Utrecht University, Computing Science, year 2019-2020

Advanced Functional Programming

Project Report: Website Notifier

Luuk Glorie
5972868
l.glorie@students.uu.nl

Hanno Ottens
5980933
h.p.ottens@students.uu.nl

Marco van de Weerthof
5871476
m.vandeweerthof@students.uu.nl

Introduction

The goal of this project was to make a web scraper that notifies users (e.g. students) of important changes on a website when this is not supported. The main motivation was our own experience with these information sites over the past couple of years. Even though it is not much trouble to check the websites by hand, doing so every day can be cumbersome, not even talking about possible caching issues which may cause confusion.

In this document we hope to inform you about our project, the techniques we used, the troubles we ran into, how the application turned out and what we think we could have done better.

The problem

At the moment, students have to check the course sites themselves for updates every now and then. Some courses use Blackboard instead, where students are notified when something new is uploaded. However, not all courses use Blackboard and sometimes Blackboard is less convenient than a regular course website.

When following multiple courses, the student also has to check for updates, deadlines or assignments on multiple websites. Not all websites have the same layout, so looking for the right information can be cumbersome. A better overview can be given when this information is displayed on one page.

To solve this problem, we made a web application. This application consists of two parts, the web server and the web scraper. The web server handles the interaction with the user and stores information about the users, websites and targets. The web scraper regularly downloads the targeted webpages, checks for changes and notifies the users which are subscripted to that target if a change has occurred.

For the different entities in the problem, we modeled them as data structures in our project the same way they are stored in the database. The users of the application can register and login with their username and password. They can add targets for which they will be notified when there is an update. Targets consist of a website, a possible selector and extra information like the hash of the target contents and the user who owns the target. Websites contain the URL, hash of the entire webpage and the time the website was updated. To be able to send notifications to users, we have to store information about the endpoint and details to prove that we are authorized to send notifications to a certain user. We also store all notifications that have been sent, so the user has an overview of all notifications. These notifications contain the body of the message and the time they were sent at.

Concepts and techniques

Servant

For serving the API, we use Servant[5]. Servant gives us the ability to define a strictly typed API. It also allows us to set up authentication to protect parts of the API with user sensitive data.

We use cookies and XSRF tokens to manage authenticated requests. This makes sure that every request needs a unique token to be verified. This unique token is returned from the login action and every authenticated request afterwards. The XSRF token helps against XSRF attacks[8], where an attacker steals the authentication token. This way each token is only valid once.

We set up the file structure to mimic the MVC style[7]. The files in the `Handlers’ folder handle requests like controllers, and the `Models’ folder speaks for itself. However, we do not create dynamic HTML on the server side. The structure of our API allows users to access all pages, after which any dynamic data is loaded using AJAX calls and processed with JavaScript. This omits the need to construct dynamic HTML, but is not type safe.

Monad Transformers

Throughout our entire application, we wanted to have a general configuration. To do this, we created a Config object which is passed around with a ReaderT monad transformer called ‘AppConfig’. With a monad transformer, we can use the Servant functions ‘serveWithContext’ and ‘hoistWithContext’ to wrap it around all the server instances.

For authentication, we did not want to pass the user object to every single action. Therefore, we transform the AppConfig monad with a StateT transformer when a user gets access to the protected part of the API. This action is then run with Servant’s functions and the ADT of the logged in user.

Sqlite

The database we used in the project is SQLite[4]. The sqlite library provides an interface to execute queries on a database. We decoupled our database functions, so all functions take a connection parameter and make changes on the connection. Another function can execute these functions by creating a connection to the database file specified in the configuration. Different database functions are defined to add, delete or edit one of the data structures in our model. Each function has a SQL statement as string, which is prepared by the database and applied with the required arguments. All interactions with the database are not strictly typed when using this library, therefore errors can occur if the database model is not equal to our in-code model.

All SQL tables are defined in a separate file (sqlite.tables). The tables include all entities described in the problem section. When starting the program, the file is read and for each create table statement, a query is executed which adds the table if it does not exist already.

Polling & scraping

For polling the websites, we use CRON-jobs[6]. CRON-jobs are great for defining tasks that need to be repeated at intervals. A CRON-job is configured by a simple string that defines at which specific minute, hour, day etc. a task has to be performed[1]. This gives us fine grained control over when to poll websites. Users cannot define this interval, and it is defined once for all targets.

When a website is polled, we first get the site content using Haskell's networking client. We then compute the hash of the entire webpage. If the hash does not match the one found in the database, we update the hash and check every target that is aimed at that website. If the hash is the same, there cannot have been a change on the website unless a very rare hash collision occurred, and we do not proceed to check the targets.

When a user does not define a specific target on a page, we send a notification that something on the website has changed. If the user did specify a target, we convert the HTML into its tags using the TagSoup library[2]. We then parse the selector using the CSS selector standard and find the corresponding element.

From this element we scrape all displayed text (no comments, tags etc.) and concatenate it together using newlines. This text is then hashed and compared against the old hash from the database. If this hash does not match, we compute the difference and add all differing lines under three groups: added, changed and removed. These three headers are also displayed in the notifications (if applicable) with the corresponding lines. If there is no change, we do not send out a notification.

Web-push

To send out the notifications, we use the Notification API[3]. This API is supported by all modern browsers and gives us an easy way to allow users to subscribe to the notification feed.

To set it up, we first needed to request our private keys, which only has to be done once. The keys are saved in a text file and read in after each startup. When a user wants to receive notifications for the subscribed websites, all he needs to do is press the subscribe button (and give the site permission to do so).

After a user presses the subscribe button, the server and the client exchange keys. The client sends his private key and the server replies with the public server key. The web browser then handles all client side notifications. Whenever we need to send a notification, we only need to make a request to the Notifications API.

Testing

To test the different endpoints of the API, we used the Hspec library to make unit tests. Because each unit test requires a certain state of the database, we use a test database where all tables are reset in between different tests. To access the API endpoints, all specifications require the web server as an Application instance. In each specification, we can then generate get and post requests, which are handled by the server. To test if the server responds correctly, the specifications contain expectations for the response code and response body, such as JSON data. To test protected endpoints, a user is first registered and then logged in for each request in the specification. The Hspec library for testing such applications does not keep the session state such as cookies for each consecutive request, therefore the user is logged in every time before each request[10].

The scraper has also been unit tested. A test html string has been created which contains many possible caveats (nested elements, text inside tags, comments etc.). This string is then scraped into all possible targets, with and without attributes or with non-existing attributes. The scraper assumes that the HTML is properly written, so no missing end tags. We did not test the scraper on malformed HTML, because the notifications which would be sent out by those are usually non-intelligible.

Results

From the proposal document, we had the following goal:

The user should be able to set up targets that can be scraped. Options for targets should include the following:

1. *An entire website: e.g. cs.uu.nl/docs/vakken/afp/;*
2. *A specific page: e.g. cs.uu.nl/docs/vakken/afp/index.html;*
3. *An HTML-element: e.g. `#content > ul' on a website/page;*
4. *A new HTML-element: e.g. a new 'li' in a specific 'ul'.*

Of these four points we implemented point 2, 3 and 4. We found out that even though it is possible to scrape every page on a website, it was not very practical to handle the output. Any website can have a very large (sometimes thousands) of pages, which would put a lot of stress on the server. A user can still target as many pages as he wants from a single website.

The second and third options are implemented, and the fourth is also partly implemented but in the same way as the third. There is no specific option to check for a new HTML-element, but the same result can be achieved by setting the scraper to a specific element, which will then catch the new element as a change.

The user interface works quite well and allows editing the targets, managing devices that receive notifications and managing the notification history. Because we have authentication and it works for multiple users, the application can actually be hosted on a domain.

Reflection

We used a lot of different libraries. According to the build-depends in the cabal file, we import 25 different packages excluding the libraries for testing. Some of these packages are essential, such as servant for the server and sqlite-simple[9] for the database, but some packages only contribute a very small piece which could have been handwritten if needed. All these packages created regular conflicts with dependencies on other libraries, but in the end there is a working configuration. With the enormous amount of packages, including multiple packages of the same library (tagsoup and tagsoup-selection for instance), maybe we could have done with less at the cost of implementing more ourselves.

Our front-end was built using HTML, CSS and JavaScript. These pages were written in these languages, without using any Haskell or other functional language such as Elm. For the content of the assignment, if there was time left we would have considered writing these pages using a code generation package such as GHCJS or Blaze. However, because these pages were of high priority and needed for easy testing, we decided to write them using technologies we already are familiar with.

The current interface to the database does not provide any type safety when adding or modifying data. We could store an integer in a text field or provide more arguments than needed to a query, which then may cause an error to be thrown. To solve this, we could have used the persistent library[11]. This library uses Template Haskell to generate data structures, SQL tables and provides functions to manipulate this data in a type safe way, based on the defined data models.

There are still possibilities for extension. For instance, support for polling websites with authentication (or other headers), allowing the configuration to be read from a YAML/JSON file or giving statistics for website changes could be added.

Conclusion

Overall, we are very happy with the result. The result is a functional application that works well enough that we could use it ourselves to track websites (hosted locally). It was quite an experience writing such an application around Haskell, as we all only used it for courses like the bachelor courses Languages & Compilers and Concepts of Program Design.

Sources

1. CRON-format: http://www.nncron.ru/help/EN/working/cron-format.htm
2. TagSoup: http://hackage.haskell.org/package/tagsoup
3. Push notifications: https://developers.google.com/web/ilt/pwa/introduction-to-push-notifications
4. SQLite: https://www.sqlite.org/index.html
5. Servant: https://www.servant.dev/
6. CRON package: https://hackage.haskell.org/package/cron
7. MVC: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
8. XSRF/CSRF: https://owasp.org/www-community/attacks/csrf
9. Sqlite-simple: https://hackage.haskell.org/package/sqlite-simple
10. Hspec WAI: https://hackage.haskell.org/package/hspec-wai
11. Persistent database package: https://hackage.haskell.org/package/persistent