BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

THESIS TITLE
NÁZEV PRÁCE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **JAKUB ZÁRYBNICKÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. ONDŘEJ LENGÁL, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2019

Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

Keywords

Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.

Klíčová slova

Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.

Reference

ZÁRYBNICKÝ, Jakub. *Thesis title*. Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Lengál, Ph.D.

Rozšířený abstrakt

Do tohoto odstavce bude zapsán rozšířený abstrakt práce v českém jazyce, bude mít rozsah 2 až 6 normostran a bude obsahovat úvod, popis vlastního řešení a shrnutí a zhodnocení dosažených výsledků.

Thesis title

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Ondřej Lengál, Ph.D. All the relevant information sources used during preparation of this thesis are properly cited and included in the list of references.

.....
Jakub Zárybnický
May 13, 2019

Contents

List of Listings

Chapter 1

Introduction

Imagine you are building an application for an event your company is organizing. There will not be reliable internet connection in the area of the event, so it needs to work offline. You are on a tight budget, so implementing one version for each platform is not feasible, but it needs to work reliably across all mobile platforms and ideally in the browser as well. What is the easiest way to accomplish that?

This situation is exactly where a rather new concept of a Progressive Web Application (PWA) is the best solution. Generally said, a PWA is website capable of running offline, using mobile notifications, or synchronizing data in the background, things previously specific to native mobile applications.

Moreover, consider that the application server is written in Haskell, a statically typed, purely functional programming language. We want to reuse the business logic already written there to avoid duplicating code, so we search for a way to write a Web application in Haskell. We find many resources and a quickly growing community but while creating the application, we soon step into the unknown. A medium-scale application needs a large number of capabilities, but the ecosystem of frontend Haskell is not yet big enough to support many of them, and many libraries in the area are either exploratory or one-off projects. In this work, we will try to fill in many such gaps, with the goal of creating a framework for creating Progressive Web Applications in Haskell.

We will first go through the details of what a Progressive Web Application is (Chapter 2) and what features are common in today's Web frameworks (Chapter 3), followed by a quick introduction to Haskell and an evaluation of the Haskell ecosystem in the area of Web development (Chapter 4). After seeing what is missing, we will walk through the implementation of several libraries in the area (Chapter 5), have a look at how to create an application using these components (Chapter 6), and conclude with four separate case studies (Chapter 7).

1.1 Related work

TODO: Move to chapter 3?

The volume of work in the area of frontend Haskell is not large, as the Haskell-to-JavaScript compiler GHCJS is only available since 2013, and also due to the fact that Haskell in general is only recently becoming a mainstream language and used in commercial projects. Academic work in this area is sparse, but there are several mature projects under active development, usually commercially sponsored. Reflex and Obelisk are two projects from Obsidian Systems [32], a UI framework and a deployment tool respectively. Tweag [44] is working on a Haskell-to-WebAssembly compiler, Asterius, and QFPL [22] has created many learning materials for frontend Haskell.

Chapter 2

What is a PWA?

2.1 Web Development Trends

TODO: Maybe too short paragraphs, connect them?

Today, web pages are more than just static HTML markup. We have moved from the Web without JavaScript where any interactions needed to be processed by the server, to using JavaScript for animations and small ease-of-use features, to building entire applications in JavaScript that only talk to servers in the background.

When JavaScript received the ability to execute HTTP requests and get or save data in the background, a technology then called AJAX, it was only used for small pieces of functionality. Only later came the concept of Single Page Applications (SPA), where the application was only loaded once at the beginning and only communicated in JSON or XML afterwards. This brought new challenges to JavaScript developers, challenges like rendering templates in the browser, managing application state or routing (the mapping between application state and the URL displayed in the address bar of a browser).

Not only was the Web developing, but also mobile devices and their applications, also called native applications today (as opposed to web applications). These applications had access to the full capabilities of the devices they were running on and could, at the start, do a lot more than a web application.

This meant that a company that wanted to reach as many users as possible had to develop many versions of the same application: for the web, desktop, and also for several varieties of mobile devices. Later came projects like Cordova and Electron that made it possible to use JavaScript and other web technologies not only in the browser but also in native mobile and desktop applications, but the ambition of making the web the universal application platform stayed there.

Under the umbrella of the Web Platform project, many new JavaScript extensions were developed, interfaces that gave JavaScript access to many facets of the device on which the browser runs. Some examples are the Location API that gives access to the GPS location of a device, or the Screen Capture API that enables a website to capture the screen of the browser or another application.

Given that the device type most often for browsing the Web is a mobile device and has been for a few years already [38], the effort to equalize the capabilities of web and mobile applications is still ongoing, and a large step towards this goal is the concept of a Progressive Web Application.

2.2 Progressive Web Applications

The term Progressive Web Application (PWA) is an umbrella term for several relatively new, closely related technologies. While many of them are useful on all devices, the main target audience are mobile Web browsers.

The goal of the PWA project is to enable developers to create Web applications that are equal in functionality to native ones in many domains like shopping or productivity. The technologies that comprise the term PWA allow applications to be installable directly from a web browser to the device, to become capable of working offline, to use some background services of the device, all features formerly only available to mobile applications. To use the marketing terms, converting a web application into a PWA improves user experience and brings higher user engagement and retention.

The term PWA has an exact specification in a checklist created by Google [16], which describes two levels of PWAs, a Baseline PWA and an Exemplary PWA. The defining characteristics of a Baseline PWA are, as quoted directly from the checklist:

- Pages are responsive on tablets and mobile devices.
- All application URLs load while offline.
- Site uses cache-first networking.
- Page transitions do not feel like they block on the network.
- Pages use the History API.
- Each page has a URL.
- Metadata provided for “Add to Home screen”.
- Site appropriately informs the user when they are offline.
- Push notifications (which consist of several related requirements).

While there are several more requirements for an Exemplary PWA, we will focus mostly on the Baseline PWA ones. The technologies used to fulfill these requirements are relatively recent developments, but they are supported in all major Web browsers. The technologies are the following:

- Service Workers
- Web App Manifest
- IndexedDB

- Web Platform APIs

A service worker is a JavaScript program that an application can request to install. It is functionally a configurable network proxy [29] that can intercept outgoing requests from the browser and that has access to a browser cache, which, among other things, enables applications to become available offline. The service worker may also handle push notifications and background synchronization, two new features that were traditionally available only to native applications.

Push notifications are short messages sent by the application server to any client using browser-specific channels (e.g. Firebase Cloud Messaging for Chrome and Android browsers, Apple Push Notification for Apple browsers), that are shown to the user as a popup or a notification regardless of whether the application is open or closed on the device.

The Background Sync API enables the service worker to retry requests made while the application was offline as soon as the device goes online, even when the application is not open at that moment, which also enables some degree of offline capabilities, as any data updates can be queued and eventually executed in batch at some point in the future.

The Web App Manifest is a W3C-standardized JSON file [47] that contains the metadata that describe an application: its name, icons, splash screen, or locale. If a page contains a link to a manifest, it indicates to the browser that the page is a part of an application and that the application can be installed on a device locally. For the user, this means that the application can request to be installed via a dialog window asking them to "Add to Home Screen".

IndexedDB is the only browser storage that is accessible to both the browser and the service worker. It is a document store that supports transactions, schema versioning, and indices. Using IndexedDB, the application is able to synchronize its state with the server even when it is closed, using the Background Sync API of the service worker.

The Web Platform is a set of APIs that expose capabilities of the underlying system. Examples include geolocation or audio/video capture [4]. Of the many APIs that comprise the Web Platform, it is the History API and Network Information API that are necessary for a PWA. The History API is the feature that enables the so-called *single page applications*, where the application is loaded only once despite the user being able to navigate between different URLs. This is achieved via artificial *navigation actions* and intercepting user navigation actions like "Go to previous page". The Network Information API is what enables the application to find out whether the it can currently access the Internet. Other APIs mentioned in the *Exemplary PWA* requirements are the Web Share API and Credentials API that expose more of the underlying device capabilities, sharing via other applications and the device credential storage.

Chapter 3

Web frameworks of today

TODO: Rename chapter, "Existing Web Frameworks"?

TODO: Chapter intro

3.1 Features of Web Frameworks

The basis of a web framework is the *UI toolkit*, which defines the structure, architecture, and paradigm of the rest of the application. I am intentionally using the now-uncommon term toolkit, as the UI frameworks we will see vary in their scope, e.g. React is just a library with a small API, whereas Angular provides a quite opinionated platform. Individual frameworks are quite disparate, with large differences in the size of their community, maturity, developer friendliness, and the breadth of features or available libraries.

Frameworks usually have one defining feature they are built around (virtual DOM for React or event streams for Angular), but there are many other concerns that a framework needs to take care of. *Templating* is one of the essential ones. It is a way of composing the HTML that makes up an application, which also usually includes some "view logic" and variable interpolation. In some frameworks the whole program is a template (purely functional React), some have templates in separate files and pre-compile them during the build process or even in the browser (Angular). Templates may also contain CSS as well as in the recent CSS-in-JS trend.

The second defining feature of frameworks is *state management*. This rather vague concept may include receiving input from the user, displaying the state back to the user, communicating with APIs and caching their responses, etc. While state management is simple at a small scale, there are many problems that appear only in larger applications with several developers. Some approaches include: a "single source of the truth" and immutable data (Redux), local state in hierarchical components (Angular), or unidirectional data flow with several entity stores (Flux).

Another must-have feature of a framework is *routing*, which means manipulating the displayed URL using the History API, and changing it to reflect the application state and vice-versa. It also includes switching the application to the correct state on start-up. While

the router is usually a rather small component, it is as fundamental to the application in the same way as the previous two items.

A component where frameworks differ a lot is a *forms* system. There are a few layers of abstraction at which a framework can decide to implement forms, starting at raw DOM manipulation, going on to data containers with validation but manual rendering, all the way up to form builders using domain-specific languages. The topic of forms includes rendering a form and its data, accepting data from the user and validating it, and sometimes even submitting it to an API.

There are other features that a framework can provide, like authentication or standardized UI components, but frameworks usually leave these to third party libraries. There is one more topic I would like to mention that is usually too broad to cover in the core of a framework, but important to consider when developing an application. *Accessibility* is an area concerned with removing barriers that would prevent any user from using a website. There are many parts to it, and while the focus is making websites accessible to screen-readers, it also includes supporting other modes of interaction, like keyboard-only interaction. Shortening *load times* on slow connections also makes a website accessible in parts of the world with slower Internet connections, and supporting *internationalization* removes language barriers.

Accessibility is something that requires framework support on several levels. Making a site accessible requires considerations during both design (e.g. high color contrast) and implementation (semantic elements and ARIA attributes), and that is usually left to application code and accessibility checklists, with the exception of some specialized components like keyboard focus managers. There are, however, tools like `aXe-core`, which check how accessible a finished framework is, and these can be integrated into the build process.

Internationalization is somewhat easier to support in a framework, as it includes so many cross-cutting concerns. At the most basic level, it means simple string translations, perhaps with pluralization and word order. Going further, it may also mean supporting RTL scripts, different date/time formats, currency, or time zones.

As for *load times*, there are many techniques frameworks use to speed up the initial load of an application. We can talk about the first load, which can be sped up by compressing assets (CSS, fonts, scripts) and removing redundant ones, or by preparing some HTML that can be displayed to the user while the rest of the application is loading to increase the perceived speed. After the first load, the browser has some of the application's assets cached, so loading will be faster. One of the requirements of a PWA is using the Service Worker for instantaneous loading after the first load.

There are two patterns of preparing the HTML that is shown while the rest of the application is loading, so called *prerendering*. One is called *app shell*, which is a simple static HTML file that contains the basic structure of the application's layout. The other is server-side rendering, and it is a somewhat more advanced technique where the entire contents of the requested URI is rendered on the server including the data of the first page, and the browser part of the application takes over only afterwards, without the need to fetch any more data. There is another variant of server-side rendering called the "JAM stack" pattern ("JavaScript, APIs, Markup" [42]), where after application state changes, the HTML of the entire application, of all application URLs is rendered all at once and saved so that the

server does not need to render the HTML for every request. These techniques are usually part of a framework's *supporting tools*, about which we will talk now.

TODO: A separate sub-section "Supporting tools"?

Developers from different ecosystems have wildly varying expectations on their tools. A Python developer might expect just a text editor and an interpreter, whereas a JVM or .NET developer might not be satisfied with anything less than a full-featured IDE. We will start with the essentials, with *build tools*. Nowadays, even the simplest JavaScript application usually uses a build step that packages all its source code and styles into a single bundle for faster loading. A framework's tool-chain may range from a set of conventions on how to use the compiler that might get formalized in a Makefile, through a CLI tool that takes care of building, testing, and perhaps even deploying the application, to the way of the IDE, where any build variant is just a few clicks away.

Debugging tools are the next area. After building an application, trying it out, and discovering faulty behavior, these tools help to pinpoint and fix the underlying error. There are generic language-specific tools, a stepping debugger is a typical example, and there are also framework-specific tools, like an explorer of the component hierarchy (React) or a time-traveling debugger (Redux). In the web world, all modern browsers provide basic debugging tools inside the "DevTools": a stepping debugger and a profiler. Some frameworks build on that and provide an extension to DevTools that interacts with the application in the current window, some provide debugging tools integrated into the application itself.

When building or maintaining a large application with several developers, it is necessary to ensure good practices in all steps of the development process. There are two general categories in *quality assurance tools*: testing (dynamic analysis) tools and static analysis tools. In the commonly used variants, tests are used either as an aid while writing code (test-driven development), or to prevent regressions in functionality (continuous integration using unit tests and end-to-end tests). Static analysis tools are, in the general practice, used to ensure a consistent code style and prevent some categories of errors ("linters"). Frameworks commonly provide pre-configured sets of tools of both types. If necessary, e.g. in integration testing, where the burden of setup is bigger, they also provide utility libraries to ease the initial setup. Some frameworks also use uncommon types of tests like *marble tests* used in functional reactive programming systems.

Editor integration is also important in some ecosystems. This includes common features of Integrated Development Environments like auto-completion or refactoring tools. Recently the Language Server Protocol (LSP) [26] project played a big role in allowing editors to support a wide variety of languages by implementing just an LSP client and being able to communicate with any language-specific language server. There are some parts of editor support that can be framework-specific, like supporting an embedded domain-specific language or integrating framework-specific debugging tools.

While we were talking about Web frameworks so far, some of them support not only running inside the browser but also being packaged as a *mobile app* for Android or iOS, or as a *native desktop application* for the many desktop operating systems. For mobile support, frameworks often provide wrappers around Apache Cordova, which is a thin wrapper around a regular website exposing some extra capabilities of the device. Some, however, go even further and support fully native mobile interfaces controlled by JavaScript, like React Native. The situation is similar for desktop support, just with Electron used as the

base instead of Cordova. The main benefits of packaging a Web application instead of just running it inside a browser are performance (they are usually faster to load and to use), access to device-specific capabilities (direct access to the file system), or branding.

The last point to mention is *code generation*, of which there are two variants: project skeleton generators, which create all files necessary for a project to compile and run and which are provided in a large majority of frameworks. Then there are component generators, which may include generating a template, a URL route and its corresponding controller, or an entire subchapter of a website. These are less common but some frameworks also provide them.

3.2 Web Frameworks in JavaScript

The features we just went through are features that are widely available in JavaScript and its frameworks. We will now go through some of them to see how they approach the implementation of these features.

The most popular JavaScript frameworks of today are React and Angular [2]. Vue.js is close behind them, a relatively new framework that is quickly gaining popularity.

Angular is an integrated framework that covers many common use cases with many supported features in the base framework. On the other hand, React and Vue are both rather small libraries, and most of the features described in the previous section are implemented only as third-party libraries or tools. While React and Vue are sometimes called frameworks as well, they mostly serve as the central library of an ecosystem built around them.

As for the topics mentioned in the previous chapter like routing, forms, or build tools: most of them are built into Angular, while React and Vue do not include them and thus users need to use third-party libraries instead. This ties into the most common complaint about the JavaScript ecosystem: there are dozens of small libraries that accomplish similar things, many are, however, incomplete or unmaintained, and there is no good way to decide between them. There are several projects that attempt to alleviate this problem by combining a set of libraries into a more cohesive framework closer in scope to Angular.

Chapter 4

Haskell and the Web

TODO: intro

4.1 Haskell

```haskell
type HackageAPI =
  "users" :> Get '[JSON] [User] :<|>
  "user" :> Capture "login" Login :> Get '[JSON] User

getUsers :: Handler [User]
getUser :: Login -> Handler User

server :: Server HackageApi
server = getUsers :<|> getUser

getUsersClient :<|> getUserClient =
  client @HackageApi "http://hackage.haskell.org"
```

Listing 1: An example of a web server in Haskell

Haskell is described as a "statically typed, purely functional programming language with type inference and lazy evaluation" [20]. It is originally a research language, developed as a vehicle for new research in the area of programming languages since 1990 [17]. It has served as such, and in fact it still is the target of active research. Some larger ongoing research projects are Dependent Haskell [14] and Linear Haskell [6].

Only recently has it been used in commercial work, as exemplified by Facebook's Haskell spam filter [24]. While there are many benefits to using a strongly typed functional language (it eliminates entire classes of programming errors [31], anecdotally shown by the common saying that "If it compiles, it works") it is conceptually different from languages commonly taught at universities. An example of Haskell code is included in Listing 1, which contains a web server whose API is completely defined by the type `HackageAPI`, from which the types of the server and client functions are determined using type-level functions.

As for using Haskell in the browser, it may seem strange at a first glance to want such a thing when JavaScript is the only language supported by Web browsers. There is, however, a growing number of languages that compile to JavaScript, which use it as their compile target instead of Assembly or LLVM, that can be done either by translating the logic of the program into JavaScript as is (transpiling), or by implementing an alternative runtime environment in JavaScript, which then interprets the byte- or source-code. Another technology that enables languages to run in the browser is WebAssembly, an alternative assembly language and a runtime designed specifically for the Web.

Web developers have been using JavaScript compilers for a long time. CoffeeScript is rather popular language announced in 2010 [3]. Also the new ECMAScript 6 or 7 features have only been usable via compilation until browsers implemented them natively. There are other, more advanced languages built with compilation to JavaScript in mind, e.g. TypeScript, a superset of ECMAScript 6 [27], or Elm, a framework with its own language based on Haskell [11]. The need to compile your code before running it is now quite accepted in the world of Web development.

The currently accepted way of running Haskell in the browser is via GHCJS, a Haskell-to-JavaScript compiler, although there are two active projects in the process of creating a Haskell-to-WebAssembly compiler: WebGHC [15] and Asterius [45].

4.2 Haskell Ecosystem for the Web

We will now go through Haskell libraries for Web development, using the same structure as we did in the chapter describing general Web framework features.

There is significant focus on the semantics of libraries in the Haskell community, e.g. writing down mathematical laws for the foundational types of a library and using them to prove correctness of the code, so UI libraries have mostly used Functional Reactive Programming (FRP) or similar approaches like the *Elm architecture* [23] as their basis, as traditional imperative event-based programming does not fit those criteria well.

There are five production-ready *UI toolkits* for the Web that I have found. Of these five, React-flux and Transient are unmaintained, and Reflex, Miso, and Concur are under active development and ready for production use. Each one uses a conceptually different approach to the problem of browser user interfaces, and they differ in their maturity and the size of their community as well.

Reflex [34] (and Reflex-DOM [35], its DOM bindings) looks like the most actively maintained and developed one. Reflex is also sponsored by Obsidian Systems [32] and is the most popular frontend framework in the Haskell community, so its future seems promising. Reflex follows the traditional FRP approach with events and behaviors, adding *dynamics*, and building a rich combinator library on top of them. There is an example of Reflex code in Listing 2, where `eClick` is an event of unit values and `dCount` is a value containing a dynamically changing integer.

Miso [19] is a re-implementation of the *Elm architecture* in Haskell, which means that it uses a strictly uni-directional data-flow with a central data store on the one hand, and the view as a pure function that takes the state and creates a view on the other hand, where the view can change the state using strictly defined events. The ecosystem of Miso is not as

```haskell
main :: IO ()
main = mainWidget $ do
  eClick :: Event t () <- button "Click me"
  dCount :: Dynamic t Int <- count eClick
  display dCount
```

Listing 2: An example of Reflex code (a counter)

well developed as Reflex's, and the overall architecture is quite limiting, which I consider to be a large disadvantage. You can see an example of Miso code in Listing 3, in which all local variables from the `where` clause are bound in the expression `App {..}`. In particular, you can see the `Action`, the `model` (a simple integer), the `update` function, and the `view`, which together form the basis of the application.

```haskell
data Action = AddOne
  deriving Eq

main :: IO ()
main = JSaddle.run 8080 $ startApp App {..}
  where
    initialAction = AddOne
    model  = 0
    subs   = []
    events = defaultEvents
    mountPoint = Nothing

    update AddOne m = noEff (m + 1)

    view x = div_ []
      [ text (ms x)
      , button_ [ onClick AddOne ] [ text "Click Me" ]
      ]
```

Listing 3: An example of Miso code (a counter)

Concur [18] tries to explore a different paradigm by combining the best of the previous two approaches. The developers have so far been focusing on exploring how this paradigm fits into browser, desktop or terminal applications, so it has a quite small range of features. It is a technology I intend to explore in the future when it is more mature, which, however, does not seem suitable for a large application so far, at least compared to its competitors. An example is included in Listing 4, where you can see the operator `<|>` used for combining widgets inside `main` and `»` for sequencing in `increment1`.

In all of these frameworks, *templating* is a feature that has been side-stepped by creating a domain-specific language for HTML mixed with control flow. There have been attempts at creating a more HTML-like language embedded into Haskell or external templates, though there is no such project that is both feature-complete and actively maintained. It is, how-

```haskell
main :: IO ()
main = do
  initConcur
  void $ runWidgetInBody $ void $ flip execStateT (0 :: Int) $
    forever $ increment1 <|> displayCount
  where
    increment1 = lift (el_ E.div [] $ button "Click Me") >> modify (+10)
    displayCount = do
      count <- get
      lift $ el_ E.div [] $ text $ show count ++ " clicks"
```

Listing 4: An example of Concur code (a counter)

ever, possible to reuse existing JavaScript components using the foreign function interface (FFI) between Haskell and JavaScript, and that it exactly what one of the unmaintained frameworks did to use React as its backend (react-flux).

State management is where the frameworks differ the most. Miso follows the Elm architecture strictly with a central data store that can be only changed by messages from the view, whereas Reflex and Concur are more flexible, allowing both centralized and component-local state. A common complaint regarding Reflex is that there is no recommended application architecture. It errs on the other side of the flexibility vs. best practices spectrum.

As for *routing*, Miso has routing built into its base library. There are several attempts at a routing library in Reflex, though the situation is the same as with templating libraries. Concur with its small ecosystem does not have routing at all, it would be necessary to implement form scratch for a production-ready application.

In *forms* and UI components in general, the selection is not good. There are several components collections for Reflex which use popular CSS frameworks (Bootstrap, Semantic UI), though each has many missing pieces and they lack components that need to be re-implemented anew in each application, forms in particular. Miso and Concur do not have any publicly available UI component libraries, or at least none that I was able to find.

Accessibility as a whole has not been a focus of Web development in Haskell. It is possible to reuse JavaScript accessibility testing tools, however, though I have not seen any sort of automated testing done on any of the publicly available Haskell applications. The only area with continued developer focus is *loading speed*, as the size of build artifacts was a problem for a long time. That has been ameliorated to the level of a common JavaScript application, however, so that is not a critical concern. *Prerendering* is also supported by Miso and Reflex, which helps speed up load times as well.

Moving on to the topic of *build tools*: there are three main options in Haskell: Cabal v2 [7], Stack [9], and Nix. There are also other options. Snack [25], aiming for the best of these three but not yet ready for production use, or Mafia [41], which is not too popular in the community at large. Cabal is the original Haskell build tool which gained a bad reputation for some of its design decisions (the so-called "Cabal hell"), though most of them were fixed in "Cabal v2" which puts it on par with its main competitor, Stack. Stack tried to bring Haskell closer to other mainstream programming language by introducing

several new features like automatic download of the selected compiler or a curated subset of the main Haskell package repository, Stackage. It succeeded in that, becoming the tool of choice for a large part of the Haskell community in the process. Nix, as mentioned in the previous chapter, is a general-purpose build tool and not a Haskell-specific one. It is especially capable for cross-compilation, however, which is the reason why it is used for frontend Haskell.

Glasgow Haskell Compiler (GHC) is the main Haskell *compiler* used for the creation of native binaries. Compilation to JavaScript, as required for frontend development, is supported by a separate compiler, GHCJS, which uses GHC as a library. Setting up a GHCJS development environment with Cabal is not a trivial process and using Stack limits the developer to old GHC versions, so it is Nix that is usually recommended. When set up correctly, Nix offers almost a one-click setup, downloading the compiler and all dependencies from a binary cache or compiling them if unavailable. Reflex especially, in the reflex-platform project [36], uses the cross-compilation capabilities of Nix to allow applications to compile for Android, iOS, desktop, or the web simultaneously.

The main problem of GHCJS has been speed and the size of the produced JavaScript. The latter has been gradually improving and is now mostly on par with modern JavaScript framework, the former is harder to improve though, and GHCJS applications are still within a factor of 3 of native JavaScript ones [30]. However, this should be improved soon by compiling to WebAssembly instead of JavaScript. There are two projects trying to create a Haskell-to-WebAssembly compiler in parallel: Asterius [45] and WebGHC [15]. They are so far in alpha, but I expect them to be production-ready by the end of 2019.

Moving on to the topic of *debugging tools*, this is where Haskell on the frontend is lacking the most. While it is possible to use the browser's built-in DevTools and their debugger and profiler, the compiled output of GHCJS does not correspond to the original Haskell code too much, which makes using the debugger quite hard. There are no other debugging tools, though in my experience I did not ever feel the need to use anything else than writing debugging output to the console.

In contrast, there are many *quality assurance* tools available for Haskell in general, of which almost all are available for use in frontend development. Starting with static quality assurance, Hlint is the standard "linter" for Haskell, well-supported and mature. There are several code formatters, Hindent is the most widely used one, which enforces a single style of code as is common in other contemporary languages (e.g. gofmt for Go). As for test frameworks, there are many options. HSpec or HUnit are examples of unit- or integration-testing frameworks, property-based testing is also common in Haskell, with QuickCheck [8] being the most well-known example. For end-to-end testing in the browser, there are libraries that integrate with Selenium.

Haskell has a quite bad reputation for the lack of *editor integration*. The situation is better with the recent Language Server Protocol project, where haskell-ide-engine, Haskell's language server, enables users to write Haskell in contemporary editors like Atom easily. The language server supports type-checking, linting and formatting, and also common IDE features like "Go to definition" or "Type at point".

Compiling applications as *mobile or desktop apps* is well-supported in Reflex, though not in Miso or Concur. Using the scaffolding of reflex-platform makes supporting different platforms almost automatic, as Nix takes care of switching between compilers: GHCJS

for the Web, regular GHC for the desktop and cross-compiling GHC for iOS or Android. Bundling the compiled applications for distribution for each platform is a bit more involved, though there are efforts to automate even that.

Code generators are quite limited in Haskell. Stack has a templating system for new project initialization, though there are no templates for frontend development so far. Cabal comes with a single standard template for a blank project but lacks customization options for creating framework-specific templates. And Nix does not do code generation at all. The common practice so far is to make copy of a repository containing the basics, edit project-specific details, and use that as a base for a new project. I have not found any attempts at component generation in Haskell.

In summary, while there are several UI toolkits available for browser applications in Haskell, individual components that are required for easy application development are either not available at all or not too well developed.

Chapter 5

Creating the framework

5.1 Implementation plan

In the previous chapter, I presented my research into Haskell and its library ecosystem for browser applications. Now it is time to select which components need to be created to fulfill the goal of this thesis, i.e. creating a framework for development of Progressive Web Applications. Here are the requirements for a Basic PWA reiterated:

- Pages are responsive on tablets and mobile devices.
- All application URLs load while offline.
- Site uses cache-first networking.
- Page transitions do not feel like they block on the network.
- Pages use the History API.
- Each page has a URL.
- Metadata provided for "Add to Home screen".
- Site appropriately informs the user when they are offline.
- Push notifications (which consist of several related requirements).

We will go through them one by one to see which components already exist and which are left to be implemented.

TODO: How CSS provides responsiveness?

Responsiveness is accomplished via CSS and is therefore out of scope, we are focusing on the JavaScript part only. The next two requirements (offline, cache-first networking) need to be implemented in a service worker, which is not covered by any existing library. Non-blocking page transitions and the use of History API are similar requirements that can today be implemented manually, but a routing component is desirable to remove the large amounts of boilerplate code necessary and to fulfill the next requirement of each

page having a URL. The metadata for "Add to Home screen" need to be specified in the Web App Manifest, which is currently not supported by any existing library, but can be created manually as well. Indication of online/offline status is supported by the basic DOM interaction library. Push notifications require three components: in the browser, in the service worker, and on the server. Only the server-side component is currently available in Haskell.

There are some features that are beneficial for a PWA but not included in the explicit list of requirements, one of them is being able to provide at least basic functionality even offline. Doing that requires either API caching (using a service worker) or offline storage, neither are supported by any existing library, however.

I have selected the components that would, in my opinion, provide a solid basis for further expansion while fulfilling our requirements. Implementing a framework that covers all features missing in frontend Haskell is a topic for a multi-year project for a team of developers, so the scope of my work is limited by the available resources, both in time and in human resources. The selected components are:

- a full-featured browser routing library,
- a service worker generator and push notification support for the client and the server,
- Web App Manifest generator, and
- a basic key-value storage library with backends for both the browser and server (to support prerendering).

These components will be usable both on their own and in combination, as a framework. While I developed these components incrementally, extracting common patterns from applications written without them, I will not describe the individual iterations but instead walk through the design choices made in the process and some interesting parts of the implementations, as I believe that will make for a more concise and informative presentation.

5.2 Routing

A router is one of the basic components of a modern web application. There are several features a router is concerned with: parsing the initial URL on application start-up, changing it according to user navigation actions, storing the navigation state for the rest of the application. In types, this might be expressed as shown in Listing 5.

```
parseRoute :: URL -> Route app
dispatchRoute :: Route app -> m ()
renderRoute :: Route app -> URL
```

Listing 5: Router API

5.2.1 Previous work

There are several widely used options for a server-side router, which has the same responsibilities as a client-side one, and a very similar interface, for the most part. These options differ in several ways, the most fundamental one being the representation of the route, which in turn defines the basis of the client API.

We will go through the routers of Yesod, Happstack, and Snap, all of them popular Haskell frameworks for server-rendered web applications, and then move on to Servant, a general-purpose routing solution for web services.

Yesod uses a special DSL (Domain Specific Language) for its router, which is implemented via quasi-quoting, a specific flavor of meta-programming where an arbitrary string is parsed into a Haskell expression. In this way Yesod generates several type-class instances, implementations of the above-mentioned functions, and a sum type containing all possible routes in an application. The route itself is then just a plain data constructor of this sum type.

Happstack and Snap both offer a choice between using non-typed routes based on strings, or type-safe routes similar to Yesod's approach above. For type-safe routing, they both use the same library, `web-routes`. To use this library, the user defines a sum type containing all possible routes in an application and then uses library combinators to define a parser/encoder manually. The parser/encoder is represented as a so-called *boomerang*, a composable object containing both directions of the transformation.

Servant is newer than the above options, and it is the most popular solution for creating web APIs in Haskell at the moment. In Servant, an API is described using a single large type in its entirety, created by composition using type-level operators (`:<|>`, `:>`). This type is then processed using type-classes to create specific types suitable for implementing a server or for creating type-safe links. This type can also be interpreted using other libraries to generate API documentation or clients in a variety of libraries.

Of these options, Servant's approach seems to be the most flexible one as is also demonstrated by the large number of libraries that build on the Servant core, although the complexity of using type operators and type interpreters may be intimidating to developers looking beneath the user-facing API, at least compared to the simplicity of the other two approaches which use plain functions and simple sum types at their core.

5.2.2 Servant

Servant is a general type-level DSL (Domain-Specific Language) in the domain of web routing. An API defined using Servant is merely a type, a tree of type-level terms composed using type operators. This API type is then interpreted using type-level functions into value-level functions, e.g. routers.

In Listing 6, we can see that a single Servant endpoint `GetUsers` is a composition of type-level strings and so-called *combinators* like `QueryParam` and `Get`, which are usually defined as data types without any constructors as shown in the first part of the listing. These endpoints are then composed together using type-level operators "then", `:>`, and "and", `:<|>`, as shown in the first part of the listing.

```haskell
data (:>) (a :: Type) (b :: k)
data (:<|>) (a :: Type) (b :: Type)
   = (:<|>) a b
data QueryParam (name :: Symbol) (a :: Type)

type GetUsers = "users" :> QueryParam "sortby" SortBy :> Get '[JSON] [User]
type CreateUser = "users" :> ReqBody '[JSON] User :> Post '[JSON] UserId
type UserAPI = GetUsers :<|> CreateUser

server :: Server UserAPI
server = (\sortBy -> return [users]) :<|> (\user -> saveUser user)

getUsers :: SortBy -> ClientM [User]
getUsers = f
  where
    (f :<|> _) = client (Proxy @UserAPI)
```

Listing 6: Servant API definition

A server implementing such an API is defined in a very similar way, the handlers for individual endpoints are composed together using the value-level operator `:<|>` (a constructor of the type `:<|>`), as can be seen in the definition of `server`. A client for the API is not created by composition but by decomposition of the `:<|>` constructor as shown in the last part of the listing.

An alternative approach to defining an API is using records. This approach uses Haskell's support for datatype-generic programming to convert between a record into a tree that uses `:<|>` on both the type- and value-level. It is easier to work with larger APIs in this way and it makes for easier-to-read type errors. It is also possible to refer to individual endpoints using record accessors, instead of (de)composition of the entire server or client. The code in Listing 7 is functionally equivalent to the previous listing.

The interpretation of an API type into values is done via type classes, a language feature that is often compared to interfaces in object-oriented languages, but in this case its use is a bit more involved. The API tree is traversed recursively from the top along the `:<|>` and `:>` operators, one combinator at a time starting from the outermost `:<|>`. In the case of a server, the API type of each endpoint is also translated into the type of the handler function using an associated type family. Despite its name, a type family defines a type-level function: "given a type of an endpoint, find the type of a handler" in this case.

We will see this process in more detail in a later chapter, when defining an entirely new interpretation of an API type in the creation of a client router, and when extending an existing interpretation to support prerendering of applications on the server.

```haskell
data UserAPI = UserAPI
  { _getUsers :: "users" :> QueryParam "sort" SortBy :> Get '[JSON] [User]
  , _createUser :: "users" :> ReqBody '[JSON] User :> Post '[JSON] UserId
  } deriving (Generic)

server :: Server (ToServant UserAPI)
server = toServant $ UserAPI
  { _getUsers = \sortBy -> return [users]
  , _createUser = \user -> saveUser user
  }

getUsers :: SortBy -> ClientM [User]
getUsers = _getUsers apiClient
  where
    apiClient = genericClient @UserAPI
```

Listing 7: Servant Generic API definition

5.2.3 Reflex

Before we dive into the implementation of the router, we also need to go through the basics of Reflex, as its philosophy and building blocks constrain the shape of any function we design.

As mentioned in the introductory chapters, Reflex is a general *Functional Reactive Programming* (FRP) library. FRP in general is a way of programming where the program consists of a network of time-varying values and functions combining such values.

The basic building blocks of FRP are events, objects which have a value only on a specific moment, and behaviors, which have a value at any point. Reflex adds a third primitive, a *dynamic*, which is a pair of a behavior and an event which fires whenever the behavior changes.

Reflex is a general FRP library, to interact with the external world it needs bindings to read external values and translate Reflex events into external actions. There are several such bindings: `reflex-dom` for the browser, `reflex-backend-wai` for the WAI web server interface, `diagrams-reflex` for SVG animations, and several others. The one we will use in the rest of this work is `reflex-dom`, which contains the necessary building blocks for web applications: functions to create and animate HTML elements, listen on browser events, or perform HTTP requests.

Reflex and Reflex-DOM provide the basic building blocks for creating applications, but they do not fall to a natural structure for bigger applications the way object-oriented frameworks do as in MVC and its variations. In fact, one of the most common complaints of developers exploring Reflex is the lack of a developed application architecture.

It is possible to recreate patterns like the Elm architecture in Reflex, as well as more fine-grained architectures that use smaller stateful components communicating each other using top-level application logic. Several patterns have emerged so far, but none has been

generally accepted so far, and the most accepted one (Gonimo architecture [21]) requires a large amount of trivial "plumbing" code.

There are, however, several smaller structural patterns that have slowly emerged as "rules of thumb". "Dynamics as component inputs, events as outputs" is one such, which has been somewhat formalized as a combination of monad transformers (`ReaderT` and `EventWriterT`) in Reflex itself.

Reflex is composed of several fine-grained typeclasses. These are abstract, and they are translated into a series of monad transformers and their interpreters on the top level.

There are several common methods of formalizing application architecture in Haskell. Each method tries to abstract implementation details from application logic by identifying all side-effects that a program requires and decomposing them into individual effects. The methods are:

- monad transformers and MTL-like typeclasses,
- ReaderT with a top-level application state, and
- effect interpreters like free or freer monads.

Each one has its advantages and disadvantages, and while they can be mostly arbitrarily intermixed, each application or library usually chooses one. The most popular in the Haskell community and used by the majority of libraries is monad transformers and MTL-like classes, which is also the method that Reflex uses.

A signature of a component in a program structured in this way would look something like Listing 8, where first two constraints of `userView` would be executed using the function `runApp`, with the remaining `MonadWidget` being executed by the top-level rendering function.

```
userView ::
     (MonadReader AppState m, MonadRouter AppRoute m, MonadWidget t m)
  => Dynamic t User
  -> m (Event t UserEdit)

runAppM :: MonadWidget t m => RouterT AppRoute (ReaderT State m) a -> m a
```

Listing 8: MTL-based API

5.2.4 Implementation

I have decided to use Servant's approach in my work, as it seems to be the most flexible and extendable one.

My contributions in this area are:

- a client-side router using Reflex's FRP types composed of a dispatch component and in-application links and

- an extension to the server-side Servant router that supports rendering Reflex applications.

I have also created a proof-of-concept of a static site generator using these components, as well as a combinator that allows easier manipulation with record-based Servant types that I will contribute to the main Servant repository.

We will start with the client-side router, defining the routes type and the handlers. This is where we will see how to create a new interpretation of a Servant API type.

A regular Servant type has endpoints that end with the terminator `Verb`, which represents a HTTP verb like GET or POST and the return type of the handler. Given that a Reflex application does not have a value that it can return, we will define a new terminator `App`. An API type containing an `App` will then be interpreted by a type class `HasApp`, as we can see in Listing 9.

There, we can see what it looks like to interpret a Servant type. The type family `MkApp` will produce the type of a route handler when evaluated. The result of the `MkApp` of a single endpoint is a function, whereas applying `MkApp` to the API type will result in a tree of route handlers, which can then be converted to/from a record of handlers.

The function `route` is the actual function used for choosing a handler based on the current location: a recursive function that will either produce an error or the handler to run when given a tree of handlers and the current location.

The first instance, `a :<|> b`, is the branch instance. The `route` function uses the monoid instance of the type `Either`, effectively running the left branch and running the right branch only if it fails.

The next instance, `Capture sym a`, is an example of a decision instance, where the `route` function processes a single segment of the URL, parses it, passes the parsed value to the handler function, and recurses. The `MkApp` instance declares this explicitly: the handler for a `Capture` needs to accept a value of type `a`.

The `App` instance is the end of the recursion chain, where neither `MkApp` nor `route` recurse anymore. The `MkApp` type declares the handler of an `App` to be an action, and the `route` function only checks that we have parsed the entire URL, and returning the final handler.

This, in summary, is what it looks like to interpret a Servant type. As for the server part of a router, I will not include the relevant code here, however, as the Servant server uses special machinery to run a series of optimizations that preprocess a tree of handlers. The resulting code is not too readable, so I do not think it is worth including it here. I will sketch what the code does, at least:

For the server part, we only need to implement a single instance, the instance of `HasServer` for our `App` type. It will be different from the client instances, not only for the above-mentioned reasons, but also due to the fact that while we have a handler action, we cannot run it directly, but we need to render it into a HTML string before sending it to the client. The Servant server uses an additional type parameter, a context using which it is possible to pass values to API combinators. The `App` instance reads a rendering function from the context, runs the route handler using it, and returns the result.

Back to the client-side router: while we have a `route` function that will return either an error or a widget, we need to connect it to the browser in some way. To do that, we need

```haskell
data App :: Type

class HasApp api where
  type MkApp api (m :: Type -> Type) :: Type
  route :: Proxy api -> MkApp api m -> Loc -> Either Err (m ())

instance (HasApp a, HasApp b) => HasApp (a :<|> b) where
  type MkApp (a :<|> b) m = MkApp a m :<|> MkApp b m
  route _ (a :<|> b) = route (Proxy @a) a <> route (Proxy @b) b

instance (FromHttpApiData a, HasApp s) => HasApp (Capture sy a :> s) where
  type MkApp (Capture s a :> sub) m = a -> MkApp s m
  route _ f loc = case locPath loc of
    [] -> Left Err404
    x:xs -> case parseUrlPiece x of
      Right p -> route (Proxy @sub) (f p) (loc { locPath = xs })
      Left _ ->
        let s = T.pack $ symbolVal (Proxy @sy)
        in Left Err400

instance HasApp App where
  type MkApp App m = m ()
  route _ f loc = case locPath loc of
    [] -> Right f
    [""] -> Right f
    _ -> Left Err404
```

Listing 9: Router: API types

a component for manipulating the URL, either using the Location API or hash fragment changes, and when we have it, we can write the router itself.

In Listing 10, we have a simplified version of the library router. In there, we have a function that takes a tree of handlers, a URL manipulation component, and an action to show possible routing errors, and produces a piece of dynamically changing content. The function uses *recursive do* to make it possible to refer to variable before they are defined (the `rec` keyword). Reading from the top, we obtain a dynamic containing the current location, use it to run our `route` function defined above, rendering any errors, and finally run this dynamically changing piece of content to get the event that changes the current URL.

```
runRouter ::
     forall t m api. _
  => Proxy api
  -> MkApp api (EventWriterT t Loc m)
  -> (Event t Loc -> m (Dynamic t Loc))
  -> (Err -> EventWriterT t Loc m ())
  -> m ()
runRouter api handlers url showError = do
  rec
    dUrl <- url eUrl
    let widget = case route api handlers <$> dUrl of
          Left err -> showError err
          Right f -> f
    ((), eUrl) <- runEventWriterT (dyn widget)
  pure ()
```

Listing 10: Router: URL binding

The second part of the router are links from one part of the application to another. To do that, we need another interpretation of the API type, as we need to process a dynamically changing input into a link, and not produce an action given a static list of parameters.

The types here are slightly more complex as I wanted to achieve an easy-to-use user interface that can be seen in the first part of Listing 11, which just needs an event with a tuple of all required parameters of the route. To achieve that, we first need to collect all route parameters, collecting them to a type-level list using the `GatherLinkArgs` type family, convert it to a tuple using the `TupleProduct` type family, and only then can we use it. The `toAppLink` function is again recursive, and it builds up a URL from the endpoint type and from the provided arguments, starting from an empty URL.

TODO: A complete example of a router: type, handlers, links

5.2.5 Possible extensions

There are several possible directions in which to expand this router. One idea available in server-side API routes is encoding authentication constraints in the endpoint type itself using a combinator like `AuthProtect User`. I would like to be able to encode not

```haskell
viewUserItemsLink :: Event t (UserId, ItemType) -> m ()
viewUserItemsLink = appLink viewUserItemsRoute

appLink ::
     forall t e rs m. _
  => (rs AsApi -> e)
  -> Event t (TupleProduct (GatherLinkArgs e))
  -> m ()
appLink _ args =
  tellEvent $
  safeAppLink (genericApi (Proxy @rs)) (Proxy @e) (Loc [] []) <$> args

class HasAppLink api where
  type GatherLinkArgs api :: [*]
  toAppLink :: Proxy api -> Loc -> TupleProduct (GatherLinkArgs api) -> Loc

instance (KnownSymbol sym, HasAppLink sub) => HasAppLink (sym :> sub) where
  type GatherLinkArgs (sym :> sub) = GatherLinkArgs sub
  toAppLink _ l = toAppLink (Proxy @sub) $ l
    { locPath = locPath l ++ [toUrlPiece . symbolVal $ Proxy @sym]
    }

instance HasAppLink App where
  type GatherLinkArgs App = '[]
  toAppLink _ l _ = l
```

Listing 11: Router: in-application links

only authentication checks but authorization checks in the endpoint type as well, perhaps `AuthProtectRole User 'RoleAdmin`.

It would be possible to expand the proof-of-concept of a static site generator that uses the routing component created here into a fully fledged library, and it would also be a continuation of the theme "Reflex everywhere" that seems to pervade the Reflex ecosystem, not only Reflex in interactive browser applications and on the server, but also static sites generated using Reflex.

A harder problem but possible more beneficial: instead of using a special `App` combinator to render Reflex applications, it might be possible accomplish the same using a special content type. This would allow one endpoint to return e.g. JSON data or a HTML file on the same endpoint, depending on the request headers. I tried this approach at the start but did not succeed, so I moved on to other approaches, but I expect that a more skilled Servant developer would find a way.

5.3 Service Workers

To reiterate the description of a service worker from the introductory chapters: it is a JavaScript script that can, among other things, intercept requests initiated by the application that installed it and respond to them from cache, redirect them to another domain, or modify their response. The worker can also listen for incoming push notifications and display them to the user, or save requests that the application made while offline and retry them whenever the device goes online, regardless of whether the application is running or not (Background Sync API).

5.3.1 Requirements

The Service Worker features that we aim to support are: precaching, fetch control, and push notifications, keeping Background Sync for a possible extension of this library.

Precaching means storing the files essential for the application into cache as soon as the Service Worker starts. This way, the application prepares to run offline. These files usually include `index.html`, the application entry point; `bundle.js` (or similar), the JavaScript bundle containing the entire application, and `bundle.css`, a file with all application stylesheets. Application icons and fonts are usually included as well, as are analytics libraries for usage tracking.

Fetch control in this context means intercepting all outgoing requests from the application, and deciding what to do with them based on the URL or method. This feature has many use-cases, e.g. using the precached application files when offline, checking for a new version of the application and notifying the user; storing external fetched resources into cache to save data, or storing outgoing analytics requests into a queue when offline and only sending them when the user later connects to the Internet.

Push notifications are the feature for which service workers are most well known. They allow the server of a web application to send notifications to any of its clients, where the application can choose to arbitrarily process the notification.

The basis of the implementation is a single dependently typed record that contains the entire configuration of the worker. This record is then used in three different contexts: to generate the worker JavaScript and serve it over HTTP, in the client for any interactions with the worker (e.g. to subscribe to push notifications), and on the server for sending the notifications, as illustrated by Listing 12.

```
generateWorker :: ServiceWorker push -> ByteString
runServiceWorkerClientT ::
  ServiceWorker push -> ServiceWorkerClientT push m a -> m a
runPushServerT :: ServiceWorker push -> PushT push m a -> m a
```

Listing 12: Service Worker API

While I had originally intended to write the service worker directly in Haskell and compile it using the GHCJS, there is an obstacle that prevents that: service workers do not run in the same way that a regular browser application does. A browser can terminate a service worker at any time to save computing resources, and restarts it when it is needed to process application events, as a service worker is expected to contain mostly just event handlers.

This is, however, at odds with the GHCJS execution model which relies on `setTimeout` or `requestAnimationFrame` to support multiple threads, asynchronous execution, and other features needed to run the entirety of Haskell in the browser. That means that we cannot use GHCJS to create Service Workers and need to generate plain JavaScript code instead.

5.3.2 JMacro

Of the options available for generation of JavaScript in Haskell, only the library JMacro is suitable for this task, as it is the only library intended for this purpose, none of the other libraries are very user-friendly.

JMacro allows the user to write plain JavaScript code embedded in Haskell via quasi-quotation, which is a method of meta-programming that makes it possible to transform arbitrary strings into Haskell expressions. The library supports the entirety of ECMAScript 3, so most existing JavaScript code can be copy-pasted without the need for changes, as long as it does not use the features of newer ECMAScript versions. JMacro is untyped, it recognizes two forms of JavaScript code, expressions and statements. It also supports injection of Haskell variables using anti-quotation. An example of JMacro code can be seen in Listing 13.

5.3.3 Implementation

Of the three features of service workers that we want to support (prefetch, fetch control, push notifications), prefetch is the simplest. It only requires adding a bit of code to the `install` event listener in which we add the required files into cache, as can be seen in Listing 14.

```
handleFetch :: JExpr -> JStat
handleFetch fn = [jmacro|self.addEventListener('fetch', `(fn)`);|]

sw :: JStat
sw = handleFetch [jmacroE|
function(evt) {
  console.log("The service worker is serving the asset.");
  evt.respondWith(fromNetwork(evt.request, 400).then(null, function () {
    return fromCache(`(cacheName)`, evt.request);
  }));
}|]
```

Listing 13: An example of JMacro

```
generatePrefetch :: Text -> [Text] -> JStat
generatePrefetch cacheName urls = [jmacro|
  return caches.open(`(cacheName)`).then(function (cache) {
    return cache.addAll(`(urls)`);
  });
|]
```

Listing 14: Service Worker: prefetch

Fetch is a bit more involved. In the onFetch handler, we need to find out if the outgoing request matches any of the configured filters, so we go through the filters in order and if a request matches, the selected cache strategy is executed.

There are many possible behaviors with regards to caching and network access. We cannot cover all possible cases, but we can cover the most common ones. These are encoded as a plain sum type in Listing 15. Most strategy names are self-explanatory, I will mention only `StaleWhileRevalidate` and its `Notify` variation: these serve the currently cached version of a resource, and attempt to fetch a newer one, which will then be stored into cache for later requests. This strategy is often used for main application files, which is the reason for the `Notify` variation, which will also notify the application itself if there is a newer version available so that the application can then notify the user.

The encoding for request matchers that I chose is not a complex one: a request is matched on its method, path, and query string. The method matcher accepts three values, any method, a specific method or a list of possible ones. There are two types of path matchers: a regular expression matcher and a path component-based one, in which e.g. the path `/article/5` is matched using the matcher `matchPath "article" ./ matchInteger`. The query string matcher is a list of key-value matchers. While this is not the most expressive or fluent encoding of a request matcher, it suffices for common use-cases of fetch control, as with the limited palette of cache strategies.

Handling push notifications is not trivial either. While using them in the most basic way is as simple as calling `showNotification` on the body of the incoming message, it is possible

```haskell
data CacheStrategy
  = CacheFirst Text
  | CacheOnly Text
  | NetworkFirst Text
  | NetworkOnly
  | StaleWhileRevalidate Text
  | StaleWhileRevalidateNotify Text
  deriving (Eq, Ord, Show)
```

Listing 15: Cache strategies

to do more, like passing the notification to the application using `postMessage`. Like with cache strategies, it is not possible to cover all possible use-cases with predefined options so again, we add the common ones. This time, they need to be encoded as a *GADT* (Generic Algebraic Data Type), an extension of Haskell data types that allows us to specialize the type of a data constructor, which we can use to specialize the types of sending and receiving functions in client and server code.

The options I have selected for the library are included in Listing 16. `Ignore` has the type `Void` as its parameter, which is an empty type that can have no valid values (excluding `undefined`), which means that it is impossible to call a sending function in server code. `Ignore` has no handler code generated in the service worker. `ViewOnly` displays a notification without any further handling. `ViewAndOpen` and `ViewAndProcess` both add another event handler that listens for the user clicking on the notification, which will open the application if closed, and switch to the application window if open but not focused. `ViewAndProcess` and `ProcessOnly` will also pass the message to the application for further processing via `postMessage`.

```haskell
data PushBehavior a where
  PushIgnore :: PushConfig Void
  PushViewOnly :: PushConfig ()
  PushViewAndOpen :: PushConfig ()
  PushViewAndProcess :: FromJSON a => PushConfig a
  PushProcessOnly :: FromJSON a => PushConfig a
```

Listing 16: Push behaviors

The rest of the service worker generation code consists of mostly boilerplate so I will skip it. The interested reader can find it on the attached data storage.

The server part of this component is made up of two parts: generating and serving the service worker code, and sending push notifications.

Serving the service worker is done by extending the Servant code for rendering values into transport formats where we create a new content type `JS` and specify how to render a service worker into `JS`. The code is quite simple and is included in Listing 17.

```haskell
data JS
instance Accept JS where
  contentType _  = "application" // "javascript"

instance MimeRender JS (ServiceWorker p) where
  mimeRender _ v = generateServiceWorker v
```

Listing 17: Service Worker: serving the JavaScript

Adding the ability to send push notifications is slightly more complex. The capability of sending a notification is specified by a type-class `PushNotify` with a single method, `sendPushNotification`. This type-class is then implemented by a `ReaderT` monad transformer that contains required configuration information like signing keys and push server URLs.

The code that demonstrates this is included in Listing 18. The `PushNotification` type is slightly simplified, there are more elements in the real type. The type class `HasPushConfig` contains one value, a lens, which can be described as a combined getter and a setter. It is used here because an application usually has only a single `ReaderT` with the entire application runtime context, so a lens is used as a getter to fetch the `PushConfig` from the context.

```haskell
data PushNotification a = PushNotification
  { pnTitle :: Text
  , pnDescription :: Maybe Text
  , pnData :: a
  }

class PushNotify a m | m -> a where
  sendPushNotification :: PushProvider -> PushNotification a -> m ()

class HasPushConfig a where
  pushConfig :: Lens a PushConfig

instance (MonadReader cfg m, HasPushConfig cfg) => PushNotify a m where
  sendPushNotification p x = do
    pushConfig <- views pushConfig
    sendPushNotificationImpl pushConfig p x
```

Listing 18: Service Worker: push notifications on the server

The browser part that receives push notifications from the service worker is very simple: we listen to the `postMessage` event and if the incoming message has the right type, we attempt to deserialize the JSON value into its Haskell equivalent, and then trigger a Reflex event.

TODO: include browser implementation listing

5.3.4 Possible extensions

The obvious follow-up work is supporting more features of service workers: fine-grained cache control with resource expiration based on its age or available storage space; or *Background Sync*, an API for retrying requests made when the device was offline whenever it goes online again, whether the application is open or closed.

Supporting more exotic use-cases is also possible next work, use-cases like communication between multiple instances of an application using the service worker as a relay, or using fetch control as a load balancer to dynamically switch between servers from which the application downloads data.

However, there is another approach that would obsolete most of the work on this component: after creating this component, I have discovered a project trying to create a typed DSL (Domain-Specific Language) for generating JavaScript, `jshark` cure:jshark. While I originally disregarded the approach of making a typed DSL instead of a library with a fixed selection of options, as the DSL would need to be able to represent arbitrary JavaScript logic, using this library (or a similar one) would allow building a hierarchy of functions hiding more and more of the underlying logic. However, as of the time of writing, this library is still unfinished, so writing a service worker builder using a typed DSL stays a project for the future.

A hypothetical example of such approach can be seen in Listing 19, which demonstrates more complex usage of fetch control, dispatching requests based on their destination (the originator of a request, e.g. \ `"style"` corresponds to a `<style>` tag or a CSS include).

```
sw :: WorkerM ()
sw = self `on` fetch $ \event -> do
  dest <- event ^. request . destination
  switch dest $ do
    case_ "font" $
      respondWith event cacheOnly
    cases_ ["style", "script", "document", "image"] $
      respondWith event networkFirst
    default_ $
      respondWith event networkOnly
```

Listing 19: Service worker using a JavaScript DSL

This approach may also be combined with code generation from WebIDL, an interface definition language for the Web [48] used e.g. in the Chromium browser, to produce an API that exactly corresponds to the underlying JavaScript one, only with strong types. Generating an API from WebIDL has a precedent in the library `ghcjs-dom`, a library that provides a strongly-typed interface to most browser APIs, which generates most of its code in this way.

5.4 Storage

A storage library can be implemented in many ways, from the simplest variations that store single values or key-value maps, all the way to a full-fledged database and query engine.

On this scale, we are aiming to create only the most basic storage library that is able to work with a map of key-value pairs of a single type, merely a building block for further expansion. This storage can then implement multiple backends: a simple in-memory map, a LocalStorage-backed store, or a set of bindings to a database.

The API of this storage is simple, as shown by Listing 20, but it can serve several purposes: as a cache, as an offline storage, or as a way to directly access a database when rendering a Reflex application on the server.

```
class MonadKVStore e t m | m -> t where
  get :: Dynamic t (StoreKey e) -> m (Dynamic t (Maybe e))
  getAll :: m (Dynamic t (Map (StoreKey e) e))
  put :: Event t (StoreKey e, Maybe e) -> m ()
  putAll :: Event t (Map (StoreKey e) e) -> m ()
```

Listing 20: Storage API

Implementing an instance for such a type is not complicated. To bind to a database backend on the server, we can run arbitrary code in the underlying monad, so if we have a function to execute database queries available e.g. using a `ReaderT` monad, we can sequentially construct the query, run it and wait for the result, and then return a `Dynamic` with the query's result.

On the frontend, we can use a combination of a `ReaderT`, to make the underlying map available for reading from anywhere, and an `EventWriterT`, to collect all `put` events.

The code is very similar to the code we saw in the implementation of the router, so I will not include it here. There are three implemented variants on the attached data storage: client-side bindings using an in-memory map or persisted using LocalStorage, and a single serve-side binding to the Persistent database library.

TODO: include listing demonstrating basic usage

There are several ways to extend this storage library. The first one is simply adding more backends, e.g. to support IndexedDB in the browser or other database engines on the backend.

The storage can also be specialized to work as a cache, which would mean extending the API e.g. with expiration, automatic or manual, so that it can support other use-cases like a function `getCachedOrFetch`.

Another option would be to expand the API to support more complicated SQL-like queries, so that it can better serve as a client-side database.

5.5 Web App Manifest

This component does not require much description. It only contains several data types, transcribed almost line for line from the official manifest specification [47], and the functions to serialize it to and from JSON.

Chapter 6

Application development

In this chapter we will go through some principles and techniques that I used while creating the case studies described in the next chapter, of which most can be applied to developing Reflex applications in general.

6.1 Design

While there are not yet many patterns specific to FRP or frontend applications, one common way to structure a Haskell application in general is the so-called *three layer cake* [37], which is as applicable to Reflex applications as to any other Haskell application. This architectural pattern describes three layers of code, where each one uses a different approach and different sort of types.

The innermost layer contains only plain data types and pure functions, it is the core of an application. This layer should be designed in such as way as to be easily testable using property-based tests or unit tests, so it should not interact with the outer world at all.

The intermediate layer consists of domain-specific effects, often written using a domain-specific language. In the specific case of a Reflex application it means extending the base monad using monad transformers, both library-provided and application-specific ones. Each function should list out only the effects it uses and not specialize the underlying monad transformer stack, so that it is possible to test such functions using other interpretations of the effects, ones that do not need the full environment of an application.

Finally the outermost, top-level layer contains the interpreters for the effects, connecting the application to the rest of the world. Testing this layer is usually done via end-to-end tests, running the full application.

There are as many approaches to designing a Haskell application as there are for any other language. One axis along which it is possible to describe possible approaches is bottom-up/top-down, where bottom-up development starts at the innermost layer, designing the entities used in an application and basic operations on them, and top-down, which starts from the simplest possible working solution (the outermost layer), slowly formalizing the effects and domain of an application.

While I used the top-down approach when initially creating the applications described in Chapter 7, we will walk through them the other way around, as top-down development is often iterative in nature and describing the individual iterations I went through would make for an unnecessarily long text.

6.2 Tools

Haskell developer tooling is often said to be one of its weakest points, and that is also true in Haskell on the frontend. While the situation is improving, the tooling is still not on par with more mainstream languages. Despite that, my personal developer experience with Haskell has been rather more pleasant than my experience when writing React.js applications in JavaScript.

What follows is a description of the specific tooling used in the creation of this thesis, both the libraries and applications described here. While all of this information is still valid as of the time of writing, there are some tools created after I started creating this thesis: Obelisk [33], a command-line tool that wraps `nix-build`, `nix-shell`, and `ghcid` for easier onboarding experience, or Lorri [46] which wraps `nix-shell` and `direnv`. I did not take the time to incorporate these tools into my workflow, but both are mostly a formalization of best practices, and so would not likely change much.

The central tool of this workflow is Nix [12], described as a purely functional package manager with a focus on reproducibility and isolation. Packages built using Nix are compiled in a sandbox and immutable afterwards. Dependencies are tracked per-package, multiple versions of a shared library can be safely used in parallel. There are other tools built on top of Nix: NixOS, a declarative operating system, and NixOps, a cloud deployment tool [13], but the main reason we will use Nix is the ease of setting up a cross-compiling toolchain, for compiling to JavaScript or Android/iOS.

Nix contains several command-line tools, of which two are interesting to us. The tool `nix-build` evaluates the recipe for a package (called a derivation) and executes it, in our case producing a Haskell binary or a JavaScript bundle. The second tool, `nix-shell`, evaluates a recipe for a package, builds all dependencies and build tools, and starts a terminal session with specially crafted environment variables that has all tools and dependencies available.

Nix has a large repository of package definitions called `nixpkgs` [10], which among other contains the definitions of several versions the GHC compiler including GHCJS and of most Haskell packages. It is possible, among other things, to build a single packages using multiple versions of the compiler by simply varying a `nix-build` command, or to add arbitrary build logic like "use this set of flags for GHCJS and add an extra native dependency when cross-compiling to Android".

Reflex-platform is a set of extensions to `nixpkgs`, which includes a set of overrides that work together well for building a single package for the Web and mobile (Android and iOS), as well as a set of Nix functions for working with multi-package projects (`project.nix`). These functions also make it easy to start a `nix-shell` with additional build tools. One notable example is Hoogle, an API search engine for Haskell that indexes all dependencies used in a project.

To be more specific, a project will contain one file `default.nix` that calls the `project.nix` function of reflex-platform with all Haskell packages in the project and any possible package overrides, like using a code from a remote Git repository or using an older version of a package. This file `default.nix` is then used by all Nix commands invoked in the directory that contains it.

A command like `nix-build -A ghcjs.my-project` will then produce a directory `result/` with the result of the Nix build recipe, a set of JavaScript files and a file `index.html` in the case of GHCJS. Calling `nix-build,` however, runs many steps by default: compiling object code and profiled object code, generating API documentation, and linking any executables into binary files or JavaScript bundles, together with any other user-specified post-processing steps like compressing the generated JavaScript code using `closure-compiler`. Running all of these steps is quite slow though, so we use different tools for compilation during development.

Inside a `nix-shell`, we have tools like `ghci` or `cabal repl`, interactive Haskell interpreters that can quickly load source code. We can use these tools to simply reload any changed files while skipping unchanged ones, which is a lot faster than compiling the entire package from scratch.

We can go a step further and set up a background process that watches the source code for a project for any changes and reloads them whenever any file changes, and optionally calls a function if the files load without any compile errors. This means we can have e.g. a development web server that is always running the latest code. This functionality is implemented in a tool called `ghcid` ([28], "GHCi daemon"), and it is now so common in the Haskell community that some developers report that Vim and ghcid are the only two tools they need.

Such a setup makes developing a lot faster, especially given that it is possible to run browser applications in the same way using the library `jsaddle-warp`. It works around the slow compile times of GHCJS by using the GHC interpreter and using a specific execution model in which as much code as possible is executed natively in GHCi, and only the necessary parts are executed in a browser which is connected to the server running in GHCi by WebSockets.

A tool that makes working with Nix shells easier is `direnv` [1], which is a general tool that changes the environment variables in a terminal according to the directory into which a user navigates. In a Nix project specifically, is the file `.envrc` with the contents `use nix` exists at the root of the project, a `nix-shell` is loaded whenever a developer navigates into the project directory or any of its subdirectories.

The editor I use, Emacs, uses all of the components described above to provide a full-fledged Haskell development environment. Using `direnv-mode` and `dante-mode`, two Emacs extensions, the editor loads the nix-shell immediately after opening a file in a project, and starts a ghci process in the background to check the file for any errors or warnings, which are then reported on the relevant lines. Other editors like Visual Studio Code or Atom are also supported using the Language Server Protocol and its Haskell server Haskell IDE Engine.

The interested reader can try editing Haskell in a preconfigured Emacs or Visual Studio Code using the Nix expressions in the directories `src-snippets/editor-emacs/` or `src-snippets/editor-vscode/` in the included files.

TODO: Niceties: hlint for code style and avoiding dumb errors, it is possible to extend using project-specific rules/suggestions; hoogle for easy access to the correct versions of API doc.

6.3 Workflow

TODO: When starting a project, create a project skeleton - *.cabal file, default.nix and possibly a submodule of reflex-platform, .envrc for direnv support, and source code stubs (src/Main.hs).

TODO: include a listing of src/default.nix for a multi-package TODO: include a `$ tree` of a project directory of full-stack app (src-bin/main x src/Project.hs (main)

TODO: ghcid + cabal new-repl for testing localhost server, included as the script ghcid-here in `src-snippets/ghcid-here/`.

TODO: If external services are necessary, use the orchestration service needed - NixOS containers, NixOps machines, Docker containers - according to the deployment strategy. It is also possible to run a database without a container, SQLite or gargoyle for postgres.

TODO: Regarding code structure:

- splitting frontend, common, backend packages
- common = entities and logic, inner layer of 3layer + API definition, routes, ...
- src-bin/main.hs only a redirect to src/Project.hs (main)
- project-specific Prelude

TODO: Components:

- sketch HTML, then animate/bind events
- prefer top-down data flow, do not often use top-level storage/... constraints => more reusable components
- Dynamic inputs, Event outputs
- think hard about whether a piece of state should be local or global, it makes reuse harder
- limit recursive-do blocks, they lead to hard-to-debug errors ("causality loop")

6.4 Deployment

The options for deploying a Haskell program are generally the same as deploying programs in any other compiled language that does not use intermediate object code like Java. The two most popular options in the Haskell community, not including Nix, are: deploying a statically linked executable file, and using Docker containers.

As we use Nix as our build tool, there are a few other options. If we have Nix available at the target machine, we can simply run `nix-build` and copy the package and all its dependencies to that machine using `nix-copy-closure`.

If Nix is not available at the target machine, we can build a static executable or produce a container. We can also build truly static executables that do not depend on the target machine's glibc standard library by using the musl overlay of nixpkgs, simply by replacing `pkgs` with `pkgsMusl` in the project's Nix files. To build a Docker image, we can use the nixpkgs function `pkgs.dockerTools.buildImage`.

Also, using Nix to build packages gives us the option to use NixOps as an orchestration tool, which is a way of managing NixOS systems across a variety of different cloud providers, from Amazon Web Services to Google Compute Engine. An example of an expression that deploys a simple web server can be seen in Listing 21.

```
{
  network.description = "Web server";

  webserver = { config, pkgs, ... }: let
    appPkgs = (import ./release.nix { inherit pkgs; });
  in {
    networking.firewall.allowedTCPPorts = [ 80 ];
    services.nginx.enable = true;
    services.nginx.virtualHosts.default.locations."/" = {
      proxyPass = "http://localhost:3000";
    };
    systemd.services.app-server = {
      wantedBy = [ "multi-user.target" ];
      serviceConfig.ExecStart = "${appPkgs.server}/bin/server";
    };

    deployment.targetEnv = "virtualbox";
  };
}
```

Listing 21: NixOps deployment

When deploying a web server, there is also the need to deploy static files as well, assets like application style sheets or icons. An important question here is whether the assets will be served by the same server as the application. If no, we need to produce two or more packages in the build process, which will be deployed separately. If yes, we can again produce assets as a separate package, but we can also bundle them into the same package using an additional Nix build recipe.

For a GHCJS application, such a post-processing step is nearly mandatory, as the JavaScript files produced by the compiler are rather large (over 5.9 MB for a simple Reflex application), but processing them with a minification tool like `closure-compiler` and further shrinking them with a GZIP compressor reduces the size to a reasonable size (1.9 MB minified and 350 kB compressed for the same application).

Chapter 7

Case studies

In this chapter we will go through three Reflex application in the order of increasing complexity, applications that use the components created in previous chapters. I have used applications with publicly available specifications that are intended to help developers compare frontend web frameworks. The interested reader can compare the implementations created here and the implementations available for comparison side-by-side, but in this chapter, we will only go through the basics of each case study, the overall structure and interesting parts of each application.

The applications chosen are:

- TodoMVC, a to-do list application (storage and routing components),
- HNPWA, a reading application for the news platform Hacker News (routing and service worker components), and
- RealWorld, a simplified version of the publishing platform Medium (storage, routing, and service worker components).

7.1 TodoMVC

There is an abundance of web frameworks, and there are several projects that aim to give developers a side-by-side comparison of them. Out of these, the original and most well-known one is TodoMVC [40], which is aimed at "MV* frontend frameworks". There are currently 64 implementations of their specification, although some frameworks are represented multiple times.

We will start with TodoMVC as it is the simplest of the three. TodoMVC is, as the name hints, a web application for managing a to-do list. It is not a complex project but it is intended to exercise fundamental features of a framework: DOM manipulation, forms and validation, state management (in-memory and in LocalStorage), and routing.

Going from the bottom up, the definition of a task is as simple as possible: a task consists of a title, a binary value indicating whether it is complete, and according to the specification, a task saved in persistent storage also needs a unique identifier. One possible representation

is having a task be a two-member record and the application state a mapping from an integer to a task, as shown in Listing 22.

If the tasks were to be also transferred from/to a server and saved in a database, the record would look quite different: the identifier might be a UUID (Universally Unique Identifier), the entity would likely contain information about when and who created or modified it, but considering this is a client-only application that does not need this kind of complexity, we can use the simplest possible solution.

```
data Task = Task
  { title :: Text
  , completed :: Bool
  } deriving (Eq, Ord)

type DB = Map Int Task
```

Listing 22: TodoMVC entities

Further describing the application domain, we can now define the operations on these entities. They follow the acronym CRUD (Create, Read, Update, and Delete): create a task, read the task list, update the task title or completion status, and delete a task. There are also several more specific operation required by the application specification: read a subset of tasks (all, active, or completed), toggle all tasks' completed status, and delete all completed tasks.

These operations can all be implemented using plain functions over a task or a map of tasks, which will then be tied into the storage component implemented in the previous chapter. The implementation of the operations is not particularly interesting, the interested reader can look them up in the attached files (`src-demo/todomvc/src/Main.hs`).

The HTML structure of the application is given by the specification, and contains three natural sections: an input for creating new tasks at the top, a task list for editing or deleting existing tasks, and a navigation bar at the bottom.

As this the first application we are going through, we can look at a single component in more detail: the component `newTaskBox` is included in Listing 23, where we can see what a simple GUI component in Reflex might look like. The functions `el` and `elClass` generate static HTML elements, the `inputElement` function generates an `<input>`, and the last line prepares the return value of the function, an event containing the current value of the text box whenever the Enter key is pressed in it. The `rec` mark the beginning of a *recursive-do* block where it is possible to use variables before they are bound, which is translated into a fixpoint computation by the compiler (in the form of `fix (\out -> do ...; return out')`. This is necessary as the text box needs to be emptied when we press Enter in it, as can be seen on the last but one line of the listing.

The next component, `taskListItem`, is slightly more complicated. According to the specification, double-clicking on a list item switches it to *edit mode*, which is, however, not captured anywhere in the global application state. When in edit mode, the component may revert to previous state when the Escape key is pressed, or update the task with new contents of the text box when Enter is pressed instead. This is achieved by adding a small

piece of state in the component, a boolean value saying whether this task is currently being edited.

The code of this and the remaining components is not too interesting, so we can skip directly on the next step, the structure of the application's monad transformer stack.

```
newTaskBox :: MonadWidget t m => m (Event t Text)
newTaskBox =
  elClass "header" "header" $ do
    el "h1" (text "todos")
    rec
      textbox <- inputElement $ def
        & elementConfig . initialAttributes .~
          ("class" =: "new-todo" <> "autofocus" =: "autofocus" <>
           "placeholder" =: "What needs to be done?")
        & inputElementConfig_setValue .~ ("" <$ keypress Enter textbox)
    return . ffilter T.null $
      T.strip <$> current (value textbox) <@ keypress Enter textbox
```

Listing 23: TodoMVC: New task box

The application uses two global pieces of state, the task list, persisted to LocalStorage on each change, and the router. While the order of the monad transformers may matter in some special cases like `ExceptT` or `ContT`, the exception and continuation transformers, neither of the transformers we want to use, `StorageT` and `RoutedT`, affect program flow, they simply add new capabilities to the base monad. Also, neither transformer uses the capabilities of the other, which means we can nest them in an arbitrary order. The newtype of the resulting monad can be seen in Listing 24, as is the type synonym that contains most constraints needed in application code.

The top-level interpreter of the monad is also included. As we can see, we unwrap the `AppT` from the outside, starting from the newtype wrapper, running the router, and running the storage last. The router is not interpreted with the default interpreter that uses the Location API as, according to the specification, we need to route using the hash fragment only (the part after `#` in e.g. \ `http://localhost/#!/active`). The storage needs to be persisted from and to LocalStorage, so we do not use the simple in-memory interpreter, but `runLocalStorageT` instead.

TODO: screenshot

This concludes the TodoMVC application, implemented according to its specification. If compiled using GHCJS, we get a bundle of JavaScript files and an `index.html` which is the entry point. As described in Chapter 6.4, we can add a post-compile step that compresses these files and adds any necessary assets like CSS or icons. It is possible to go a step further and include a service worker using the service worker component implemented in this work, but that is what we do in the next application.

```
newtype AppT t m a = AppT
  { unAppT :: RoutedT t AppRoute (StorageT t Task m) a
  } deriving ( Functor
             , Applicative
             , Monad
             , MonadRouted t AppRoute
             , MonadStorage t Task
             )

type AppM t m =
   ( MonadRouted t AppRoute
   , MonadStorage t Task
   , DomBuilder t m
   , PostBuild t m
   )

runAppT :: _ => AppT t m a -> m a
runAppT = runLocalStorageT . runHashRoutedT . unAppT
```

Listing 24: TodoMVC: Base monad transformer

7.2 HNPWA

HNPWA [39] is a client for Hacker News, a technological news site. Unlike TodoMVC, HNPWA does not provide a rigid specification and consists only of a rough guideline of what to implement. The task is to create a Progressive Web Application that displays information from a given API. This application has 42 implementations, a smaller number than the number of implementations of TodoMVC but it still provides a good comparison for a frontend framework for PWAs.

To describe the functionality of the application more, we will be fetching data from the official Hacker News API and displaying it. We need to display article lists, article details with comments, and user details.

The specification of the application is not as well defined, it only consists of a text document describing the desired functionality. In particular, it does not include the HTML structure and CSS styles of the application unlike TodoMVC, so I have used the HTML and CSS from one existing implementation of HNPWA, PreactHN [5].

Moving on to the actual implementation of the application, we will again start with the entities and operations on them. The official API from which we will be fetching data has a textual description of the entities, which describes only two entities, a user, and an "item" that can represent either a top-level post or a comment. The items form a tree that we will need to traverse and recursively fetch.

Some implementations of the HNPWA assignment have included their own server preprocessed the data from the official API, as it is not too suitable for direct consumption: we need to fetch an item before we know what are its children. While that would be unusable in a production application, it is not such a big problem in a demonstration application.

We will put only a single layer between the application and the API, and that is the service worker cache. We can cache all responses to the API requests and return the previously fetched response on repeated requests. The service worker will also be used for prefetching and caching the core application files, so that it fulfills the assignment requirements and is available offline.

As for the other components, we will use multiple stores again (for item lists, items, and users), and also the web app manifest generator and the router, this time the Location API-based interpreter.

There is not much else to write about the application's component structure as it is very similar to the structure in TodoMVC, there are again larger components that work with the top-level application state and small components with dynamics as their inputs and events as outputs.

TODO: screenshot

The resulting application is a valid PWA that works offline, showing cached data. Its only major deficiency is that it uses the official hierarchical API and therefore it loads content gradually and not as fast as it could if it had a dedicated server that preprocessed the data into a more suitable format.

If we had a server, we could also implement another feature, prerendering. By using the server instance of the storage component, we would be able to generate fully filled-out HTML in such a way that even browsers without JavaScript support would be able to interact with the application without obstacle.

7.3 RealWorld

RealWorld [43] is the most complex of the comparison projects. It is a clone of Medium, an online publishing platform, so it requires everything a "real world" application would. This comparison project also contains a server component, it is not a comparison of only frontend web frameworks. The numbers of implementations are: 18 frontends, 34 backends, and 3 full-stack implementations. The three full-stack implementations include both frontend and backend components, and are usually written in frameworks that have special communication channels between them and thus cannot use other backend implementations.

The task is split into a backend component that is defined by an API specification, and a frontend component defined by a number of tasks that it needs to support and a HTML structure. There is a number of features that the application must have: JWT (JSON Web Token) authentication with registration and user management, the ability to post articles and comments, and to follow users and favorite articles.

One implementation note: while the specification includes an API specification in the form of an OpenAPI file, I did not find a server that fulfills it exactly so I chose the Scotty server written in Haskell and adapted the frontend to its inaccuracies.

A large benefit of having a machine-readable API specification is that we can use it to generate the client for it. Using the tool `swagger-generator`, we can get the definitions of all entities and API endpoints, In this application we do not need to change any entities, so this suffices for out purposes.

Moving on to the capabilities the application requires, we need to persist the user access token if the user is currently logged in. While the storage component is a key-value storage not really meant for single values, we can use a map with text keys and text values instead. If we wanted to, we could use two more stores for articles and comments, but considering the scale of the application, using only the service worker cache is an easier approach.

The routing and service worker components are the same as in the HNPWA application, routing using Location API and caching requests to the API.

TODO: interesting components? TODO: screenshot

The created application fulfills all requirements of the specification and of the PWA checklist. As with HNPWA, it would be possible to improve the application by using a custom server for prerendering and for unifying the Servant types of the API and the application into a single large type.

Chapter 8

Conclusion

In this work, I have led the reader from a general introduction to modern Web technologies, through an overview of the capabilities of contemporary Web frameworks, to an analysis of the capabilities of Haskell on the frontend and specifically the state of available features in its library ecosystem.

In the second half of this work, I have designed and implemented three components, a router, a service worker generator with supporting libraries, and a key-value browser storage library, that together make a significant contribution to the ecosystem of Haskell on the frontend. These components do not comprise a framework equivalent to most popular JavaScript frameworks, but they enable creating Progressive Web Applications in Haskell, which was the set goal of this work.

8.1 Future work

The work that needs to immediately follow the submission of this thesis is publishing the components created here and seeking feedback from the Haskell community. This includes fulfilling all the formal requirements necessary for publishing the individual packages to Hackage, the package repository for Haskell, and writing up their documentation in two tiers: API documentation and user manuals. For the manuals and showcases, I will likely reuse some of the case studies presented in the previous chapter.

I expect to spend some time adapting my work according to any feedback from the community: expanding documentation, creating adapters to other libraries, implementing more requested functionality, and other necessary work.

With the libraries implemented in this work, there is, however, still a number of capabilities that Haskell lacks, compared to developing browser applications in JavaScript:

- a palette of pre-built GUI components,
- internationalization,
- a unified command-line interface to build tools,
- code generation, and

- debugging tools for the frontend, e.g. variable watching, inspecting application state

There is also a number of other ideas with various usefulness that would make building web applications in Haskell easier. Some are natural extensions of the implemented components, others are independent projects that implement other functionality that would make building web applications in Haskell easier. What follows is an incomplete list of such project topics:

- CSS-in-Haskell (similar to CSS-in-JS),
- crash reports (traceback, application state) for the browser,
- end-to-end tests that can run assertions on both the client and server,
- dynamic user-provided content, i.e. HTML-like markup that can use preregistered named components, a user-friendly editor,
- typed components that use assets, like `<img>` or `<link>`,
- forms: a set of components, validation, automatic derivation from a datatype,
- a query language for browser storage, using IndexedDB,
- automatic synchronization for browser storage,
- authentication in the router: "user is logged-in", "user has role X", "user can perform action Y",
- HTTP/2 Push support on the server: sending all necessary assets together with the first request,
- WebIDL and a JavaScript-generating DSL for service workers,
- effect system for Reflex, as a more flexible extension mechanism, and
- serializable effects that can be interpreted both in the browser or on the server if the client is missing required data.

To summarize this work, I have studied the current state of Haskell on the frontend, expanded the library ecosystem with three new additions, implemented a number of example applications, and suggested follow-up projects to remedy the remaining deficiencies compared to the features available in JavaScript.

Bibliography

[1] direnv/direnv: Unclutter your .profile.
Retrieved from: https://github.com/direnv/direnv

[2] The State of JavaScript 2018: Front-end Frameworks - Overview.
Retrieved from: https://2018.stateofjs.com/front-end-frameworks/overview/

[3] Ashkenas, J.: CoffeeScript hits 1.0 – Happy Holidays, HN | Hacker News.
Retrieved from: https://news.ycombinator.com/item?id=2037801

[4] Bar, A.: What Web Can Do Today.
Retrieved from: https://whatwebcando.today/

[5] Baxter, K.: PreactHN.
Retrieved from: https://github.com/kristoferbaxter/preact-hn

[6] Bernardy, J.-P.; Boespflug, M.; Newton, R. R.; et al.: Linear Haskell: practical linearity in a higher-order polymorphic language. *arXiv preprint arXiv:1710.09756*. 2017.

[7] Cabal Team: The Haskell Cabal.
Retrieved from: https://www.haskell.org/cabal/

[8] Claessen, K.; Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*. vol. 46, no. 4. 2011: pp. 53–64.

[9] Commercial Haskell: The Haskell Tool Stack.
Retrieved from: https://docs.haskellstack.org/

[10] community, N.: NixOS/nixpkgs: Nix Packages collection.
Retrieved from: https://github.com/NixOS/nixpkgs/

[11] Czaplicki, E.: Elm: Concurrent FRP for Functional GUIs. 2012.

[12] Dolstra, E.: *The purely functional software deployment model*. Utrecht University. 2006.

[13] Dolstra, E.; Löh, A.: NixOS: A purely functional Linux distribution. In *ACM Sigplan Notices*, vol. 43. ACM. 2008. pp. 367–378.

[14] Eisenberg, R. A.: Dependent types in haskell: Theory and practice. *arXiv preprint arXiv:1610.07978*. 2016.

[15] Fancher, W.: WebGHC.
Retrieved from: https://webghc.github.io/blog

[16] Google: Progressive Web App Checklist | Web.
Retrieved from:
https://developers.google.com/web/progressive-web-apps/checklist

[17] Hudak, P.; Hughes, J.; Peyton Jones, S.; et al.: A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. New York, NY, USA: ACM. 2007. ISBN 978-1-59593-766-7. pp. 12–1–12–55. doi:10.1145/1238844.1238856.
Retrieved from: http://doi.acm.org/10.1145/1238844.1238856

[18] Jain, A.: ajnsit/concur: An unusual Web UI Framework for Haskell.
Retrieved from: https://github.com/ajnsit/concur/

[19] Johnson, D.: dmjio/miso: A tasty Haskell front-end framework.
Retrieved from: https://github.com/dmjio/miso/

[20] Jones, S. P.: *Haskell 98 language and libraries: the revised report*. Cambridge University Press. 2003.

[21] Klotzner, R.: Gonimo Architecture.
Retrieved from: https:
//github.com/gonimo/gonimo/blob/master/front/doc/Gonimo-Architecture.md

[22] Lab, Q. F.: Queensland FP Lab - Home.
Retrieved from: https://qfpl.io/

[23] Loder, W.: *Web Applications with Elm: Functional Programming for the Web*. Apress. 2018. ISBN 9781484226100.
Retrieved from: https://books.google.cz/books?id=KnhqDwAAQBAJ

[24] Marlow, S.: Fighting spam with Haskell. 2015.

[25] Mattia, N.: nmattia/snack: Nix-based incremental build tool for Haskell projects.
Retrieved from: https://github.com/nmattia/snack/

[26] Microsoft: Language Server Protocol Specification.
Retrieved from:
https://microsoft.github.io/language-server-protocol/specification

[27] Microsoft: TypeScript - JavaScript that scales.
Retrieved from: https://www.typescriptlang.org/

[28] Mitchell, N.: ndmitchell/ghcid: Very low feature GHCi based IDE.
Retrieved from: https://github.com/ndmitchell/ghcid

[29] Mozilla: Using Service Workers - Web APIs | MDN.
Retrieved from: https://developer.mozilla.org/en-US/docs/Web/API/
Service_Worker_API/Using_Service_Workers

[30] Nanda, S.: Benchmarks: GHCJS (Reflex, Miso) and Purescript (Pux, Thermite, Halogen).
Retrieved from: https://medium.com/@saurabhnanda/benchmarks-fp-languages-libraries-for-front-end-development-a11af0542f7e

[31] Nanz, S.; Furia, C. A.: A Comparative Study of Programming Languages in Rosetta Code. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. May 2015. doi:10.1109/icse.2015.90.
Retrieved from: http://dx.doi.org/10.1109/ICSE.2015.90

[32] Obsidian Systems: Obsidian Systems | Cutting edge software solutions.
Retrieved from: https://obsidian.systems/

[33] Obsidian Systems: obsidiansystems/obelisk.
Retrieved from: https://github.com/obsidiansystems/obelisk/

[34] Obsidian Systems: reflex-frp/reflex.
Retrieved from: https://github.com/reflex-frp/reflex/

[35] Obsidian Systems: reflex-frp/reflex-dom.
Retrieved from: https://github.com/reflex-frp/reflex-dom/

[36] Obsidian Systems: reflex-frp/reflex-platform.
Retrieved from: https://github.com/reflex-frp/reflex-platform/

[37] Parsons, M.: Three Layer Haskell Cake.
Retrieved from:
https://www.parsonsmatt.org/2018/03/22/three_layer_haskell_cake.html

[38] StatCounter: Desktop vs Mobile vs Tablet Market Share Worldwide.
Retrieved from:
http://gs.statcounter.com/platform-market-share/desktop-mobile-tablet

[39] TasteJS: HNPWA.
Retrieved from: https://hnpwa.com/

[40] TasteJS: TodoMVC.
Retrieved from: http://todomvc.com/

[41] The Haskell Mafia: haskell-mafia/mafia: Provides protection against cabal swindling, robbing, injuring or sabotaging people with chopsticks.
Retrieved from: https://github.com/haskell-mafia/mafia/

[42] The JAMstack team: JAMstack - JavaScript, APIs, and Markup.
Retrieved from: https://jamstack.org/

[43] Thinkster: gothinkster/realworld: "The mother of all demo apps".
Retrieved from: https://github.com/gothinkster/realworld

[44] Tweag: Tweag I/O - software innovation lab.
Retrieved from: https://tweag.io/

[45] Tweag: tweag/asterius: A Haskell to WebAssembly compiler.
Retrieved from: https://github.com/tweag/asterius

[46] Tweag.io: Introducing Lorri, your project's nix-env.
Retrieved from:
https://www.tweag.io/posts/2019-03-28-introducing-lorri.html

[47] W3C: Web App Manifest - Living Document.
Retrieved from: https://www.w3.org/TR/appmanifest/

[48] W3C: WebIDL Level 1 - W3C Recommendation.
Retrieved from: https://www.w3.org/TR/WebIDL-1/

Appendices

Appendix A

Contents of the attached data storage

TODO: fill in