots of information, labels and lines upon lines of text thanks to the precision that mice offer. A cluttered interface is not the best to begin with, however the overly spacious Metro from Windows 8 was also not ideal. Neon strikes a balance by delivering clickable areas that are just large enough to be easily reached by fingers on a touchscreen but not so large as to feel empty and space-wasting on classic desktop PCs.

Visually, the language is fairly simple, compact, flat and without any special effects or shadows, with the exception of transparency. The visuals are dominated by sharp edges, straight lines and flat filled areas, using black or white upon which lies text and heading in large but thin Segoe UI font. Regular text size is 15px for optimal readability but headings are large and distinctive. The overall layout and spacing is generally more condensed, making it comfortable for mouse manipulation but actionable and clickable areas are at least 32 pixels tall or wide, allowing it to be large enough for touchscreen devices. A notable part of some applications is the navigation drawer, a vertical list of icons with text, similar to that found in in Material Design, only here it is placed to the left hand side of the application, almost exclusively, in compact thin form. It expands upon clicking the "Hamburger button". Similarly, this inspiration was drawn upon for the toolbar that hosts frequently used actions, represented by icons.

This design language might seem a little vague in it's appearance, but the new category of hybrid devices with touch screens and detachable keyboards is where it truly shines. Users of such devices have the comfort of using a precise mouse cursor but also the ability to utilize gestures that are tailored for physical interaction with the screen with their fingers.

At the time of writing, this particular design language is undergoing another shift, or rather extension. Microsoft is currently experimenting with new blur effects, colorful toolbars bearing larger text that is also reactive to scrolling. Basically bringing this design language more in line with Material Design. These new changes can already be seen in the pre-installed Windows 10 applications such as Movies & TV and Groove Music. More are scheduled to follow with next update in the fall.

This visual refresh is known as Project Neon and I also chose to reference it as such, rather than MDL. This is to prevent ambiguity because the M could potentially refer to either Microsoft or Material.

Unfortunately, the design documentation is limited and Microsoft provides very little guidance on basic measurements, sizing or spacing. This forces my implementation of this design language in Flexus to be an approximation, based on existing Microsoft's (and third party) applications for Windows 10. I downloaded, studied and measured a number of

these applications to ensure as close a match as possible. Additionally, researching Neon changes are considerably difficult at this undocumented stage. I did, however, manage to dissect Microsoft's WinJS library, which allowed me to determine the proper sizes of fonts and toolbar components.

Despite its many challenges, I chose Neon for the challenging yet forward thinking interface that traverses a large variety of different devices.

### **2.3.3 iOS Human Interface**

Due to the popularity of its devices, iOS has garnered a massive user base. Unfortunately, Apple does not provide any real information into their design guidelines. Detailing mostly large white planes that host a thin, uniform, toolbar on the top of every view, leaving the developer with the freedom to express his or her own style. Due to the iOS design guidelines and the relative complexities involved in supporting Material Design and Microsoft Design Language, porting the best features of both over to an iOS application in the future should pose no issues. However, when considering support for iOS, I found that I needed to reverse engineer the design specifications (sizing, spacing etc) themselves. As the development environment for iOS requires an iPhone and Macintosh, which I do not have available to me, I was unable to consider adding iOS support into Flexus at this time.

## Chapter 3

# Goals of the Framework

Many existing platforms, languages and component libraries are vastly different and incompatible so committing to these environments, precludes code sharing and reusability across other platforms. One solution is the use of web technologies and, although much progress has been made in the creation of an environment for native HTML applications, it remains unpopular due to the lack of interest of the platforms authors to provide meaningful tools and user interface building blocks because they usually prefer a platform's primary programming language. The main argument for this is, of course, performance, as native languages are optimized to run closely tied to the hardware whereas Webview introduces another layer of abstraction. I, however, hold the opposite view. Smartphones were conceived as a multipurpose device, one of these purposes being a web browser. Their popularity sparked a wave of optimization amongst vendors on one side and web developers, more strictly following good practices of development, on the other. Smartphones are becoming increasingly faster with four core CPUs and GPUs capable of intensive 3D graphics. With new deeper level APIs granting developers access to GPU acceleration, this makes HTML-based development performant and viable. The final missing piece is a library for creating compelling user interfaces that would scale and adjust.

This is where Flexus steps in with four core principles.

- A single application codebase that automatically adjusts to two design languages, various screen sizes, input types and platforms.
- Opinionated, yet extremely simple code.
- Powerful and extensible when required, but intelligently inferring when not.
- Precise implementation of Material Design and Neon languages.

Flexus is a framework for building user interfaces in HTML, CSS and JavaScript languages. It is not just a library that only provides sets of components. It is a framework that cements itself into the language by introducing new elements for scaffolding the application's layout, while allowing developers to reuse all existing HTML tags e.g. `<button>`, unlike others tools like Paper Elements library <sup>1</sup> which enforces the use of custom replacements, `<paper-button>` for example. These elements are self configurable and aware

---

<sup>1</sup><https://www.webcomponents.org/collection/PolymerElements/paper-elements>

of context, platform, size, input type, they provide robust customization, most notably `<flexus-toolbar>`, and their inner code does not leak into application code.

The visuals of the application are defined by two CSS files with the implementation of both Neon and Material Design that can be swapped as required, that allows developers to meet the design specifications for the platform that the application is launched on. Fonts and icons are included, as well as a wide color palette that can be utilized for styling and customization. The layout of the application is automatically adjusted to screen type, size, orientation, effects disabled on slower devices or when the battery is running out, and more.

Simplicity was key for designing this framework, which could be used to create applications with next to no additional setup. For example `<flexus-tabs>` element can be created to control currently shown page in `<flexus-pages>` and both elements are capable of finding and attaching themselves into each other, without the need of specifying `[for]` and `[id]`, if they are both located within the same `<flexus-view>`. Flexus also delivers substantial built-in behavior that automatically sets up the environment and takes care of the usual hard work associated with repetitive, so called “boilerplate” code that is generally involved in the development of an application. It is simplistic, in terms of a prototype building tool, where output is not just a prototype, but an actual working interface.

This project is an exploration of what the ideal framework should look like, both from a code perspective and, also, technologically. It is built on top of modern web platform features and specifications that are being standardized and implemented in browsers right now (or in the near future). That said, it was always meant as a serious, production facing open-source project and not simply an experiment. Flexus has been released on Github <sup>2</sup> where it is available for download and use. The currently targeted platforms are Android, Chrome OS, Windows 10 and Web. It does this by implementing two design languages (so far) with the intention of expanding to other platforms in the future, namely iOS and MacOS, however these come with specific hardware requirements that are not currently available to me.

When compared to other HTML, CSS and JavaScript based libraries or frameworks developed by large teams, Flexus may, currently, only provide a small amount of what these other frameworks can offer, but what it does, it tries to do correctly.

## 3.1 Flexus Modules

The scope of this framework is relatively wide, so it was necessary to set out the functionalities and to separate them into sub-modules:

- **Flexus UI**

By default, a HTML document has no style with no visual elements and text in the serif typeface. Flexus provides two rigid, comprehensive stylesheet files, one for each design language implemented to a high level of detail and containing styles covering everything from basic elements to complex layout schemes. Most HTML built-in elements like `<button>`, `<input>` text fields, checkboxes, radio buttons, as well as custom components like `<flexus-toolbar>` are automatically styled upon importing the CSS file to a HTML document. Color palette and typography with icons are predefined and properly spaced, including a wide range of custom attributes that can be used to customize any part of the application.

---

<sup>2</sup><https://github.com/MikeKovarik/flexus>



Flexus UI provides everything the application could need when it comes to design, so ideally the developer wouldn't have to write any CSS, unless it's for some detailed customization matter that cannot be solved by provided modifier attributes. These styles can be used standalone, without any other Flexus modules, but then the automatic adjustment to screen size and form factor would be lacking.

- **Flexus Custom Components Library**

Despite respecting built-in HTML elements and encouraging developers to use them, Flexus introduces a host of new custom elements that extend the language and can be used like any other. For example `<flexus-toolbar>`, which is semantically similar to the built-in `<header>`, but has a lot of advanced behavior built in.

These elements are self configuring, context aware and can talk to each other to work for developers with minimal effort. That is ensured by building them on top of new standards called Web Components which allow the enclosure all of the framework's source code into what, from the application's standpoint, appears as a single HTML element, without spilling out of context and causing confusion.

- **Flexus Core Support Library**

As the name implies, the support library has many objectives required for the proper functioning of other modules or the framework as a whole.

- Collection of classes, decorators and helper functions that are shared, used or inherited by the custom components classes.
- Platform detector gathers information about operating system, runtime or browser, screen size, input type, device form factor, performance and battery.
- Fills in missing meta tags, configuration, ensures proper scaling, disables zooming and overall prepares the environment for application development.
- Loads polyfills for missing platform features, if they're not included
- Loads proper design language if it's not defined
- Adjusts look and behavior.
- Disables or re-enables device processor intensive effects, based on device and battery status.

- **Ganymede**

At the beginning of writing this thesis, there were two unfinished and partially implemented versions of Web Components standards [17]. Ganymede was created as an abstraction layer from this inconvenience and later turned into a lightweight library. It handles the registration, creation (and later destruction) of component and mapping instance properties to DOM attributes of the element.

## 3.2 Code Simplicity

HTML is a very expressive language that allows programs to be written in declarative manner. Being built on top of it, Flexus framework shares some of that language's traits and introduces opinionated concepts. It guides and sometimes enforces the developer to write code in a particular way, but the tradeoff is much simpler application code.

### 3.2.1 Custom Attributes

The way HTML elements are traditionally styled is by using classes or IDs. Classes supplied by UI libraries could be used with custom styles written by the developer, but this way is prone to indistinguishability.

```
<h2 class="text-primary custom-heading">
```

Two classes are used in this common example: `.text-primary` is Bootstrap library class for adding color, `.custom-heading` is hypothetical class by the application's developer to further enhance look of the element.

HTML elements can also contain additional information in form of attributes. Only few are built into the language by default, the most known are `[disabled]` for actionable form elements, or `[type]` that changes behavior and look of the universal `<input>` element to be either a text field `<input type="text">`, checkbox `<input type="checkbox">`, or a radio button `<input type="radio">`. And while it was able to query attributes using CSS, it was not a popular styling option among developers up until widespread adoption of Web Components standards which reinvented what a good and bad practises for the language are, styling attributes being one of them. Part of Flexus is a set of custom components that use attributes as a means of configuration of the element as well for styling since it would be impractical and inefficient to duplicate parts of the code as a CSS class. Equally impractical would be requiring use of classes for built-in elements and attributes for custom elements.

Design decision has therefore been made to utilize attributes everywhere, not only in custom elements, in order to fall in line with the philosophy of cementing Flexus into the language, giving a notion of what HTML for application development always should've been, because use of attributes implies built-in behavior which is exactly what Flexus does. That way there's a clear distinction between developer's custom code and the rest of the environment and third party tools.

One of the new attributes is `[tinted]` for applying color to elements. With it, the previous example could therefore be rewritten as:

```
<h2 tinted class="custom-heading">
```

This code immediately indicates that the class is supplied by developer to customize heading which has additional styling built into it.

Another example is the use of HTML's built in `[hidden]` and `[disabled]` attributes. The navigational drawer `<flexus-drawer>` automatically adds and removes `[hidden]` to itself upon opening or closing and it could be used by the user as well to configure that it should be hidden by default. On the other hand `<flexus-tabs>` uses `[disabled]` on its children to denote the inaccessibility of a specific tab.

```
<flexus-tabs>
  <a href="...">Speed dial</a>
  <a href="..." disabled>Recent calls</a>
  <a href="...">Contacts</a>
</flexus-tabs>
```

This approach is much friendlier as it recycles pre-existing and well known concepts instead of artificially creating new ones.

### 3.2.2 Comparison to Other Existing Libraries and Frameworks

The following code is an example of a simple button with a star icon and the text „Favorite“ inside it, using the Material Design Lite library, by Google.

```
<button class="mdl-button mdl-js-button mdl-button--raised">
  <i class="material-icons">star</i> Favorite
</button>
```

As can be seen, the HTML built-in `<button>` tag is used to create the button which would, however, lack any style that the library is supposed to provide. Without the additional classes where `mdl-` is a namespace, `mdl-button` and `mdl-js-button` are the mandatory classes containing the default look and, finally, the `mdl-button-` further deepens the scope to access only button related style modifiers of which `raised` adds a shadow around the button. Additionally `mdl-button-primary` can be used to embellish the element with the theme’s primary color.

Twitter’s Bootstrap doesn’t cause as much clutter thanks to omitting the namespace but retains scopes. In this case `.btn` for button related styles which leaves this solution far from perfect with implicit `.btn` and `.btn-default` classes.

```
<button type="button" class="btn btn-default">
  <span class="glyphicon glyphicon-star"></span> Favorite
</button>
```

WinJS by Microsoft follows the same pattern but lacks a named icon system so entity code has to be inserted instead.

```
<button class="win-button">&#xE734; Default button</button>
```

Unopinionated libraries sound good in theory, however practice presents the developer with repetitive boilerplate code. Combination of multiple libraries ends up producing hard to maintain code for what should’ve been only a simple, colored button with icon and text in it. Following example is an amalgamation of MDL and WinJS code to make the same code adjust to both design languages. It clearly is not an optimal way for creation of just a single button.

```
<button class="win-button-primary mdl-button mdl-js-button mdl-button--raised">
  &#xE734; <i class="material-icons">star</i> Favorite
</button>
```

Besides MDL, Google also develops Polymer and Paper Elements library build on top of Web Components standard, just like Flexus, but it strangely presents custom replacement elements for those already built-in to HTML. Instead of native `<button>` developers are forced to use `<paper-button>`. It has to be imported to the document since it’s defined in external javascript file, causing unnecessary overhead as additional javascript code runs behind every button in the app. Not to mention that icons are only available through another custom element `<iron-icon>` as seen in this example:

```
<paper-button raised>
  <iron-icon icon="star"></iron-icon> Favorite
</paper-button>
```

Flexus can be marked as opinionated since it, by default, applies styles globally to all appropriate elements. Therefore buttons will always look consistent, based on the design spec of hosting platform with no additional code to what developers have already come to know. The button from the previous examples can be only written using the following code where `[raised]` adds shadow in Material Design and `[icon]` uses CSS pseudo-elements to

inlay the icon glyph without the need of additional nested icon-hosting elements as with other libraries.

```
<button raised icon="star">Favorite</button>
```

A single button with an icon can be used as a small example, the crucial differences and simplicity starts to show up in the code that more accurately depicts an actual application. Every application usually has a Toolbar.

WinJS dynamically enhances elements into so called controls, but it does so with every child. Not only does it require additional code but also causes unnecessary performance overhead. Simple toolbar with single button written in WinJS looks as follows:

```
<div data-win-control="WinJS.UI.Toolbar">
  <button data-win-control="WinJS.UI.Command"
    data-win-options="{
      Type: 'button',
      icon: 'edit',
      label: 'Edit'
    }"></button>
</div>
```

New configuration attribute `data-win-options` is introduced, leading to steeper learning curve. The same toolbar can be written in Flexus on just a three lines:

```
<flexus-toolbar>
  <button icon="edit">Edit</button>
</flexus-toolbar>
```

More realistic toolbars are much more complicated, as in the following example taken directly from Onsen UI website [9]. It has four icons with a heading in between. That should be six elements in total, with the toolbar included. Yet the Onsen UI, much like most other similarly targeted multi platform frameworks do not provide the same comfort that I envisioned for Flexus.



Figure 3.1: Basic Material Design toolbar with a title and a few buttons.

```

<div class="navigation-bar navigation-bar--material">
  <div class="navigation-bar__left navigation-bar--material__left">
    <span class="toolbar-button toolbar-button--material">
      <i class="zmdi zmdi-menu"></i>
    </span>
  </div>
  <div class="navigation-bar__center navigation-bar--material__center">Title</div>
  <div class="navigation-bar__right navigation-bar--material__right">
    <span class="toolbar-button toolbar-button--material">
      <i class="zmdi zmdi-search"></i>
    </span>
    <span class="toolbar-button toolbar-button--material">
      <i class="zmdi zmdi-favorite"></i>
    </span>
    <span class="toolbar-button toolbar-button--material">
      <i class="zmdi zmdi-more-vert"></i>
    </span>
  </div>
</div>

```

As can be seen in previous snippet which recreates Fig. 3.1, not only does Onsen UI force developers to write verbose classes like `.navigation-bar-material__center` and `.navigation-bar__center`, that are arguably confusing due to the combination of variety of dash and underscore delimiters. However, and most importantly, these classes are explicitly only applying the Material Design look, therefore additional and separate code has to be written for another platform's design language.

This example Toolbar can be rewritten in Flexus as follows:

```

<flexus-toolbar>
  <button icon="menu"></button>
  <h1>Title</h1>
  <button icon="search"></button>
  <button icon="heart"></button>
  <button icon="more"></button>
</flexus-toolbar>

```

Not only is this code clearer but also semantically correct, this is because the the icons are meant to be clicked on like on a button. And, in Flexus, they are represented by an actual HTML element `<button>` instead of general purpose `<div>` and `<span>` elements, making it easy for machine processing by screen readers. Notably the title is also represented by the semantical heading element, in this case `<h1>` and since `<flexus-toolbar>` inherits the horizontal [layout], there's no need to wrap content into additional containers like the Onsen UI does with classes `.navigation-bar__center` and `.navigation-bar__right`.

Flexus does not force developers to relate to any particular design language in code. Both provided styles, Material Design and Neon, are built around the same single core so the application code stays untouched but appearance changes accordingly when either Neon or Material CSS files are loaded. The code from previous example is displayed as follows:



Figure 3.2: Toolbar from Fig. 3.1 recreated with Flexus in Material Design and Neon Design.

### 3.2.3 Customization

Simplicity is important but so is customization. Toolbars are one of the most important and diverse components according to the Material Design specifications and, yet, this is where most other UI libraries lack because they think of toolbar as of just a thin strip atop the screen. Toolbars can be rich, include multiple sections, images and be resizable, change color or fade out certain child elements in reaction to scroll. Not to mention that applications usually have search capabilities that are built into the toolbar, transforming its color and layout to reveal the search text-box upon clicking the search icon. None of the currently available HTML frameworks provide this level of customization that the native Android's Java AppCompat style library gives. One of the goals of Flexus was to bring this functionality to life with comparably simple code.

Customization begins by wrapping the original contents of `<flexus-toolbar>` into the `<section>` element and adds a `[multisection]` attribute to the toolbar element. This results in, visually, the same toolbar. From there additional sections can be added, `<flexus-tabs>`, or even images.

```
<flexus-toolbar multisection>
  <section>
    <button icon="menu"></button>
    <div flex></div>
    <button icon="search"></button>
    <button icon="more"></button>
  </section>
  <h2 indent display1>My Music</h2>
</flexus-toolbar>
```

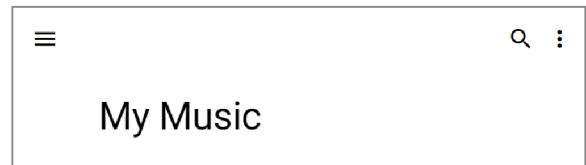


Figure 3.3: Multisection Toolbar created with Flexus.

In this snippet we can see the heading pushed further down. The `<h2>` element was chosen and, while it would be better to have it wrapped in another `<section>` element, this works just as well because both of them are block elements. Additionally, the `[display1]`, one of many typography modifiers predefined by Flexus, was used to increase the size and thickness and `[indent]` to align nicely.

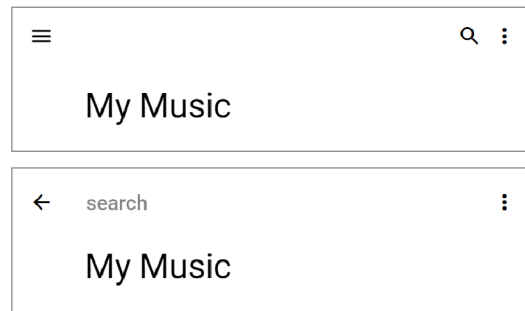
All of these customizations are included with the default CSS stylesheet so they can be used without importing the `elements/toolbar.js` custom component javascript definition file. However, adding it unlocks additional capabilities, the first of which is a quality of life improvement – automatically adding the `[multisection]` attribute.

Some applications host searchable content or a list of selectable items. The toolbar can be used to swap the main section for one that better fits the context. Support for this behavior is built into the `<flexus-toolbar>`. Adding `[search]` to `<section>` makes it hidden and it's revealed only after clicking a button with the search icon, specifically `[icon="search"]`. If the closing button is included inside the section, it will be taken care of as well. This is the most common, default behavior, but the `search-show` or `search-hide` events can be used for customized manipulation. Section with a selection context can be added handled similarly using `[selection]`.

```

<flexus-toolbar multisection>
  <section>
    <button icon="menu"></button>
    <div flex></div>
    <button icon="search"></button>
    <button icon="more"></button>
  </section>
  <section search>
    <button icon="arrow-back"></button>
    <input type="search" placeholder="search">
    <button icon="more"></button>
  </section>
  <h2 indent display1>My Music</h2>
</flexus-toolbar>

```

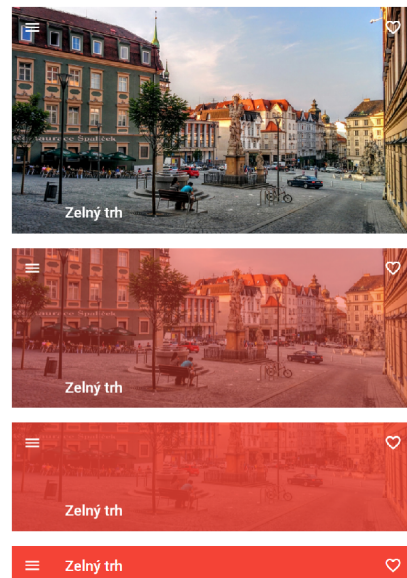


Material Design takes the capabilities of the Toolbar further with reactivity to scroll. Flexus implements this as well with opt-in attributes `[sticky]` and `[collapse]`. Both are mutually exclusive and can be added by the user manually or could be assigned automatically by Flexus' educated guesses. Sections marked with `[sticky]` will remain visible even after scrolling the content it is succeeded by, whereas the remaining sections will be hidden and/or faded out. Alternatively `[collapse]` can be applied to only those sections that are meant to be hidden.

```

<flexus-toolbar multisection>
  <section overlay sticky>
    <button icon="menu"></button>
    <div flex></div>
    <button icon="heart-outline"></button>
  </section>
  
  <section overlay indent>
    <h2>Zelny trh</h2>
  </section>
</flexus-toolbar>

```



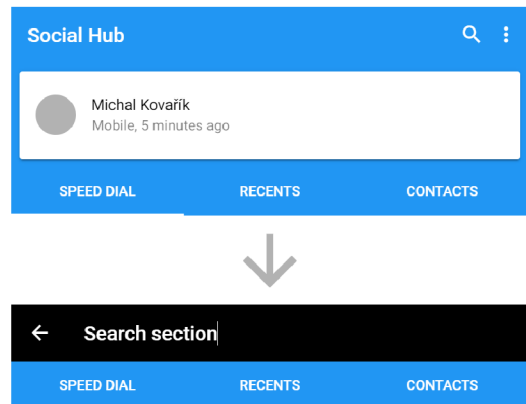
This snippet also shows the use of `[overlay]` which can be used to display the section over the collapsible content with a transparent background. The collapsible content is an image that is also automatically adjusted to fit any width.

These elementary tools unlock many uses like the following example of social application with hidden search section and a collapsible Card showing information about a missed call. Attributes inferred and automatically assigned by Flexus are represented by gray color.

```

<flexus-toolbar multisection>
  <section main sticky>
    <h1>Social Hub</h1>
    <button icon="search"></button>
    <button icon="more"></button>
  </section>
  <section search dark sticky>
    <button icon="arrow-back"></button>
    <input type="search">
  </section>
  <div card collapse fx-item>
    <img avatar>
    <div two-line>
      <div>Michal</div>
      <div muted>Mobile, 5 minutes ago</div>
    </div>
  </div>
  <flexus-tabs sticky>
    <a>Speed dial</a>
    <a>Recents</a>
    <a>Contacts</a>
  </flexus-tabs>
</flexus-toolbar>

```



### 3.3 Bridging Design Languages

Flexus forms a strong opinion on the source application code it is applied to by styling all native HTML tags for the developer without asking. That could be viewed by some as a downside since intrusive opinionated libraries make it harder to be used with other libraries. This however begs the question “Why use other libraries at all?” Flexus answers this by taking care of all of the hard work related to styling by covering most possible elements and design use cases across two distinctive design languages.

All of the application’s styles are compiled into two separate CSS files, one for Material Design and the other for the Neon Design Language, complemented by their respective icon sets. For the application to acquire the visual look, one of the styles has to be included with the `<link>` tag.

```

<link rel="stylesheet" type="text/css" href="flexus/css/flexus-material.css">
<link rel="stylesheet" type="text/css" href="flexus/css/flexus-material-icons.css">

```

Otherwise Flexus will automatically load a relevant design language based on platform the application is launched on. Unless the developer explicitly forces Flexus to load a specific design language by adding either `[material]` or `[neon]` to the `<body>` element. This is particularly useful during development and testing, but using `<link>` tags is recommended for production code to improve load time.

Both implementations are built around a single core which resets the original browser’s style, contains shared utility attributes, mixins, color palettes, and basic application layouts. Onto this core, the design language specific code is added. This makes the two stylesheets interchangeable. However, both are unique in their own way. Flexus tries to tie them together as closely as possible but there are still some specific cases that the developers might want to use to tailor their application for each design language separately.

Material Design Guidelines are a comprehensive visual guide that makes for a great foundation for some of the framework’s design decisions because neither Neon, nor iOS



(which is currently not targeted but it is intended for the future) provide much of any actual information for developers wanting to implement the design, as these guidelines are aimed primarily at designers. Whereas Material Design provides plenty information and, importantly, some naming conventions that could be used instead of reinventing custom terms.

At a first glance the two designs are similar in terms of application composition. They both use toolbars with title and a “hamburger” icon menu that opens up a navigation drawer, but deep down they are different, visually and conceptually. Material design uses a facsimile of real world paper, with layers that can stacked, placed next to each other, slide around and as such the content is enclosed onto these pieces of paper, called “cards” that cast shadow and should not be transparent.

Flexus implements this look in an attribute `[card]` that can be applied to most content containers. One of the simplicity-driven decisions was to create a universal `[card]` attribute that can be applied to wide range of pre-existing elements, instead of a strict new `<flexus-card>` component. Another reason besides simplicity was that Neon does not have this concept of paper and all of the styles applied to `[card]` in Material degrade gracefully in Neon and, what remains, is just a style-less element.

Besides cards, the toolbar is another sheet of shadow casting paper in this analogy. For this purpose, Material Design has fashioned an elevation system in which every component is placed into a 3D space. The further the element is from the plane of other content, the deeper the shadow is. Flexus provides an attribute `[elevation]` that can be used by users, and it is also inherited by some components. `<flexus-toolbar>` has, by default, elevation value of 2. This however only applies to Material design, since, again, Neon does not share this visual concept. So applying `[elevation="0"]` to `<flexus-toolbar>` makes it lose the shadow in Material version but does nothing in Neon.

This becomes more interesting when we consider that the Material paper behavior also introduces concept of “seams”. When the component does not elevate, it descends to the same depth plane as other elements, yet they do not blend together and a separation is visible between them. A seam. This description sounds oddly unique from Material philosophical perspective but, upon closer inspection, the behavior is similar to content separators like HTML’s `<hr>` and is something where Neon already bears some resemblance. But not only that, certain Neon applications like Microsoft Edge have a thin border on the bottom of the toolbar which is exactly the same look as the Material’s version of “seamed” toolbar. This behavior was therefore built into `<hr>` as well as `[seam]` attribute that can be applied to elements. Doing that will also automatically disable any shadows without the need of using `[elevation="0"]` since these two behaviors are mutually exclusive.

This is just one of the examples where these two design languages meet. Flexus is designed to take care of most of the work from small details, up to the composition of the application. Figure 3.4 shows application with two Views in Master-Detail relation and the way Flexus displays it differently in the two design languages. Neon on the right can be seen with two clean columns and a navigation drawer, whereas Material on the left side of the image transforms one of the views into a distinct card laid over the other view.

Besides the visual separation, another element, the `<flexus-scene>` can be used to stack multiple views together. This allows Flexus to change the layout depending on screen the resolution and only showing one at a time if the viewport could not fit them both as can be seen in Fig.3.5.

A big part of the visual appearance is also typography, icons, and color. Material Design uses the Roboto font. Neon Design uses Segoe UI. Both of these fonts are included

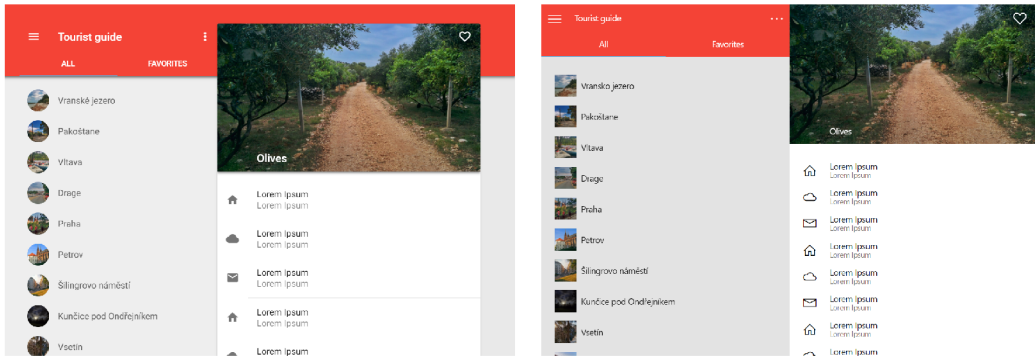


Figure 3.4: Master-Detail application created with Flexus in Material Design and Neon Design as seen on large screens.

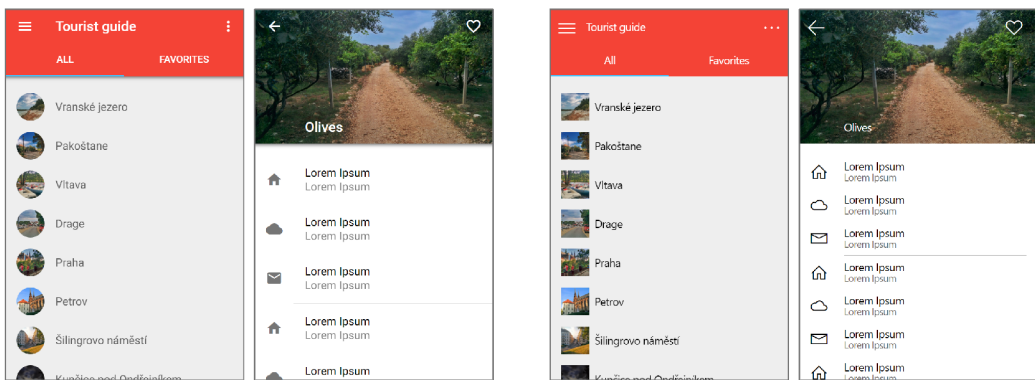


Figure 3.5: Master-Detail application created with Flexus in Material Design and Neon Design as seen on small screens.

with Flexus and dynamically loaded if they are not preinstalled on the platform. Text can be modified with a handful of attribute modifiers such as `[small]`, `[bold]`, `[italic]`, `[underline]` as well as more specific ones like `[display1]` or `[headline]` with higher specificity, just to name a few.

A useful `[icon="..."]` attribute is available to choose from over one thousand named icons, that are mixed and matched from both Microsoft's Segoe MDL2 Assets icon font <sup>3</sup> and an open source Material Design Icons library <sup>4</sup>.

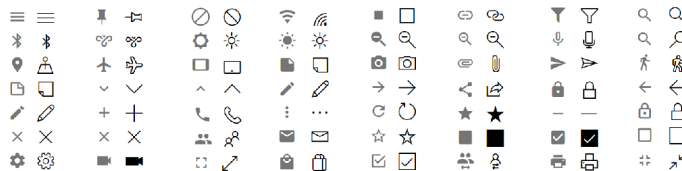


Figure 3.6: Example of differences between Material Icons and Neon Icons

<sup>3</sup><https://docs.microsoft.com/en-us/windows/uwp/style/segoe-ui-symbol-font/>

<sup>4</sup><https://materialdesignicons.com/>

A wide range of colors is also available for deep customization for which Material's color palette [1] and naming scheme is implemented. Every application can have two main colors, of which, the aptly named, primary color covers large surfaces like toolbars. It can be complemented by accent color for details such as check boxes, active tab indicators, etc. This is where the two design languages differ. Material introduces a system of two colors and encourages more vibrant applications, whereas Neon is more subtle with just a single color. It eventually comes down to the developer's taste. Should they decide to use both, they can use two attributes [`primary="..."`] and [`accent="..."`]. These attributes can be used globally for the whole application, per view, or essentially everywhere. They do not change the immediate background of the element. Instead they define the accent and primary tones for all child elements that might use color, within the scope.

With the blue and yellow example, the toolbar would have a blue background and, should it contain tabs or a button, those would be yellow. Proper foreground color is also implemented so the toolbar automatically has white text. This view itself uses a [`light`] theme and the check boxes would usually pick up the accent color, making the yellow hardly visible on the white background. However, in this special case Flexus intelligently omits yellow and uses the primary color instead. This is thanks to cleverly mixing both colors into a so called "adaptive tint". The same code, only with [`dark`] theme used instead, will result in yellow check boxes as can be seen on Fig 3.7. This is all possible thanks to the new standard CSS Custom Properties [10]. That way customization possibilities are endless and the developer is only provided with a simple [`tinted`] attribute. Of course primary and accent colors can be forced with the use of value-less attributes [`primary`] and [`accent`]. [`background="..."`] and [`foreground="..."`] are also available for granular control over the style. If the provided color palette is not sufficient, the developer can simply use custom hex values, such as `<body primary="#F00">` and Flexus will again take care of applying the color everywhere, including providing proper foreground color for text.

```
<flexus-view light/dark primary="blue"
accent="yellow">
  <flexus-toolbar tinted>
    ...
  </flexus-toolbar>
  <main>
    <h2 tinted>Tinted heading</h2>
    <div fx-item icon="home">
      ...
      <input type="checkbox" checked>
    </div>
  </main>
</flexus-view>
```

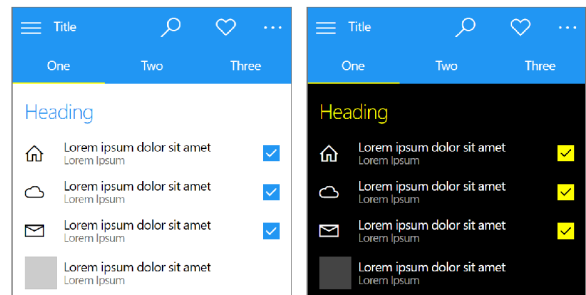


Figure 3.7: Primary and accent color shown on light and dark theme.

Both primary and accent colors (and their tags) can be omitted to leave the application colorless. Flexus, however, integrates more deeply into Windows 10 and can automatically use the local system's theme (light or dark) and color. Similarly, part of the PWA standards is the `<meta name="theme-color" content="#FF0000">` tag which can be used to define color to be used by browser and/or system. Flexus can automatically read from this tag and adjust the application, or vice versa.

# Chapter 4

## Implementation

### 4.1 Optimizations for Variety of Screens and Devices

#### 4.1.1 Scaling

Besides clarity, modern high density displays introduced a new problem, scaling. Most typical desktop monitors have mdpi pixel density, also known as “Pixel Ratio 1.0”, meaning one physical pixel represents one logical pixel. Smartphones are equipped with denser displays, for comfortable use at a much closer viewing distance, where one logical pixel has to be represented by many physical pixels on the screen. For example a 56 pixel tall toolbar takes up about 120 mm on a hypothetical 22” mdpi monitor with pixel ratio 1.0. A phone with an xxhdpi 5” screen that has a resolution of 1080x1920px at a 3.0 ratio would show the same toolbar at around 36 mm unless it’s scaled up, in which case the application perceives the resolution as only 360x640px, where 1 logical pixel is represented by 9 physical pixels on the screen. The toolbar, from a programming standpoint, still remains effectively 56px tall, even though it is displayed on 168 physical pixels, resulting in size of around 110mm, making it roughly the same size in the real world, independent of screen density. Both of the targeted design languages are aware of this and use different units, Density-independent pixels (dp) [3] in Material Design and Effective pixels (epx) in Neon [6], both having the same meaning – logical resolution. Flexus, specifically the Core Support library, takes this into account and automatically configures the proper `<meta name="viewport">` tags so that the developer doesn’t have to. And, like most other features, developers can override this default behavior by simply providing a custom meta tag.

#### 4.1.2 Responsivity

Applications are being developed for the mobile first, displaying one view at a time however, simply stretching out the width is unsuitable for a larger screen. This is where Flexus starts to change layout composition or even begins to display previously hidden elements. Of course both design languages have their own way of handling responsivity with different breakpoints. It had to be simplified into a single unified breakpoint system, with a Metrics Table <sup>1</sup> in mind. The backbone for this are three states that the application can be in, depending on screen width, or window width in case of an OS windowed mode.

- **S – Small**, up to 600px  
Covers all smartphones, shows a single view, spacing is confined.

---

<sup>1</sup><https://material.io/devices/>

- **M – Medium**, between 600px and 1000px  
Most small tablets at any orientation and larger tablets in portrait orientation fall into this category. Still only a single view is shown (with increased spacing). In Neon Design, the drawer is shown in a pinned mode and the toolbar may change in appearance.
- **L – Large**, from 1020px  
Large tablets in landscape orientation fall under this state. Multiple views are allowed to show though only two are being displayed by default in `<flexus-scene>`.

The two breakpoints are defined as CSS variables `-breakpoint-s-m` and `-breakpoint-m-l` in `:root` scope, which makes it easily customizable. The JS Core Support Library picks it up from there and sets up media query listeners that, in turn, applies the `[screenize]` attribute to the HTML tag, together with other platform information, making it easy for granular customization.

### 4.1.3 Sizing and Spacing

Both Material and Neon utilize two different values for padding contents of view, spacing between elements and the density of the entire application. The values differ though, with 16px and 24px in Material and 12px and 24px in Neon. However, it is important to note that they are used in the same manner. Small phone-sized screens make use of the lower values and larger screens utilize the larger values to ensure that the application looks more spacious, clear and comfortable to use. This also assists the “medium” breakpoint. Small tablets occupy this category with screens that are much larger than phones, but not nearly enough to host two views. The layout from the “small” category is therefore used with spacing increased to that of the 24px layout, which makes it slightly upscaled and therefore tailored correctly to the screen size.

These sizing changes can be most notably seen as a padding of toolbars and view’s content. Material Design however includes a few more deviations to this [4]. Neon’s toolbars have fixed base height of 48px, whereas Material varies with 56px on small screens, 64px on medium and large screens, with exception for non-touch devices, where only 48px is used.

### 4.1.4 Touch vs. Mouse

The size of the screen is not the only thing that Flexus optimizes for. Applications can be used on both touch screens as well as with a mouse and keyboard. The physical size of an area the human finger touches on the screen, according to Material Design Guidelines [2], is approximately 9mm. Therefore, the recommendation is to have touch targets that are, at least, 48px to accommodate this. Ideally all buttons would therefore be at least 48px wide and tall but that doesn’t make for an attractive, visually appealing, design. Typical buttons in Material design are 32 pixels tall. Empty buttons hosting only icons are even smaller at only 24 pixels. This is more than enough for mouse operated applications but not ideal for the imprecise touch operation. Scaling up to a larger size would favor touchscreens but cause the application to be impractically large for mouse operated computers.

Flexus, however, cleverly optimizes for it with use of the CSS3 pseudo-elements [12] `::before` and `::after`. These pseudo-elements are not specified in the HTML node tree, they are, instead, specified in CSS and are then injected into the element within DOM. The `[icon]` attribute uses this to inject an icon into any kind of element without the need

of any additional element. For touch enabled screens the Core Support Library adds the [touch] attribute to the <html> element, onto which selectors are hooked by adding the ::after pseudo-element into clickable elements as well. It is absolutely positioned around the element to cover, at least, 48px. This layer is, of course, invisible so that it does not interfere visually but makes the element reactive to clicking beyond its actual size and, thanks to the positioning, the element retains the same physical size so that the layout or spacings are not affected.

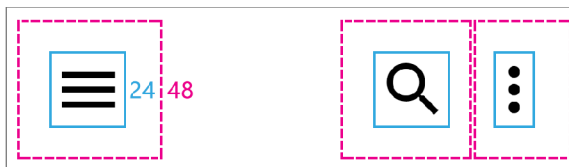


Figure 4.1: Hitbox optimization for touch screens.

Touch enabled devices not only differ in sizing and spacing but also enable other new ways of manipulating and interacting with content. One example is the navigational drawer in Fig. 4.2. A hidden panel that can be opened by clicking a related button, or, in the case of a touchscreen, dragged in from outside the edge. Drag and Drop is the usual pattern for visible components, but in this case the drawer is initially hidden and no portions of it are visible for the finger to hold on to. Here, a very similar approach is used where a thin, invisible strip is displayed on the left side of the screen. Wide enough for a finger to latch onto and initiate the drag, but narrow enough so interference with the application’s content is prevented.

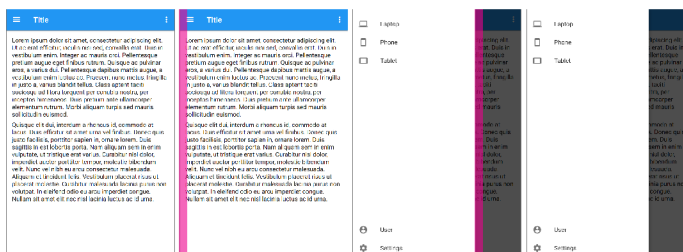


Figure 4.2: Touch screen optimization for the Navigational Drawer using the invisible strip.

Windows 10 offers a feature called “Tablet Mode” for hybrid devices with touchscreens and/or detachable keyboards. It is mostly useful on a system level because it automatically switches the current application to fullscreen and adjusts the system UI when the keyboard is removed. This feature led me to use it as a trigger for a UI change, increasing the sizes of components and spaces between them in the tablet mode. However, even though detection of the tablet mode is possible in UWP’s EdgeHTML runtime, it’s not reliable, and no similar features are available for Android, Chrome OS or Web. This did not prove to be a problem. Strictly condensing the UI in the presence of precision pointer, i.e. mouse or trackpad, is not a perfect solution for hybrid devices. Personal experience and testing demonstrates that touch interactions are likely to be used regardless. More often than not it is more convenient to tap on the screen rather than sliding a finger a couple of times across the trackpad to move the cursor to opposite side of the screen. Flexus therefore makes an

educated guess, based on form factor, screen type and application size, to determine the optimal application sizing and spacing. All touch enabled devices, including hybrids, are therefore displayed in a touch-friendly manner.

Despite this adjustment being automatic, it can be overridden. The Core Support Library adds either `[touch]` or `[nontouch]` to the `<html>` element and developers can specify one or the others to disable Flexus' detection and switching mechanism. Alternatively `[dense]` or `[spacious]` attributes can be added for more granular use. Unfortunately `<html>` has to be used instead of `<body>` which does make it less convenient but ensures the proper value of CSS `rem` unit that is used for sizing of the application.

#### 4.1.5 Composition

The core of each screen is a single view but this only applies to small screen sizes. Instead of leaving it to stretch out, the developer can opt-in for automated composition. The pattern of master-detail can be achieved with the `<flexus-scene>` element, for which a `elements/scene.js` extension must be loaded. Once that is done, this element hooks into the breakpoint responsive system and, upon changes in screen size, it is capable of displaying two views next to each other or just one at a time and transitioning between them.

Besides that, navigational buttons are being hidden or shown accordingly. The master is usually the top-level view with a hamburger menu button in the toolbar, whereas the detail view lies hierarchically below and must have a button navigating back up. That button is also placed in the toolbar but it is only usable on smaller screens where one view is shown at a time. Flexus therefore automatically hides the button with `[icon="arrow-back"]` in a detail's toolbar on a larger screen because there's no need for such a navigation step as seen in Fig. 3.4 and 3.5.

## 4.2 Pixel Perfect Implementation of Design Languages

The visuals of an application are not just the color and shadow effects. Beyond every element lie measurements of height, width, padding, margin, spacing between other elements and set of rules, that may change depending on size and input type. Flexus primarily adheres to Material Design since its design guidelines provide explicit measurements and in-depth coverage of structure [2] and terminology. Most of the concepts are applicable to Neon Design which was also given equal attention. It, unfortunately, does not have as detailed and helpful design guidelines since it is more targeted at designers instead of developers.

Every detail is painstakingly adhered to, ensuring that the implementation of the design language is as complete and true as possible.

One notable point is the indentation line 4.3. This is an invention of Material Design but applies nicely to Neon as well. It is denoted by a secondary line on the left side of the layout, around which text and elements must be aligned if they contain icons, checkboxes, avatars or are modified in any other way causing them to skip the primary line. The primary line is either 16px or 24px, depending on the screen size, from the edge of the screen and serves as an anchor where all of the content is aligned to. After that, additional 56 pixels are reserved for potential content-shifting elements such as the icons previously referred to.

Thanks to CSS variables, Flexus is able to ensure proper compliance of all texts and elements, be they plain or nested, to this indentation line as can be seen in Fig. 4.4. These

variables, `-indent` and `-size`, can be further adjusted, giving developers full customization control over these basic layouts and sizes of icons, checkboxes, avatars, etc.

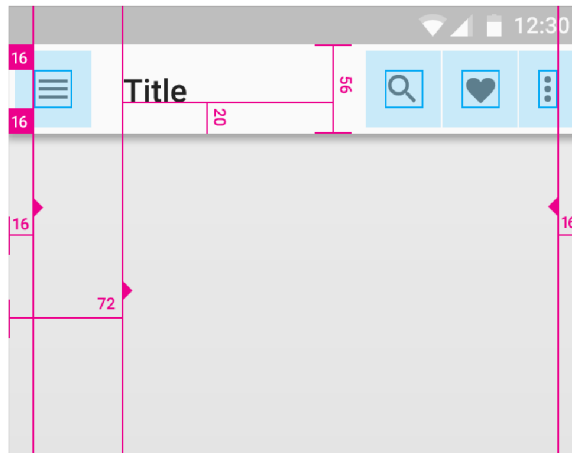


Figure 4.3: Excerpt of measurements behind Material Design's components and layout. Retrieved from: <https://material.io/guidelines/layout/structure.html>

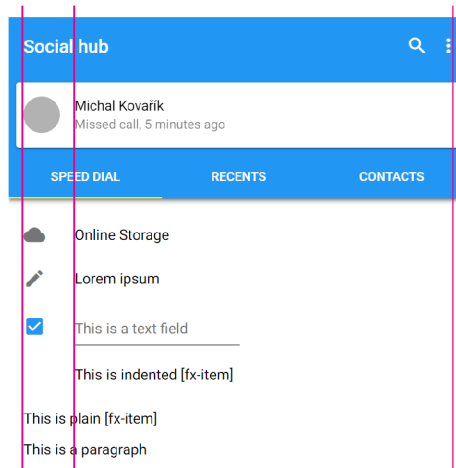


Figure 4.4: Example of a `<flex-view>` with various components aligning around indentation line.

Up until recently, HTML and CSS were not suitable for advanced element composition since the languages were lacking proper layout tools. That was until introduction of the new Flexbox [11] standard, which Flexus uses extensively, and presents developers with the utility attribute `[layout]`. It can be used in combination with the `[horizontal]` or `[vertical]` attributes to create container hosting elements in a single line, aligning them around a single axis and flexibly adjusting their size, relative to other sibling nodes as represented in Fig. 4.5. This greatly simplifies code and, since Flexbox handles children independently of the display mode, it allows for inline-block elements like `<button>` to be flexed next to block element such as `<h1>`. A prime example of this is `<flexus-toolbar>` as most Flexus custom elements inherit `[layout]` behavior, or at least portions of it.





Figure 4.5: Representation of flexbox behavior implemented in `[layout]` attribute.

Most applications consist of lists, whether they are contacts, emails, messages, to-do lists or any other group of items, they typically host an additional icon, checkbox or avatar on the left side, with the rest of it being filled with a single line of text. Flexus therefore provides styles for creating items of such list available under the `[fx-item]` attribute, Fig 4.6. This is a subset of `[layout]` with higher specificity and proper spacing between not only children, but also other sibling items and, most importantly, alignment to the indentation line. Besides just a plain text, lists items often contain icons, checkboxes or images to which `[avatar]` attribute can be added.

Behavior of `[fx-item]` is also inherited by `<flexus-toolbar>` element and its subsections.

The `[fx-item]` was intentionally designed to be an attribute instead of a custom element so that it could be applied to a range of elements such as `<label>` so that the whole item could be reactive to click and control state of checkboxes as in Fig. 4.7.

```
<div fx-item icon="cloud">Online Storage</div>
```



Figure 4.6: Example of a simple, single-line, list item with an icon, created using `[fx-item]` attribute. Material Design variation is on the left, Neon design on the right.

```
<label fx-item>
  
  <div two-line>
    <div>Michal</div>
    <div muted>Missed call, 5 minutes ago</div>
  </div>
  <input type="checkbox">
</label>
```



Figure 4.7: Example of an item with an avatar, two lines of text and a checkbox, created using `[fx-item]` attribute. Material Design variation is on the left, Neon design on the right.

### 4.3 Optimization Compromises

As was already mentioned, Flexus provides developers with a wide range of text, size and color modifying attributes to be used in conjunction with HTML elements such as `[hidden]` for hiding components. These attributes can be used either globally without a value, or with a specifier of where to apply. There are five different specifiers that were cleverly designed to be used together with very little excessive code. Those are `small`, `medium`, `large` for screen size and `neon`, `material` for design. For each of these attributes that are responsive to the specifiers, there are five statements inside the Flexus source code, such as following where `hidden` is the attribute name and star character, instead of direct equality, applies the style rule only when the attribute value contains `small`.

```
[neon] [hidden*="neon"],
[material] [hidden*="material"],
[screensize*="s"] [hidden*="small"],
[screensize*="m"] [hidden*="medium"],
[screensize*="l"] [hidden*="large"] {
  ...
}
```

This provides enough flexibility to combine specifiers, `[hidden*="medium,large"]` applies to medium and large screen sizes, effectively showing the element only on small phones. Due to performance concerns I decided against implementing more complex specifiers. It is tempting to use `small-material` that would only apply to material design phone applications, but each combination of specifiers would have to be hard-coded, resulting in seventeen selectors instead of five, leading up to hundreds of lines of additional code that could potentially cause slowdowns that I wanted to avoid. It is, however, encouraged to go beyond what Flexus has to offer for achieving the true vision that the developers may have for their application by writing custom code, only on a per-app level.

### 4.4 Experimental Standards

Web Components are a set of modern standards that change the core behavior of browser's engines and open up a future for highly composable components. Web Components consist of four separate standards; Custom Elements [13], Shadow DOM [15], HTML Templates [18] and HTML Imports [14] which are currently undergoing the final phases of standardization and are being implemented by browser vendors.

Using unfinished technology in the midst of its standardization is risky, but the foundation has already been laid. Backed up by a positive reception and demand from the community, the initial implementation in Chrome and pre-existing libraries (namely Google's Polymer and their Paper Elements components library) I was confident to move forward and build Flexus around the Web Components standards [17]. It was an important design decision, because it was critical for achieving some of the set out principles:

- Simplicity of the application's code
- Self contained functions of components
- Non-leaking separation of framework's and application's code

This however turned out to be one of the primary problems as well. Chrome (and Android's Webview) initially supported v0 standards as of the start of the development of

this thesis. Over time, v1 has been finally implemented in Chrome and Flexus adapted to it with an abstraction layer for custom components registration that subsequently became the Ganymede module. Unfortunately Edge (and the UWP runtime) does not support Web Components, yet making it much more complicated. There are ways to partially overcome this problem. One of them is the use of polyfills<sup>2</sup> which Flexus automatically loads during runtime of the application if needed. However, this solution is not perfect as the polyfills are only a simulation of the missing features, some of which are impossible to do as they deeply alter the browser's behavior, specifically the Shadow Dom standard. This led to changes in the framework's code to allow for functioning in today's environments, although with somewhat limited capabilities.

An example case is the requirement to use the `[fx-item]` attribute on all descendants of the `<flexus-drawer>` element due to the missing encapsulation the Shadow DOM provides. The drawer's descendants could not be directly addressed with `flexus-drawer > *` CSS selector. The `<flexus-toolbar>` on the other hand is feature complete as elegance of the internal element's code was sacrificed in order to work properly in both native and polyfilled environments. This will, thankfully, change in the future as Microsoft has announced that it will deliver Web Components<sup>3</sup> and they already have added another related standard, the CSS Custom Properties [10] that Flexus is heavily utilizing, in a recent April 2017 update of Windows.

#### 4.4.1 Shadow DOM

HTML documents are essentially a tree of elements stacked into a hierarchical structure. One element can have multiple children and it is itself placed in a parent element. The Shadow DOM standard enables to puncture a hole into a given element and create a secondary sub-branch aside from the primary DOM tree. This allows for the creation of a scaffolding within the shadow root that is hidden from the user but defines the outlook of the element. This underlying shadow structure can contain slots `<slot>`, serving as entry points into which the actual element's children will be redistributed into.

The shadow root is invisible and inaccessible to the outer world, ensuring encapsulation of not only HTML nodes but also styles. CSS Rules applied to inner shadow elements do not leak outside and remain immune to the application's styles with an exception of CSS Custom Properties.

Flexus utilizes this in an attempt to simplify the application's code by applying default styles to the immediate first-level children of elements such as `<flexus-toolbar>` where `<button icon="...">` looks appropriate for the context of the Toolbar, whereas the same code outside it will look like a normal content button without any additional style modifiers or added classes.

The following example shows `<flexus-toolbar>` components that self-configure themselves based on the children they contain, attaches missing attributes and redistributes the elements into an underlying shadow root. In this case, a main section complemented by a `[search]` section, an image which is recognized as collapsible content, followed by tabs in an overlay mode, resting on top of the `<img>` element.

---

<sup>2</sup><https://www.webcomponents.org/polyfills>

<sup>3</sup><https://blogs.windows.com/msedgedev/2015/07/15/microsoft-edge-and-web-component>

```

<flexus-toolbar>
  <section>...</section>
  <section search>...</section>
  <img src="">
  <flexus-tabs overlay>...</flexus-tabs>
</flexus-toolbar>

```

Listing 4.1: Original toolbar code

```

<flexus-toolbar>
  <section slot="before">...</section>
  <section slot="before" search>...</section>
  <img src="" collapse slot="collapsible">
  <flexus-tabs overlay sticky slot="after-overlay">...</flexus-tabs>
</flexus-toolbar>

```

Listing 4.2: Actual DOM code after self configuration

```

...
<div id="before">
  <slot name="before"></slot>
  <div class="overlay">
    <slot name="before-overlay"></slot>
  </div>
</div>
<div id="collapsible">
  <div id="parallaxwrap">
    <slot name="collapsible"></slot>
  </div>
</div>
<div id="after">
  <div class="overlay">
    <slot name="after-overlay"></slot>
  </div>
  <slot name="after"></slot>
</div>
...

```

Listing 4.3: Excerpt from shadow root

#### 4.4.2 CSS Custom Properties

CSS Custom properties are unique because unlike other CSS rules they do not change the immediate style of the element. A variable is created in the scope of the element and all descendants are able to retrieve its value using `var(-my-variable)` syntax. This allows for infinite theming capabilities as featured in the following approximation of how Flexus theming mechanism works. Most other frameworks currently require developers to pre-build themes into static CSS files, whereas Flexus, with a built in color palette, lets developers specify colors on any level, using attributes like `[primary="..."]`. Not only would this be unsustainable with the amount of styleable elements and CSS's default outside-in styling mechanism, but Flexus' unique feature "adaptive tint" could not be done at all.

```

[primary="red"] button {background-color: red}
[primary="red"] input {border: 1px solid red}
[primary="red"] h1[tinted] {color: red}
[primary="red"] h2[tinted] {color: red}

[primary="blue"] button {background-color: blue}
[primary="blue"] input {border: 1px solid blue}
[primary="blue"] h1[tinted] {color: blue}
[primary="blue"] h2[tinted] {color: blue}

[primary="green"] button {background-color: green}
[primary="green"] input {border: 1px solid green}
[primary="green"] h1[tinted] {color: green}
[primary="green"] h2[tinted] {color: green}

```

Listing 4.4: Conventional theming

```

[primary="red"] {--primary: red}
[primary="blue"] {--primary: blue}
[primary="green"] {--primary: green}

button {background-color: var(--primary)}
input {border: 1px solid var(--primary)}
h1[tinted] {color: var(--primary)}
h2[tinted] {color: var(--primary)}

```

Listing 4.5: CSS Custom properties way

## 4.5 Modularity

The most visible feature of Flexus is, of course, the UI styling provided by the CSS files. Those provide all of the necessary style rules for both built-in elements and custom components. It could realistically be used stand-alone without the Core Support Library, even though it is not recommended as the Core Support Library handles platform detection and adjusts to changes. However, what is recommended, is omitting the inclusion of custom component javascript files.

Folder `elements` contains many `.js` files with implementation of element's advanced behavior, starting by registering a custom element, causing the browser to actively enhance every instance of the element with a shadow root and given code. But this process is an unnecessary overhead for simple applications that might not need all of the elements. If, for example, only a simple `<flexus-toolbar>` is used, without any sections, nor the advanced collapsible capabilities, it is not needed to load a `toolbar.js` since all basic styles are already provided by a CSS stylesheet.

Components inherit the same resources from the Core Support Library, but they are built to be independent of one another to further improve load speeds and performance. The components were built to collaborate when used together, like `<flexus-tabs>` and `<flexus-page>`, but they are not mutually dependent.

## 4.6 Performance

The most common complaint against using web technologies as a reliable application development environment is performance. The problem, however, is not in the commonly

blamed JavaScript language, but the Document Object Model, a live, interactive representation of HTML code, more specifically DOM manipulations are very slow compared to JS loops and function calls. The initial render is not that big of a problem but subsequent modifications to the DOM can cause slowdowns if handled without care. I took a great deal of care to avoid this as much as possible.

Browser vendors have been progressively optimizing both JavaScript engines and DOM renderers over the past couple of years while the web platform itself has expanded in terms of, not only, new features but also new performance-related specifications. For example CSS 3 introduced GPU accelerated animations and transitions, offloading computational tasks down the stream to a GPU, when used properly. Contrary to old bad practises using obsolete libraries where every step of the animation is calculated in JavaScript and then applied to the DOM, resulting in an overhead of ineffective calls that could be handled by the Browser or better yet the GPU. This cumbersome method persists to this day on many websites and the problem lives on.

I, however, took a great deal of care during the designing of Flexus to cautiously use resources, avoid unnecessary operations, only follow the best practices and utilize new performance specifications where possible. The most important of those are: GPU Acceleration, utilizing a rendering scheduler, executing tasks in batch, and offline DOM manipulation.

#### 4.6.1 DOM Manipulation

Manipulation with Document Object Model, i.e. changing text, updating classes, attributes, styles or inserting new nodes is slow and should be decreased to minimum since it triggers many other rendering related operations. Flexus adheres to good practices of applying changes in batch instead of one after another. Additionally, “offline” manipulation outside the active DOM is preferable. `DocumentFragment` serves as a temporary host for all new or modified nodes that are then inserted into a live DOM all at once. One of the cases of offline manipulation is the creation of a shadow root of custom components, when all of the template nodes are created at first within a `DocumentFragment` which is then appended into the `shadowRoot` DOM tree.

#### 4.6.2 GPU Acceleration

Ensuring smooth transitions and animations is a difficult task because of DOM manipulations, most of the style changes<sup>4</sup> lead to a repaint and even reflow in the worst cases. These are the hidden processes of a browser’s rendering engine. Reflow is triggered when the layout of the document has changed, e.g. the CSS property margin is modified, changing the element position and forcing the geometry of all nodes to be recalculated. This is followed by a repaint phase which literally paints the document by applying colors, textures and effects onto precalculated node positions. More specifically, both of these phases are evaluated by the CPU which then offloads the actual repaint rendering onto the GPU. Luckily, usage of the CSS properties `opacity` and `transform` can optimize this process. These properties instruct the GPU to create a snapshot (a transparent layer with only the element bearing it) during the first repaint after a reflow. The first render pass is then executed as usual by stacking all layers together and outputting the result. Subsequent changes to `opacity` or `transform` avoid the repaint phase and are sent directly to GPU where only a single layer from the last snapshot is modified without interference from the CPU. This dramatically

---

<sup>4</sup><https://csstriggers.com>

speeds up the repeated modifications that animations consist of, since such operations are relatively cheap for the GPU to perform, unlike assembling everything from scratch.

### 4.6.3 Caching

Caching is an ambiguous term since it's used to describe many different operations. In my case, I used it to store values from nested properties within objects. It could be dismissed as trivial but profound differences can start to show up if the property-accessing code is called regularly, i.e. inside a loop or event callback, or even worse, when the property is deeper than a single level. The following excerpt is taken from file `elements/toolbar.js` and shows a modification of CSS transform property on the element `parallaxwrap` that is nested within `this.$`.

```
this.on('collapse', ([p, s, capped]) => {
  this.$.parallaxwrap.style.transform = `translate3d(0px, ${capped}px, 0)`
})
```

Since the path is expected to remain unchanged, it is wise to store the latest accessible object from the path to a separate variable `parallaxwrapStyle`.

```
var parallaxwrapStyle = this.$.parallaxwrap.style
this.on('collapse', ([p, s, capped]) => {
  parallaxwrapStyle.transform = `translate3d(0px, ${capped}px, 0)`
})
```

It might not be always useful as code readability is also a concern for future maintainability but the provided example is justifiable due to it being a callback to the event `collapse` which fires in bursts of hundreds as it is tied to scrolling.

It becomes especially beneficial in combination with a deeper knowledge of how rendering/layout engines work. Every DOM Element offers a set of methods and properties such as `offsetWidth`, `offsetHeight` and a little known fact is that merely reading these offset properties forces costly reflows and repaints. Caching them is therefore reasonable, specifically in case of Flexus's Toolbar element which, in some configurations, needs to perform adjustments based on size.

### 4.6.4 Scrolling with Passive Listeners

A common problem, seen with web development, is scroll performance. To remedy this, Flexus uses new standard Passive Event Listeners [19]. To see the benefit we first have to understand the basics of event handling. Every time the browser fires an event, the developer is offered a window for reaction. This could be plain update of a variable or an impacting change to DOM which modifies the behavior of the fired event, in which case a default action taken by the browser has to be canceled by calling the method `event.preventDefault()`. The browser has to presume this prevention every time it fires an event and allocating needed resources for this operation results in delayed, unpredictable reactions that become noticeable during a series of quickly fired events such as those produced by scrolling. Therefore effects that respond to scrolling often appear to be unpleasantly lagging behind. Now, registering a listener as passive marks a promise from developer to the browser not to prevent the default action but to only observe. No additional resources are wasted and results are significant. Launching the same example of collapsible toolbar in Chrome shows a smooth transformation during scrolling whereas in Edge the toolbar jumps around and

lags behind. That is because Edge has not yet implemented <sup>5</sup> passive listeners, however, this is expected to be included in future versions as this feature is already in development <sup>6</sup>.

#### 4.6.5 Scheduling the Browser’s Animation Frame

For an application to be conceived as smooth, be it through animations or scrolling, a threshold of sixty frames displayed per second should be met. Previously mentioned optimizations help reduce the overhead, allowing the browser to focus on reliably delivering those sixty frames. This means that every 16.6 milliseconds one frame has to be rendered. If some other operation takes up browser’s resources at this point, the application starts to appear laggy. The worst offenders of this are scroll and pointer events (including touch and mouse movement). Those are being fired erratically and usually in rapid bursts, possibly even multiple times within the 16ms frame between rendering. Executing callbacks on every single one of them results in disastrous performance hit if it interacts with DOM. Fortunately the `requestAnimationFrame` [16] spec gives developers deeper integration into the browser’s timer by allowing them to schedule code to be executed exactly at the moment of the frame rendering when the browser is “warmed up” for these kind of tasks. Flexus utilizes this scheduler in elements inheriting the `Scrollable` class and vastly improves scroll performance by utilizing this scheduler, therefore reducing the number of callbacks by only executing the latest ones.

---

<sup>5</sup><http://caniuse.com/#feat=passive-event-listener>

<sup>6</sup><https://developer.microsoft.com/en-us/microsoft-edge/platform/status/passiveeventlisteners/?q=passive>



## Chapter 5

# Evaluation for Real World Usage

Throughout the development of Flexus, I have been regularly testing snippets of code and also complete demo applications to ensure that the project is not only useful but also performant and on par with applications that are written in native languages such as C# and Java.

As Webview adds another step between the application code and hardware, therefore, it is only to be expected that it will be slower than native languages. This is, however, a problem of the web platform which is not designed for creating intensive 3D graphics and games to begin with and instead is ideal for applications such as to-do lists, calendars, news readers etc. In this case „slower“ is defined as tens of milliseconds. As discussed in the previous chapter 4.6, the big focus during Flexus' development was to make it performant in both the initial load time along with actual interaction within the application.

Flexus was regularly tested on a Nexus 5 device, which as of writing this thesis, is an almost four year old mid-range Android phone. The subjective load times from the users' point of view are almost indistinguishable from native applications, with no noticeable slowdown when delivering content. Furthermore, I was able to test on a Nexus 7 device (released in 2012) which is, by today's standards, a considerably slower device. Through multiple tests it was obvious that even the native Java applications were slow on this device. With this in mind, it would be fair to assume that HTML applications would be noticeably worse however, they are absolutely comparable when considering performance.

Additionally, on Android devices I tested their ability to install the applications from the web, due to the Android OS already supporting the PWA manifest. The launch speed of PWA apps is heavily dependent on the way that the application's files are served. I was unable to properly test caching the application's files locally on the device as that would require the availability of an HTTPS server. I was able, however, to test it in remote mode where at the moment of launching, all of the files would be loaded from the server. This process is also dependant on the application's content and images however, a basic app (consisting of a single view with text and the core components of Flexus) loaded within a second or two. This test was performed on a regular home WiFi connection which causes the largest performance impact by retrieving the application's files from the internet.

Another factor that increases the performance of Flexus lays in its modularity. For a simple application, only the design CSS files and preferably the Core Support Library need to be loaded, unless any advanced features like tabs/pages or toolbar are required. Unlike Google's Paper Elements library (also a web HTML based UI library), Flexus tries to keep the number of newly created custom elements as low as possible. For example, creating a list would require list item elements. Flexus merely provides the option of using the `[fx-item]`

attribute that can be applied to anything from `<div>` to `<label>`, whereas Polymer creates a whole new custom element `<paper-item>`. A custom element like this is defined in another file which then has to be imported, and the component's code registered into the browser's element registry. Then, each time an item is created, the component's underlying JS code is executed, adding unnecessary performance overhead. Following the same topic of simplification, with Flexus, icons can be added to almost anything, by applying the attribute `[icon="myicon"]`, whereas Paper Elements requires the use of `<iron-icons>` components, making it less practical (a deeper node tree) and more expensive (in both load time and execution).

Regarding its usefulness, Flexus excels at handling the visual appearance of an application by adjusting it to screen size, touch or mouse input, and provides proper styling with Material Design and/or Neon Design. Developers can focus on writing their ideas in declarative HTML and Flexus will take care of the styling and layout. It can be used to develop applications that are deployable into the Windows Store, Android Store, and Chrome OS Web Store, and along with the expansion of the PWA standards, the web.

The simplicity of the resulting application's code was discussed in chapter 3.2, but it should be stated that the code is not just clean, but also functional. Component `<flexus-toolbar>` offers powerful animations that are reactive to scrolling of view's content, which could be done with just a few lines of code. Just adding a `<img>` element as a child to a `<flexus-toolbar>` triggers a number of internal operations e.g. listening for changes in scroll position, updating state, rendering and accelerating animations through the GPU to ensure smooth scrolling. This was achieved by utilizing Web Components standards which allowed for encapsulating such complexity internally within the component (during its lifecycle) and away from the application developers.

The result of this is a highly declarative code that, in the case of simple applications, can stand on its own without the need of any additional JavaScript since, for example, the components `<flexus-tabs>` and `<pages-pages>` can find each other and link themselves. Unless the developer wants to precisely determine the relationship which could be done with matching `[id="..."]` and `[for="..."]` attributes, in the same way that HTML's `<label>` and `<input>` functions. On the other hand, the component's encapsulation ensures Flexus is compatible with any of the popular MVCs, routing or a templating library or framework.

Flexus is, however, not without its flaws. Use of the modern (and often experimental) web standards that Flexus is built on leads to the temporary unavailability of some features in some browsers and environments. Even though Flexus also contains polyfills (open source snippets of code that tries to patch or simulate missing browser features), these standards are very tricky or nearly impossible to polyfill, especially the Shadow DOM (and styling within). I have put considerable effort into making it work and it does collaborate with the polyfills, even with all of the complex scrollable toolbars, at least for the most part. However, this is at the expense of adding complexity to Flexus' source code, making it not as sleek and elegant as it could have been.

At the time of writing, the current version of Chrome (58), Chrome OS, and Android Webview, fully support all of the mentioned standards and therefore all of Flexus and the demo applications work flawlessly. Microsoft's Edge and subsequently Windows 10 UWP platform (built on top of Edge's rendering code) are currently missing the implementation of the Web Components standards. Due to this, some bugs are expected to surface and it is possible that more complex applications may not work at all. This, however, is only a temporary problem as Microsoft have pledged to support these standards in future releases as have the other major platforms.

Another limitation is caused by sandboxed environments. Flexus is able to automatically detect the platform and/or OS and tries to load up the appropriate design language automatically (if it's not defined by the developer) including polyfills. There is a limitation that is only present with UWP and Chrome OS applications as they are executed in a restrictive manner, where the CSP (Content Security Policy) prevents the application from injecting another scripts or styles. Using less restrictive shells like Electron, NW.JS or Cordova resolves and removes this limitation.

Flexus does not, currently, offer all of the components for developing intricate applications. I intend to remedy this in the future as I will be continuing the development and expansion of Flexus as an open source project, improving current functionality, introducing new components, and adding support for Apple devices through the iOS design language.

## Chapter 6

# Conclusion

In this thesis I have researched and studied the available literature relating to the design and development of mobile applications and programs. There are currently a variety of platforms and my findings proved that developing for them is difficult due to differences in programming languages executable on each platform. Another obstacle is the distinctive visual appearance that these platforms enforce. This problem is further compounded by characteristic visual and behavioral patterns of the two input types, the touch and the mouse pointer, around one of which are applications usually optimized. Despite available alternative to use web technologies for development, it is not that practical because very few multiplatform libraries and frameworks are available. And those that are do not tackle all of the aforementioned problems all at once, rendering the libraries impractical as it still leaves more work to be done by developers.

Based on these findings I have devised a simplistic API of a multiplatform framework for application development and building User Interfaces. I have implemented this framework in HTML, CSS and JavaScript languages using modern web standards. The name chosen is Flexus as the framework flexibly adjusts interface of the application to fit the device it is launched on. Flexus primarily provides two things: the visual style for the application and a set of components to build it with. Two design languages are supported – Material Design for Android and Neon Design for Windows 10. Both are complemented by large set of icons and a color palette that can be used for styling. Also provided is a set of modular custom components. Among these components are the basic building blocks such as navigational drawer, toolbar, tabs, and more. By default, their basic styling is included, but developers can selectively import respective additional javascript files to further enhance behavior of such elements when needed.

The framework was designed to be simple on the outside, yet complex and powerful on the inside. Giving developers the power they need to customize their applications if they need it, or automatically configure itself and all components with smart assumptions based on surroundings. This goal has been achieved. After a quick and straightforward setup of simply including a few JS files, the Flexus framework takes over and adjusts the single codebase for various platforms, screen sizes, and form factors. The appropriate design language is loaded, layout and structure changes with screen size, spacing and sizing is increased and touch gestures are enabled on touch screens, advanced effects can be used with minimal code and can be automatically disabled on slower devices.

This project started as an exploration of what the ideal future UI framework could be, and the result is an actual working tool. However, there's a caveat to that. Flexus is built on top of modern web standards, which poses some problems on platforms that have not

yet implemented all of these standards. However, all platforms have previously announced development of Web Components standards so this inconvenience is only temporary. Currently targeted platforms are Windows 10, Android, Chrome OS and Web. Flexus has been released as an open-source project on Github and I intend to continue development, add support to iOS and expand functionality with more components.

# Bibliography

- [1] Google: *Material Design Guidelines: Color Palette*.  
Retrieved from: <https://material.io/guidelines/style/color.html#color-color-tool>
- [2] Google: *Material Design Guidelines: Layout - Metrics & keyline*.  
Retrieved from: <https://material.io/guidelines/layout/metrics-keylines.html>
- [3] Google: *Material Design Guidelines: Layout - Units & measurements*.  
Retrieved from: <https://material.io/guidelines/layout/units-measurements.html#units-measurements-density-independent-pixels-dp>
- [4] Google: *Material Design Guidelines: Structure*.  
Retrieved from: <https://material.io/guidelines/layout/structure.html#structure-app-bar>
- [5] Google: *The Web App Manifest*.  
Retrieved from: <https://developers.google.com/web/fundamentals/engage-and-retain/web-app-manifest/>
- [6] Microsoft: *Introduction to UWP app design*.  
Retrieved from: <https://docs.microsoft.com/en-us/windows/uwp/layout/design-and-ui-intro>
- [7] Mozilla: *Progressive web apps*.  
Retrieved from: <https://developer.mozilla.org/en-US/Apps/Progressive>
- [8] Mozilla: *Web App Manifest*.  
Retrieved from: <https://developer.mozilla.org/en-US/docs/Web/Manifest>
- [9] Onsen: *Onsen UI: CSS Components*.  
Retrieved from: <http://components.onsen.io>
- [10] W3C: *CSS Custom Properties for Cascading Variables Module Level 1*.  
Retrieved from: <https://www.w3.org/TR/css-variables/>
- [11] W3C: *CSS Flexible Box Layout Module Level 1*.  
Retrieved from: <https://www.w3.org/TR/css-flexbox-1/>
- [12] W3C: *CSS Pseudo-Elements Module* .  
Retrieved from: <https://drafts.csswg.org/css-pseudo-4/#generated-content>

- [13] W3C: *Custom Elements*.  
Retrieved from: <https://www.w3.org/TR/custom-elements/>
- [14] W3C: *HTML Imports*.  
Retrieved from: <http://w3c.github.io/webcomponents/spec/imports/>
- [15] W3C: *Shadow DOM*.  
Retrieved from: <https://www.w3.org/TR/shadow-dom/>
- [16] W3C: *Timing control for script-based animations*.  
Retrieved from: <https://www.w3.org/TR/animation-timing/#dom-windowanimationtiming-requestanimationframe>
- [17] webcomponents.org: *Web Components Specifications*.  
Retrieved from: <https://www.webcomponents.org/specs>
- [18] WHATWG: *HTML Living Standard*.  
Retrieved from: <https://html.spec.whatwg.org/multipage/scripting.html#the-template-element>
- [19] WICG: *Passive event listeners*.  
Retrieved from: <https://github.com/WICG/EventListenerOptions/blob/gh-pages/explainer.md>

# Appendix A

## Content of the DVD

The attached DVD contains the following notable directories and files:

```
|-- flexus (directory containing compiled and source code of flexus and demos)
|   |-- css (directory with compiled Flexus CSS design languages)
|   |-- demo (directory with variety of demo applications)
|   |-- elements (directory with compiled Flexus custom components)
|   |-- fonts (directory with fonts and icons used by CSS files)
|   |-- lib (directory with compiled Flexus Core Support Library and Ganymede)
|   |-- polyfills (polyfills fixing missing functionality in old browsers)
|   |-- src (directory with original not compiled source codes of Flexus)
|   |-- test (directory with code snippets and testing apps used during development)
|   |-- flexus.jsproj (Visual Studio UWP project)
|   |-- gulpfile.js (build tasks used for compiling)
|
|-- thesis (directory containing this thesis and its source)
    |-- src (directory containing source code of this thesis)
    |-- thesis.pdf (compiled electronic version of this thesis)
    |-- manual.pdf (user's guide for working with Flexus framework)
    |-- promo.mp4 (brief video about Flexus framework and this thesis)
    |-- poster.png (promotional poster about Flexus framework and this thesis)
```

The directory `flexus` also contains various other files related to or required by Visual Studio project, build tasks, git, etc. The directories `flexus/fonts` and `flexus/polyfills` contain open-source files created by third parties, retrieved from <sup>1</sup> <sup>2</sup> <sup>3</sup>, which are used by Flexus or the demo applications.

---

<sup>1</sup><https://www.webcomponents.org/polyfills>

<sup>2</sup><https://materialdesignicons.com/>

<sup>3</sup><https://docs.microsoft.com/en-us/windows/uwp/design-downloads/index>



# Appendix B

## Demo applications

Following are a few examples of applications, from the flexus/demo directory on attached DVD, that were built using Flexus.

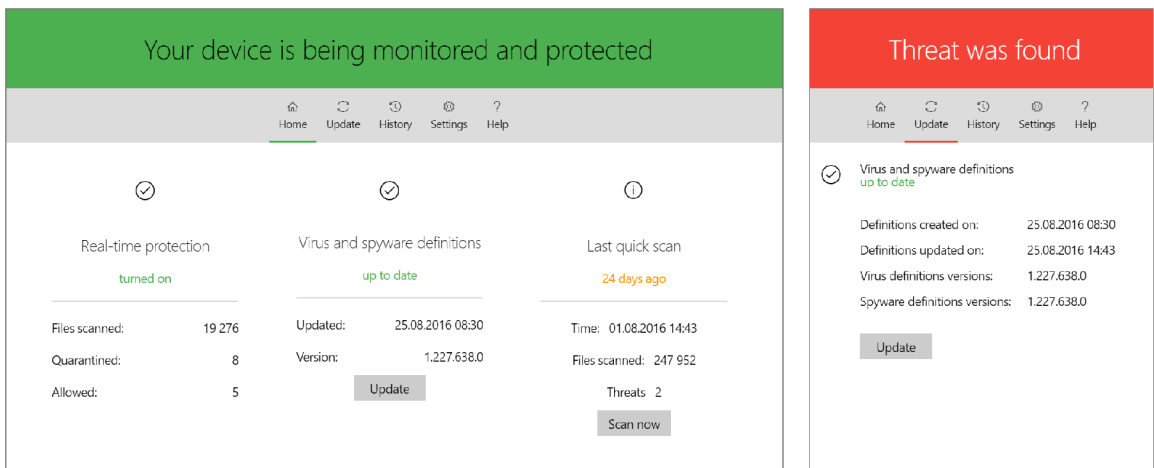


Figure B.1: Antivirus demo application.

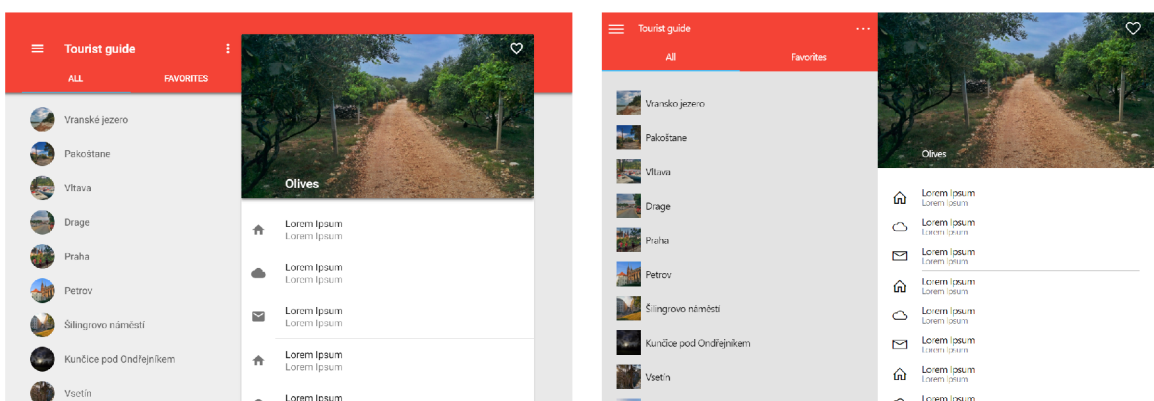


Figure B.2: Tourist guide.

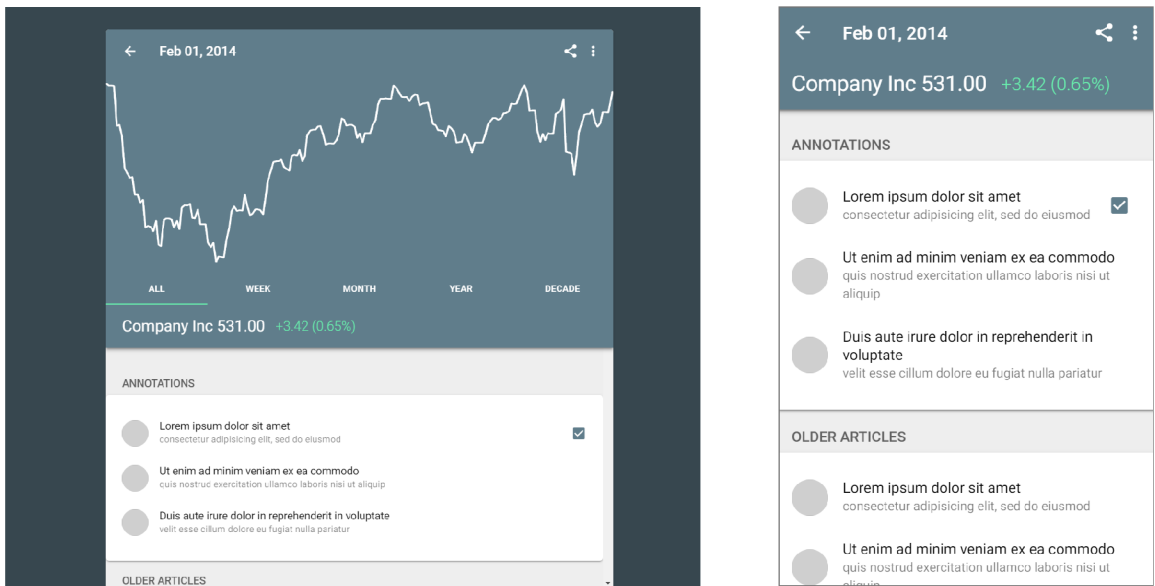


Figure B.3: Stocks application.

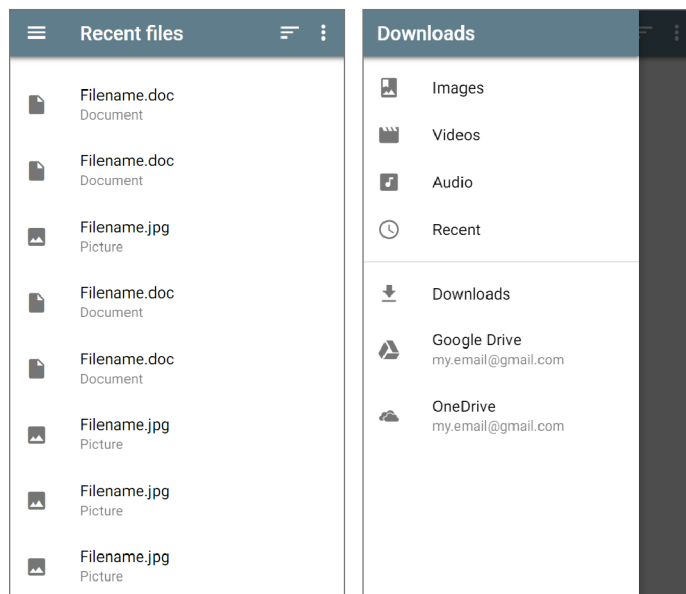


Figure B.4: Simple list of downloaded files inspired by Android's Download application.

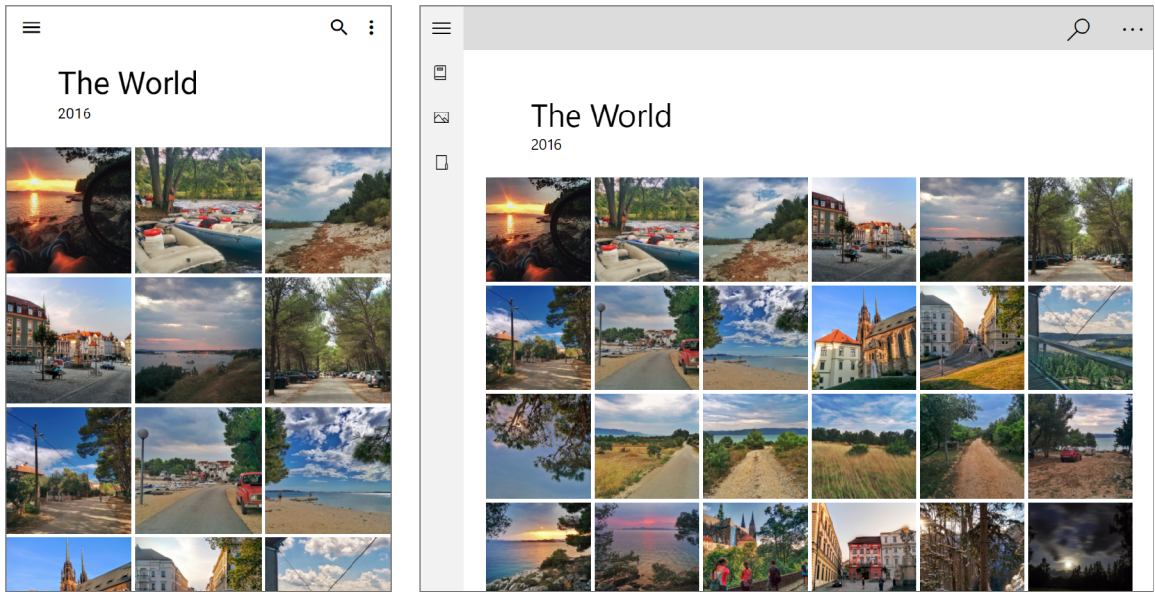


Figure B.5: Photo gallery.

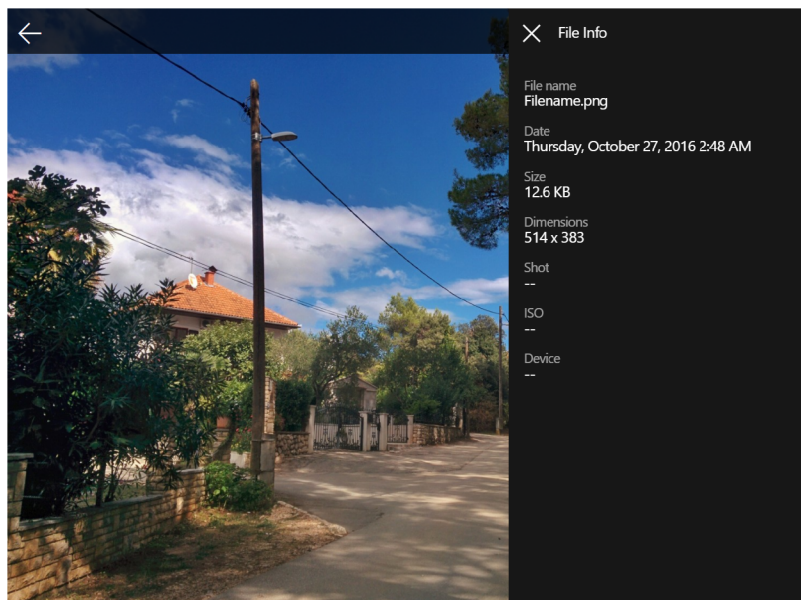


Figure B.6: Panel with details of a photo.

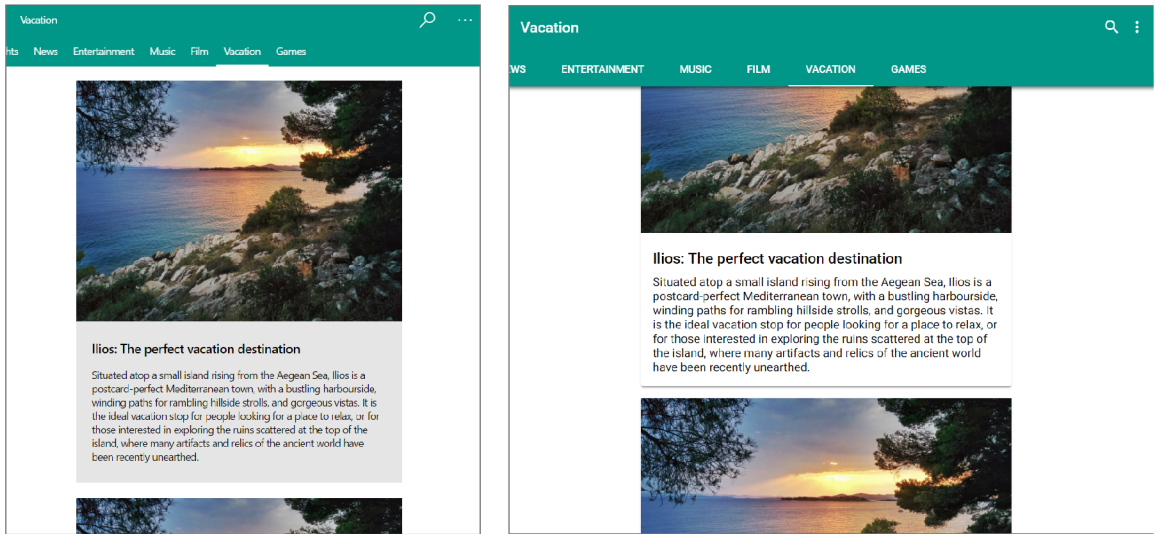


Figure B.7: List of articles represented by cards.

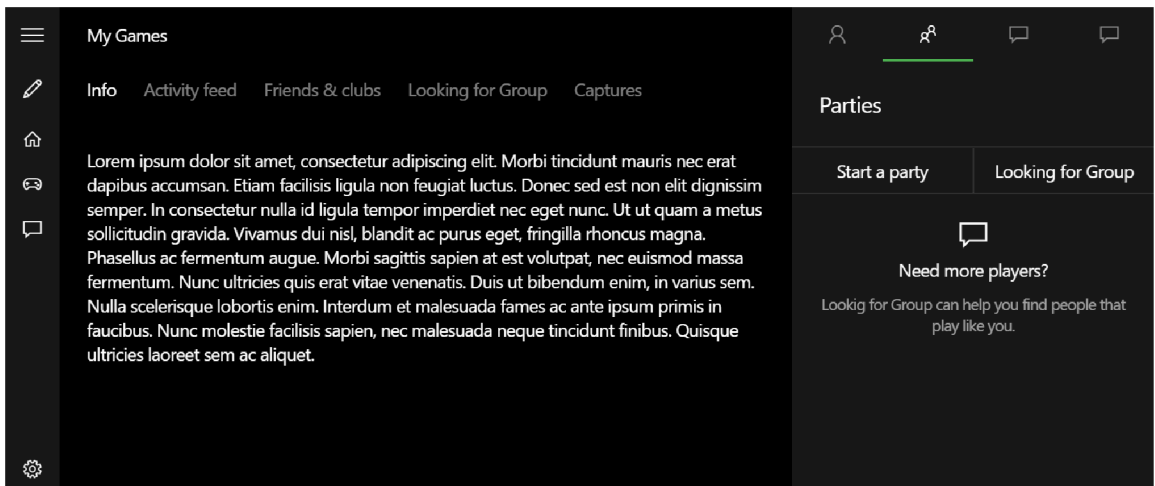


Figure B.8: Video game hub.

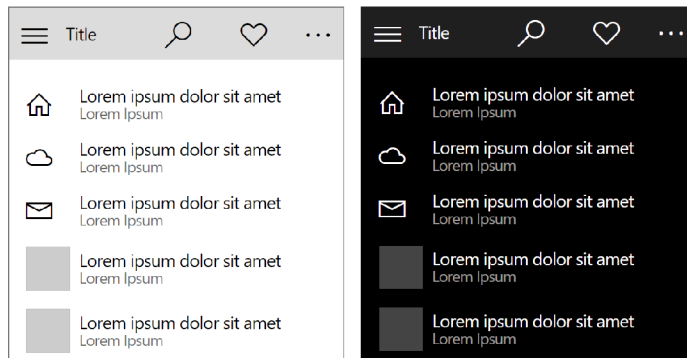


Figure B.9: Simple list application.