



university of  
groningen

faculty of mathematics  
and natural sciences

# Prometheus

## Efficiency and Usability in a Personalized Multilingual Feed Manager

Bachelor's thesis

June 2017

Student: L.A.H. van den Brand

Primary supervisor: Dr. M. Lungu

Secondary supervisor: Dr. A. Lazovik

### **Abstract**

To help students of a particular foreign language master their reading skills in a personalized and engaging manner, we propose and then develop a system that allows for selecting from a set of reading sources and listing the articles from these sources with their calculated difficulty level assigned. We gather valuable data and feedback from a set of high-school students learning French, and then conclude with the effectiveness and usability derived from its analysis.

# CONTENTS

---

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Related work</b>	<b>5</b>
2.1	Zeeguu Reader for iOS	5
2.2	Zeeguu Reader for Android	6
2.3	Effects of (Extensive) Reading	8
<b>3</b>	<b>Problem Description</b>	<b>9</b>
<b>4</b>	<b>Realization</b>	<b>11</b>
4.1	Header	11
4.2	Introducing new users	12
4.3	List of potential sources	13
4.4	Source list	14
4.5	Article list	15
4.6	Starred Articles	16
<b>5</b>	<b>Architecture</b>	<b>17</b>
5.1	ZeeguuRequests	21
5.2	FeedSubscriber	22
5.3	LanguageMenu	23
5.4	SubscriptionList	24
5.5	ArticleList	25
5.6	Cache	26
5.7	StarredArticleList	27
5.8	NoFeedTour	27
5.9	UserActivityLogger	28
5.10	Notifier	28
<b>6</b>	<b>Results and Evaluation</b>	<b>30</b>
6.1	User satisfaction	31
6.2	Click activity	32

6.3	Statistical analysis	33	
6.4	Discussion	39	
<b>7</b>	<b>Conclusion</b>		<b>40</b>
<b>8</b>	<b>Future work</b>		<b>41</b>
8.1	Article ordering and filtering	41	
8.2	Feed descriptions	42	
8.3	Additional caching	43	
8.4	Visual hints for article content	43	
<b>A</b>	<b>Workflow</b>		<b>44</b>
<b>B</b>	<b>Platforms and tools</b>		<b>47</b>
<b>C</b>	<b>Watchmen</b>		<b>50</b>
c.1	Problem	50	
c.2	Solution	50	
c.3	Architecture	51	
c.4	Results	51	
c.5	Where to find it	51	
<b>D</b>	<b>How to contribute</b>		<b>52</b>
<b>E</b>	<b>Notable code-snippets</b>		<b>53</b>
<b>F</b>	<b>Example extension</b>		<b>55</b>
	<b>Bibliography</b>		<b>56</b>

# INTRODUCTION

---

In his famous article [8], McCarthy proposes the idea that extensive reading has the potential to improve the mastery of a foreign language, a statement backed by real-world evidence [9][2] as recently as 2016. These days, the primary sources for readable and continuously generated content can be found online, and thus to master a particular language (at least literary) means to traverse a set of foreign websites and read them daily.

The majority of readable material is most often incompatible with the learner's current language skill level, and thus individuals are faced with the difficult task of not only sorting the available content from multiple sources based on their personal interest, but also on the level of difficulty. Moreover, some learners are often trying to maintain a parallel learning process of multiple different foreign languages, which further complicates the seemingly simple task of 'choosing something to read'.

This project (the Universal Multi-language Reader, or UMR) aims to create and evaluate a reading platform that allows users to identify reading sources in their target language(s) that are both to their liking and at the appropriate difficulty level - not too easy and not too hard. The content of these sources, a list of articles, will be available in a single distraction-less environment, where we will give those reading a foreign text the advantage of having translation possibilities at their fingertips for any new words they might encounter.

The latter is an entire research project of itself however, which is documented in the related thesis of D. Chirtoaca: Appollo<sup>1</sup>. In *this* thesis, we will build the heretofore mentioned system, following the research question:

**RQ1. What is the simplest design and minimal set of functionality for an environment that would support a language learning individual in easily finding appropriate material of personal interest and language of choice?**

---

<sup>1</sup> Where Prometheus is the god of forethought, Apollo is the god of analysis.

The secondary research question is more focused on the actual value our system delivers to the end-users:

**RQ2. Would a system like the one we describe in this proposal be helpful for the learners? More specifically, how much do our users value and make use of the ability to personalize the provided content to their specific interests?**

We shall look into related work and will define a proper problem description based on our findings. The realization chapter then discusses how we managed to solve the requirements listed in the problem description, followed by a chapter discussing the actual architectural choices made to implement the solution. We gather results and evaluate user statistics, which shall help us conclude with answers to the questions posed above. Finally, we provide the reader with a set of suggestion from which future work we deem most valuable to the system can be determined.

Throughout the document we refer to the Zeeguu ecosystem as discussed in [7]: "Bootstrapping an Ubiquitous Monitoring Ecosystem for Accelerating Vocabulary Acquisition". The paper presents, among other subjects, a model that can improve language acquisition. Our UMR is hosted under its umbrella of applications available for beta-testers, as part of the bigger solution it is proposing.

## RELATED WORK

---

As for all of research, this thesis can be seen as a continuation of previous work. In this section we will discuss influences, previous work, similar endeavors and how our system should improve on them.

### 2.1 ZEEGUU READER FOR IOS

Zeeguu UMR in part exists due to the need for a cross-platform version of the iOS-bound Zeeguu Reader [10] (and thus make it *Universal*). Among other things, the reader allowed for subscribing to online RSS feeds (as his thesis demonstrates through the included figure of 1) and listing the articles published by those sources along with their difficulty level.

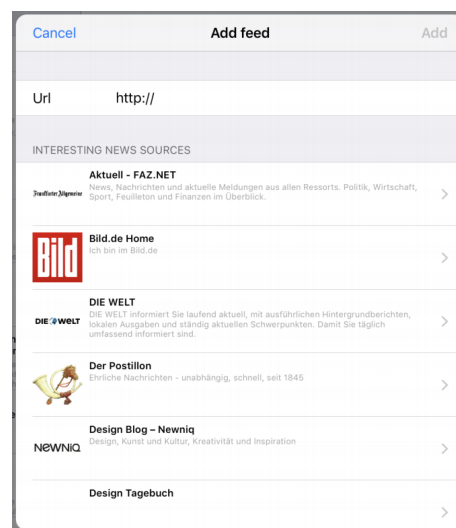


Figure 1: Adding a feed in the iOS Zeeguu Reader

Its Author, Jorrit Oosterhof, created a wonderfully stylized RSS reader application, but improvements could be made:

1. In order to make articles readable, a remote readability service is used. This service shut down after development, making it impossible to read articles using the tool. This implementation should not have been necessary as well: the Zeeguu API uses its own readability library in order to analyze articles, and thus duplicate work is performed on both the server and client.
2. Its potential user-base is small, at the moment of writing the iOS market share is 12.5%<sup>1</sup>. Furthermore, even if the application had been ported to the Android platform (see below), article reading in this environment would be limited to phone-use only. Part of the Zeeguu philosophy is that language learning should be ubiquitous in the learner's environment. Language acquisition should be able to occur whenever and wherever.
3. The *actual* user-base is even smaller, which is to say that during (and after) development the application has not been given the chance to be thoroughly tested in real-life use. The thesis conducts research through a questionnaire pre- and post-development, but the sample size is minimal.

## 2.2 ZEEGUU READER FOR ANDROID

Concurrent to the iOS version, a similar Zeeguu application has been developed by Linus Schwab that allows users to read articles of multiple languages on Android [12]. This reader improves on the availability of news sources by requesting articles and managing feeds through the Feedly<sup>2</sup> API whilst still ordering articles based on their linguistic difficulty. An example as taken from the cited thesis can be viewed in Figure 2 below.

<sup>1</sup> <https://www.idc.com/promo/smartphone-market-share/os>

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



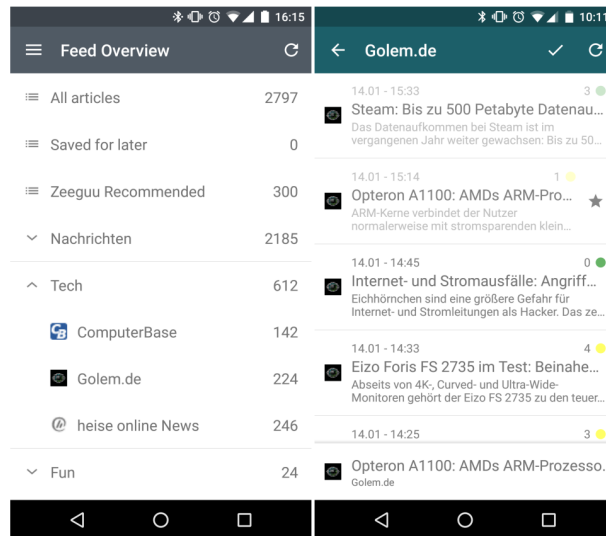


Figure 2: Browsing trough articles in the Android Zeeguu Reader

This service automatically keeps track of which articles the user is reading, which articles the user bookmarked for later reading, and more. Although the inclusion of this service into to application's foundations truly has its benefits (the project will essentially reduce to merging the Zeeguu API and the Feedly API into an efficient android activity), it is not free of risk:

1. The even more heavy link to a remote service implies that the day Feedly decides to pull down their cloud, the Zeeguu Reader of Android will render itself entirely unusable: not only will users not be able to read articles, none of their subscriptions or articles will be listed.
2. Users will need to subscribe to the Feedly service *in addition to* the Zeeguu service.
3. The application has not been evaluated beyond a qualitative case study with the secondary supervisor of the project, and thus it's real-life benefits to actual foreign language acquisition have yet to be proven.

### 2.3 EFFECTS OF (EXTENSIVE) READING

To improve writing style, classical teachers tend to resort to creating writing assignments in assistance with direct instruction. This approach to teaching has however been the target of criticism, as documented by the work of Stephen Krashen in "We learn to write by reading, but writing can make you smarter" [6]. In his article, Krashen performs a thorough literature study and provides strong arguments for improving writing through reading, something that can be beneficial to both beginners and advanced students [5].

The previously mentioned research from 2016 indicating that extensive reading can have the potential to improve language acquisition is called "The Effects of Extensive Reading on Reading Comprehension, Reading Rate, and Vocabulary Acquisition" [9], and provided interesting information such as Figure 3 depicted below.

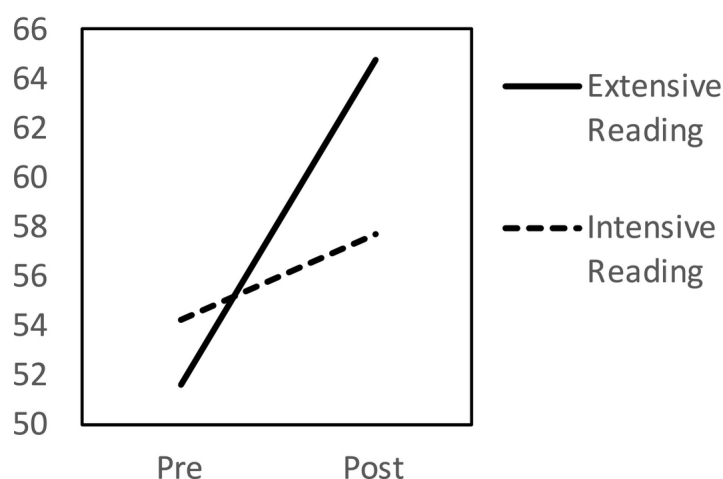


Figure 3: The increase in vocabulary score between two groups of students reading either intensively or extensively.

This Figure shows how students reading extensively have the potential to learn more words than the students reading intensively. The paper also indicates that students reading extensively tend to increase their reading rate to a level higher than other students, and that their final comprehension level of foreign material can be scored higher as well.

Earlier work, such as McCarthy in 1999 [2] or Renandya in 2007 [11], further provide empirical support for extensive reading. Both IRI and GSORT (standardized tests to determine reading comprehension) scores tend to be significantly higher for students reading extensively.

We can derive from these results that a useful reading platform would need to provide a large amount of content for students to learn most effectively, as opposed to the alternative of providing a smaller set of meticulously analyzable content.

## PROBLEM DESCRIPTION

---

The primary goal of the system we will be developing is **enabling and motivating users in finding interesting articles of an appropriate reading level based on their estimated reading experience and preferences**. In order to achieve this goal, our system should:

- Support cross-platform use.
- Be relatively independent of remote services, where crucial libraries and frameworks are either locally hosted or contain some sort of fallback service in case of failure.
- Have a high information to content ratio: the environment should be dedicated to listing the available sources and articles to read from and thus should reduce any distractions diverging attention to somewhere else.
- Have an intuitive user interface. As a difficult to use tool is a distraction in its own right, a minimal amount of instructions should be needed in order to use the system.
- Be maintainable in order to easily adapt to user-feedback and improve user-experience overall. With regards to this, high-quality source code and documentation is key.
- Perform fast enough to be perceived as seamless by the reader. Whenever a delay is irreducible, some visual feedback should be present in order to keep the user engaged.

These requirements are derived from the above mentioned related work, where we are focusing on their shortcomings and drawing inspiration from their achievements. More functionally defined, the features we will develop are:

- Feed subscription system for finding and managing a set of personally useful/interesting feeds.
- Listing the articles of all subscribed feeds.
- Listing all articles bookmarked from within the article reading tool.
- The ability to remove a bookmarked article.
- Visualizing the reading-difficulty of the article based on Zeeguu's analysis.

These features will utilize the Zeeguu API<sup>1</sup>, and thus should give access to all its resources in all its available languages.

---

<sup>1</sup> <https://github.com/mircealungu/Zeeguu-API>

## REALIZATION

---

In order to increase intuitiveness and simplicity, an application needs to have some UI guidelines that follow a consistent philosophy. With regards to this, the Material Design<sup>1</sup> movement initiated by Google fits our particular needs in many ways. Hierarchy, meaning and focus are combined with tactile and intuitive design, where dialog windows and interactive objects are bright but minimal.

Due to these guidelines we are more capable of developing an interface containing a high information to content ratio, but we must remain careful to not clutter our window with redundant information. Below we will discuss each feature of the system, where we will conclude with how they come together as a whole to provide a neat and practical solution to the problem at hand.

### 4.1 HEADER

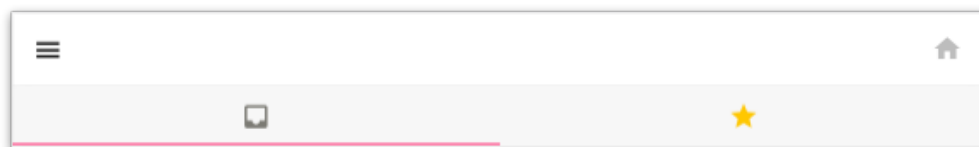


Figure 1: The header fixed to the top of the page.

The top of the page provides two buttons and two tabs, which are the primary controls needed to use the system. Following material design, clicking the top left icon results in a settings menu being opened where less-used interface controls are stored (see 'Source list' below). The top right icon returns the user to the main page of Zeeguu, and thus represents a home.

The tabs below these icons allow the user to switch between two lists. The inbox tab, selected by default, displays all articles available to the user. The

---

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

starred articles tab displays all articles starred by the user. By using tabs, we improve consistency and flexibility: the main section of the page is simply reserved for listing articles, the source of articles (feeds or starred list) can be altered without changing or reloading any of the other components.

## 4.2 INTRODUCING NEW USERS

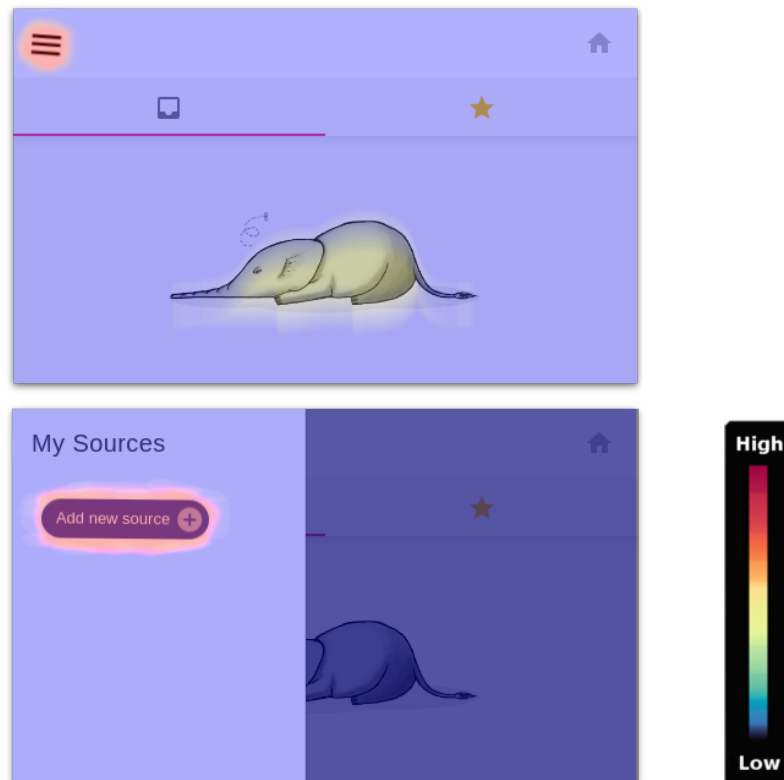


Figure 2: A demonstrative visualization of low to highly animated content on the screen when no feeds are subscribed-to.

In order to non-verbally introduce new users to the system, we decided to let the user be guided by what we will call a *wiggle tour*. A wiggle tour animates the user interface by cyclically tilting those interactive elements we wish the user to explore, and is based on the notion that motion attracts attention.

By wiggling the menu button, users are encouraged to click it. When this is followed by the menu sliding into view they are presented by another wiggling button inviting them to add their first article source. As clicking this button activates the dialog window that allows for adding new sources, they have been guided successfully to their destination.

## 4.3 LIST OF POTENTIAL SOURCES

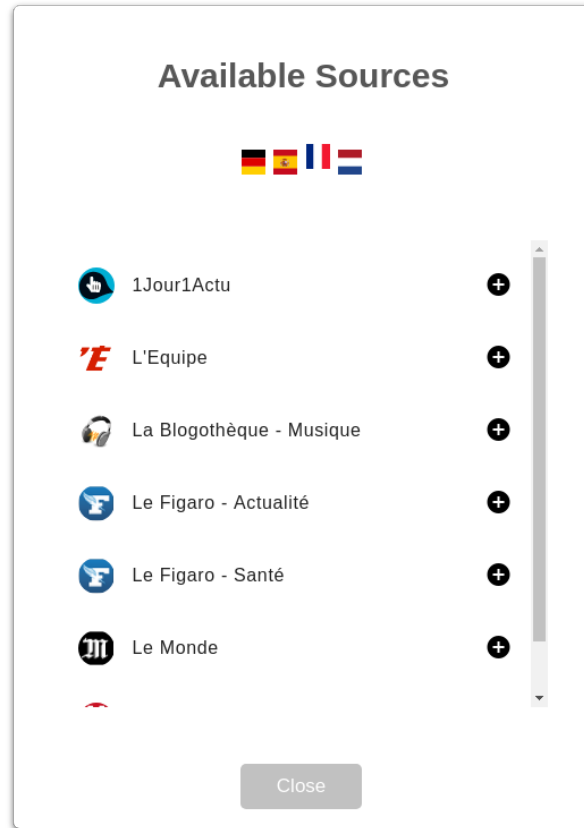


Figure 3: The dialog menu for finding reading sources.

When the user indicates that he would like to subscribe to a new source, as explained above, the subscription dialog is displayed. By having the user find new sources within a dialog window, we attract attention and focus on the task to be completed such that the user need only to pay attention to which sources are available. At the same time leaving this mode and thus hiding the dialog can be achieved through a single click, therefore making it easy to continue browsing articles once the user is finished.

Zeeguu categorizes feeds by their language, and thus we allow users to select any language available and retrieve a list of that language's sources. Languages are represented by flags, as their compact and iconic representation should be universally understood. The language the user is currently learning (as recorded by Zeeguu) is selected by default. This gives the user the largest amount of freedom in browsing the list of available reading sources, whilst still giving them the ease-of-use a more restricted single-language system would have provided.

We assigned a plus-symbol to each feed in the list, continuing an established notion from the source list discussed below. Clicking on this symbol fades-out the feed, where it immediately fades into the list of subscribed feeds.

#### 4.4 SOURCE LIST

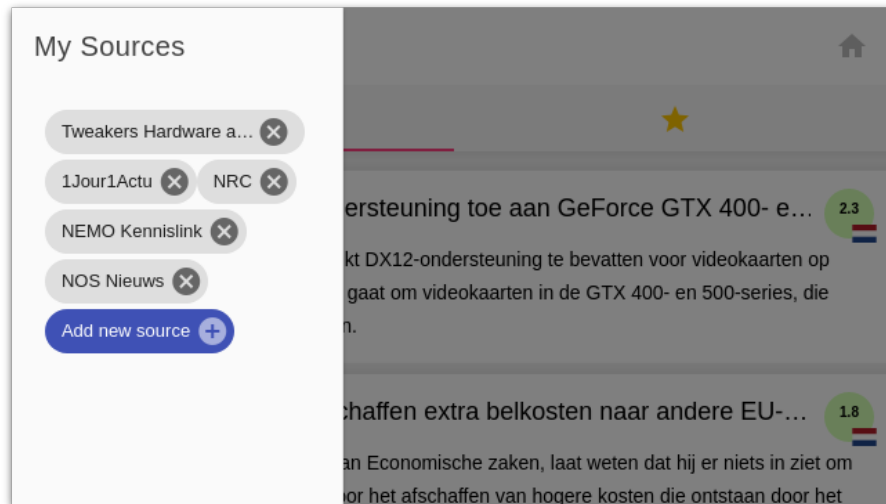


Figure 4: All subscribed sources are listed within a slide-in menu from the left.

When the user has the ability to subscribe to article sources, it also becomes essential that a list of these subscribed sources is represented somewhere in the system. When the user decides he no longer wishes to follow a particular feed, it should be possible to simply refer to this list (see Figure 4) and remove any amount of sources desired.

As the user is not expected to manage their list of sources on a day-to-day basis, we deemed it would only be distracting to include this list in the main visible area of the application. Following the convention of including configuration settings in a slide-in menu, the most intuitive implementation following our style-guide would be to list our sources there.

There is not much real-estate in this menu, especially when considered that a user can theoretically be subscribed to many sources at once and we would have to list all of them. To accommodate for this, we can represent a source using a **chip** component. A chip is a small and interactive element that indicates what complex object it represents and can contain a cross-icon to indicate that this object can be deleted.

Clicking on a cross-icon removes that particular feed from the list. Removed content fades-out in order to make the change in content less of a jarring experience and to reinforce the notion of what is happening on the back-end of the system: content is being removed. At the bottom of the list a special button



is located with a plus-icon, suggesting that this will allow the user to *add* to the list instead of detracting from it.

#### 4.5 ARTICLE LIST

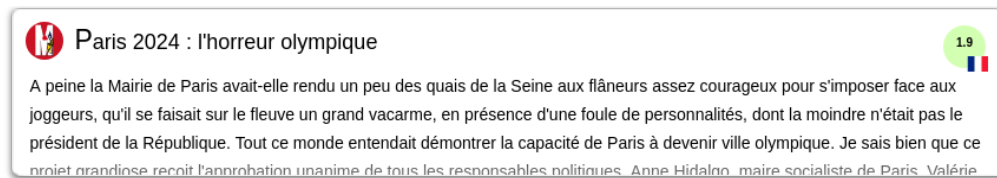


Figure 5: How each article is listed.

Whenever the list of subscribed feeds is updated we update the list of articles the user might be interested in. Each listed article contains two components, where one can be further reduced into four components on their own.

- **Article header.** This provides all primary information about the article, and should be as visible as possible.
  - **Source icon.** We continue to prefer visual representations above textual ones, and as each source has their own logo associated with them, this has been deemed the most appropriate format to represent them in relation to the listed article. This information helps the user judge what kind of article to expect, as each source generally has their own style and approach to producing content.
  - **Article title.** To increase readability each title is left aligned and has a slightly larger opening character such that the beginning is easy to find. Excessively long titles are cut-off by an ellipsis. In order to accommodate their size otherwise we would either have to vary character sizes or height dimensions between article listings, which would have made it more difficult to visually scan through the list.
  - **Article difficulty.** In order to properly visualize the reading difficulty of an article in an intuitive manner, there are two levels of information displayed here. First we allow the user to rapidly judge difficulty on an **intuitive** level by color coding the difficulty from green to yellow to red. When a particular article has grasped the user's attention, we allow for a more **cognitive** judgment by scoring the article from 1 to 10 in difficulty.
  - **Article language.** As decided earlier when developing our suggested feed dialog, multiple languages are supported at once and we represent languages through their respective flag. The language

icon slightly overlays the article difficulty, such that it also supports the notion that those digits represent the difficulty of the *language* written.

- **Article summary.** When the header fails to deliver enough information to the user, a small section of the article listing is reserved to provide a short summary. This summary is left-aligned as well, and fades out when it reaches beyond 3 lines. Our theory is that those whom desire to read more content beyond that are likely to be interested in the article in any case, and we do not want to encourage the reading of articles outside of the therefore designated reading environment.

Hovering the mouse anywhere above a listed article will make it drop more shadow and will change the cursor of the mouse into a hand. This is to suggest interactivity and puts focus on the article you are about to interact with. Clicking on the article will fade-out all others whilst the browser redirects to the article reading environment<sup>2</sup>.

#### 4.6 STARRED ARTICLES



Figure 6: How each starred article is listed.

The user has the ability to store article he or she likes to read at a later moment in time. This functionality is called *starring an article*, and all starred articles are stored on a per-user basis. Our job is to list these articles properly, which we can do following (for consistency) the same design choices as given above. There are a few differences:

- No summary or article difficulty is provided by Zeeguu, as the user has already decided the article is interesting enough to read and the title and language should provide all information needed in order to recognize the article at hand.
- A cross symbol is included to easily allow users to remove content from their starred list once they lose interest in a particular article.

<sup>2</sup> and thus to a (sub-) system outside the scope of this thesis.

## ARCHITECTURE

---

As our application will need to run on web-browsers, we decided to implement the server-side through Python Flask and the client side in ECMAScript 2015 (ES6). Due to ES6 being powerful but poorly supported, we transpile our code into a neatly packaged JavaScript file using Webpack before deployment. For more information regarding Webpack, see Appendix B.

The Flask server is very minimal, as it validates the user and on success redirects them to `articles.html`. This HTML file refers to our styling documents, contains all templates used, and links our packaged script to be executed. An example is given in Figure 1, where the `/article` endpoint redirects to the system developed in [3], and thus can be regarded as a ‘black-box’. Some classes are shared between our work and the work of D. Chrutoaca, which is why they have been kept outside of our Subscription package.

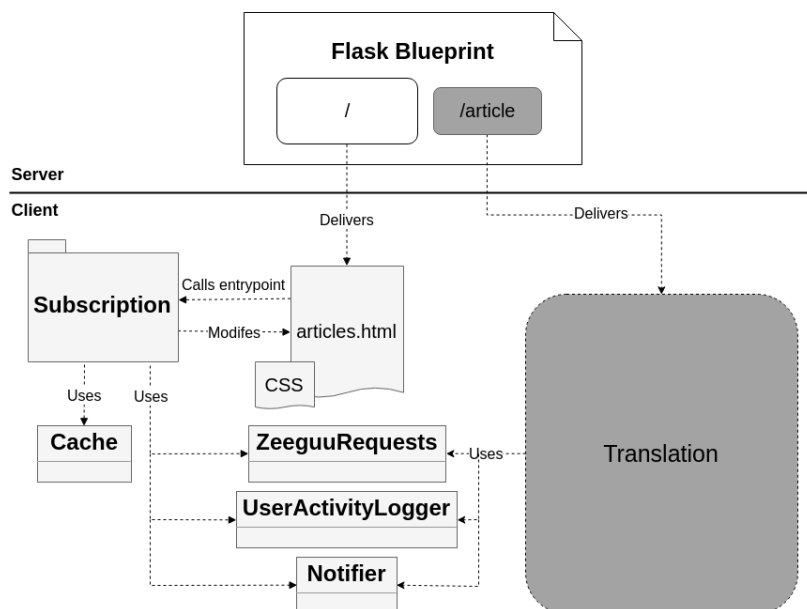


Figure 1: The server delivers, when the root endpoint is accessed, the application by returning an HTML document linking all relevant files.

To prevent browsers from caching outdated iterations, both the JavaScript and CSS files are named using the semantic versioning based schema of `PACKAGE_NAME.MAJOR.MINOR.PATCH.FILE_TYPE`. For example, version 0.8 of the system includes `subscription.0.8.0.js` and `subscription.0.8.0.css`.

We make use of the Material Design Lite framework in order to consistently construct user interfaces that follow the Material Design philosophy. Creating components such as slide-in menus and animating buttons to ripple when touched can simply be achieved through adding the right classes or placing the appropriate tags inside our HTML documents and templates. MDL properly scales to a wide variety of devices and resolutions, improving our cross-platform support significantly.

Before transpilation, our package of ES6 files can be visualized as shown in Figure 2. The `ZeeguuRequests` and `UserActivityLogger` classes are not depicted, due to their static implementation and global use throughout the system.

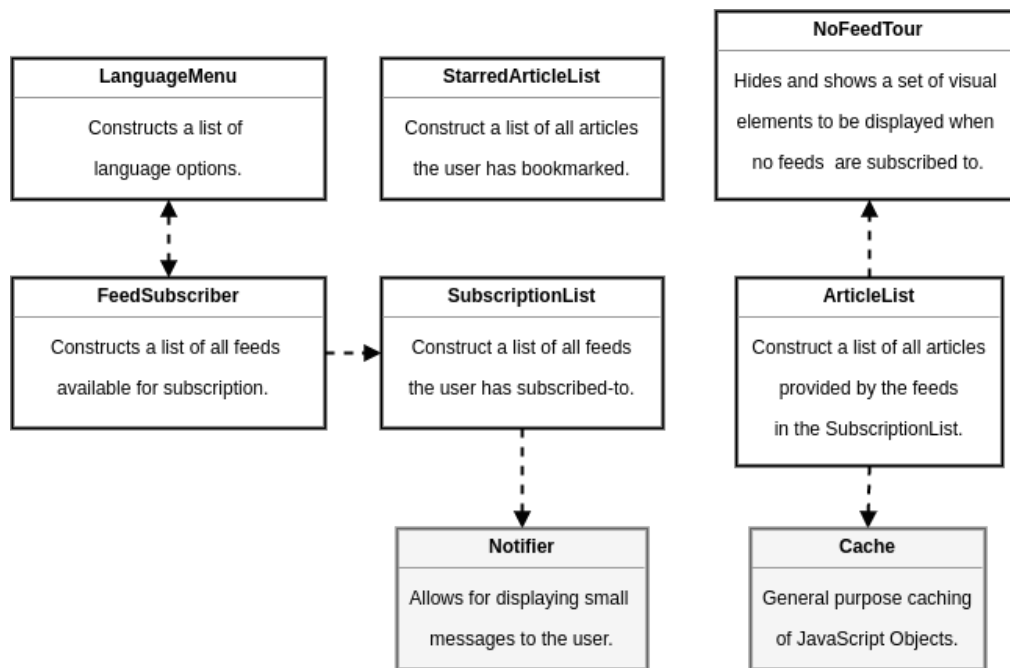


Figure 2: The subscription package.

Following an OOP design, all functionality is divided among a set of sub-problem assigned classes. We will discuss these classes in greater detail below. However, each class follows the same set of design decisions in order to help us meet requirements:

- **Do first, tell later.** Whenever we are able to perform a certain change of state without notifying the server, we should do that concurrently to the request. Waiting for the server to respond will reduce responsiveness on

the user interface, and will become excessively noticeable when many state operations are executed in a short amount of time. In other words: we design the system to assume success in order to speed up the process, and on failure we can deal with the consequences. Similar to a CPU performing calculations before a branch has been chosen, this can increase the number of operations our users can perform within a certain unit of time.

- **The view is a document, the model is a script.** As the famous Model-View-Controller pattern rightfully dictates, the visual aspect of the code should remain separate from the model describing all functionality. Due to the Mustache library, templates for reusable HTML elements can be used in order to generate UI without our code being concerned with its markup.
- **Events.** When classes depend on the state of one-another, there are three possible approaches to keep them synchronized:
  1. Actively polling the instance we depend on. This is the most inefficient manner of synchronizing state, and in general is only used as a textbook example of *what not to do*.
  2. Storing the dependent instance as an attribute of the instance it depends on, and calling its update method once the state changes. Although useful in situations where functionality is divided among multiple classes (see Decorator below), coupling two otherwise unrelated classes can make the code difficult to modify and comprehend. It is not immediately clear why two instances are bound, and a change in interface for the dependant child requires a change in behavior for the observed parent.
  3. Firing an event on change, where dependent classes are able to listen for this event and act accordingly. Although subtle, there is a difference between event-based listeners and the previous approach. With events, there is no direct coupling between instances. Instead, coupling is *implicit*. When a listener decides to change interface, it is also its responsibility to change how it handles the event. Change in code is now focused to one region instead of two. Similarly useful is that many listeners can be bound to the same event without us ever touching the observed instance.

For these reasons we decided to implement dependencies of otherwise unrelated instances<sup>1</sup> using events.

- **Decorator.** Heavily related instances are implemented using the Decorator pattern. To prevent a class from growing into an unmanageable behemoth,

---

<sup>1</sup> See the implementation of SubscriptionList and ArticleList.

it can be divided into a group of smaller more obedient instances sharing a common purpose. Each instance is designed to solve a sub-problem of the larger whole, where their accumulation into a set of attributes of the encapsulating class provides a simple interface to the solution (see Figure 3 as an example).

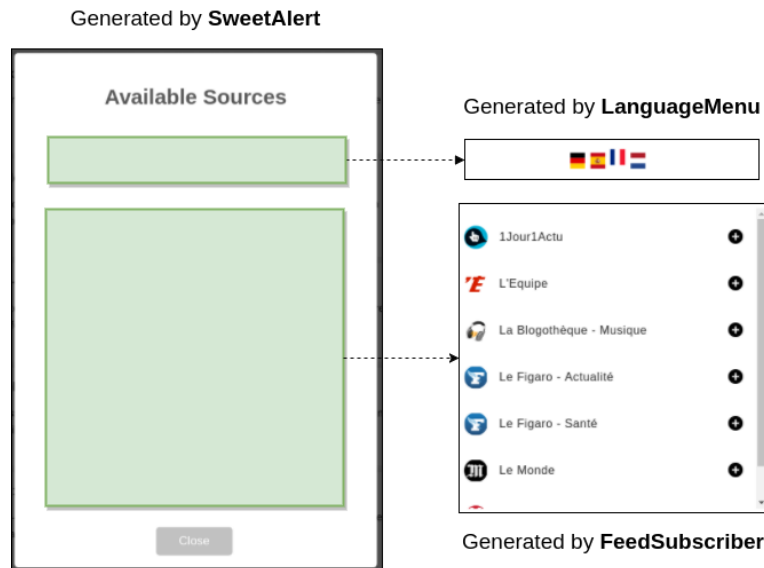


Figure 3: The dialog window is generated by three different instances collaborating to produce an intuitive and responsive interface.

- **Configuration.** Hard-coded constants are bad practice, due to them being error-prone and relatively difficult to maintain. Instead, we introduce constants at the top of classes to clearly define strings or other values used in the code defined underneath them. An exception to this occurs when multiple classes rely on the same constant, and it is not immediately obvious which class should be the one defining it. To prevent confusion and high coupling of classes, only these kind of constants are defined in a global configuration file. As a result, global constant-dependant behavior can be modified by simply changing the configuration, whilst more specific behavior requires more local code modifications.
- **Following conventions and proper naming.** When applicable, consistent naming and layout conventions have been included into the design. The python code (although minimal) follows the PEP 8 Style Guide. All ECMAScript is written following popular conventions, where practices such as underscoring the first character of private methods have been taken into account. Descriptive and clear names are preferred to short but obfuscated ones.

- **Code documentation.** Although some claim perfect code to be self-documenting, we strongly believe in the benefits of in-code documentation. By appending each method and class with a documentation comment, we can use automatic documentation tools to generate a thorough overview of the entire system.

This system is dependant on the Zeeguu API for retrieving feeds, articles, and all associated meta-data about these articles, and thus a large part of the functionality depends on requesting and posting information to and from Zeeguu.

## 5.1 ZEEGUUREQUESTS

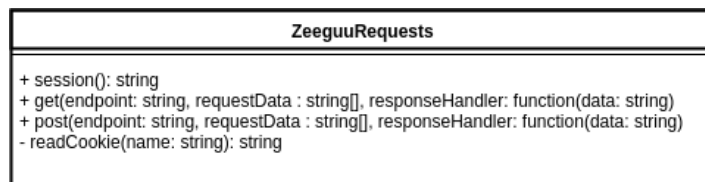


Figure 4: The static ZeeguuRequests class.

Our system interacts with the Zeeguu API by use of a set of REST endpoints that we can contact through either POST- or GET-requests. How requests are performed is abstracted by the ZeeguuRequests class, which provides the methods `get` and `post` that both take three arguments: the endpoint to use, the data to send, and the callback method to be called on reply.

Callback methods are required due to all requests being asynchronous. This is explicitly enforced such that the system remains usable whenever a request is send, which might happen often due to our desire to log user activity. Of course this can introduce race-conditions into our code, where replies are parsed in a different order than how their requests where issued. Any use of this class should take this into account.

Finally, as some endpoints require the user session to be included in the request, ZeeguuRequests automatically includes this session by retrieving it from the cookie set whenever the user logged into their account.

## 5.2 FEEDSUBSCRIBER

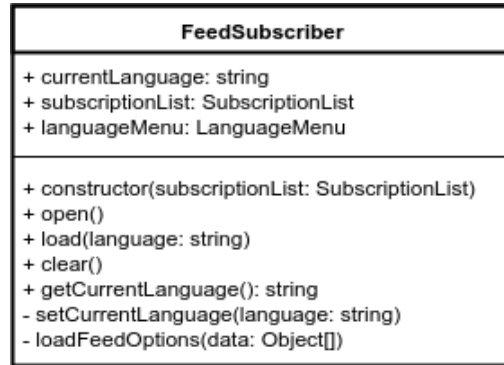


Figure 5: The FeedSubscriber class.

The FeedSubscriber represents the dialog window which allows users to browse the available list of feeds and gives them the ability to subscribe to them. It makes use of the Sweetalert library, as the MDL-framework provides a less intuitive (more basic) dialog window.

On open, we request SweetAlert to generate a dialog with our dialog-template embedded in its content. Then we request the LanguageMenu to load, after which the load method of the FeedSubscriber itself is called (see Listing E.1 in Appendix E).

The load method expects a language parameter in order to request a set of suggested feeds from Zeeguu. If not provided, it will use the previously requested language which it stores internally after each call. Given this language, the /get\_recommended\_feeds endpoint is called using a GET request.

When the callback method is invoked, we can expect a list of feeds that the user can subscribe to. For each feed we generate a new HTML element using a predefined template and the retrieved JSON element describing the feed. This JSON element contains the following attributes:

- **id**. The unique identifier of the feed, which can be used to request a subscription to this feed.
- **title**. The name of the feed.
- **url**. The url at which the feed can be found.
- **language**. The language in which the feed provides content.
- **image\_url**. The url of the feed's representative image.

Which is all the information required in order to properly list the feed.

Two listeners are bound to each feed. A click-event listener is bound to the plus-icon element, such that clicking on it sends a follow request to



the SubscriptionList and fades-out the listed feed in the dialog window. An error-event listener is bound to the feed icon element, such that broken image references can easily be replaced by a specific no-avatar icon.

### 5.3 LANGUAGEMENU

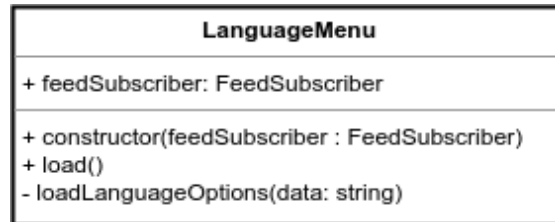


Figure 6: The LanguageMenu class.

As the LanguageMenu is only ever used and constructed by the FeedSubscriber (and thus is strongly tied to it), this class is constructed by passing a FeedSubscriber instance as an argument. This was preferred to passing the same instance to *load*, especially because the specific instance will never change. On calling *load*, a request for the list of languages is sent to the Zeeguu API using the `\get_available_languages` endpoint. The reply will invoke a private method called *loadLanguageOptions*, which the LanguageMenu uses to flexibly generate a list of flags.

A flag button is defined by a Moustache template and uses vector art from the *flag-icon-css* library. The language code assigned to the button determines the flag that is used to represent it. When generating a flag button, it is appended to the dialog window's language option list, and a click-event listener is bound to clear and load content within the passed FeedSubscriber whenever one of the flags has been clicked (explaining the two-way relation given in Figure 2).

This architecture makes supporting more (or less) languages possible without requiring changes to the actual code-base.

## 5.4 SUBSCRIPTIONLIST

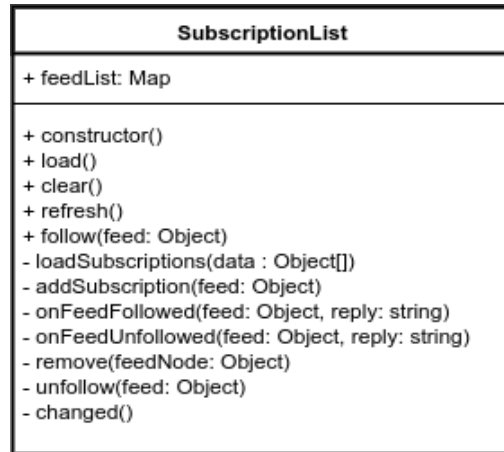


Figure 7: The SubscriptionList class.

Before we are able to retrieve articles, we first need to acquire a list describing each feed the user is subscribed to. The SubscriptionList maintains a set of these sources, such that each feed can only be listed once. This was done to introduce idempotent operations to the class, where adding or removing a particular feed can be performed multiple times in succession whilst producing the same result.

On load, the class will request `/get_feeds_being_followed` to return the currently followed feeds. The callback method is returned a list of JSON elements similar to the structure as discussed for the FeedSubscriber, and will combine that with our template to generate a list of removable MDL chips. A listener is bound to the cancel button of these chips, which will invoke the `unfollow` method. A `changed`-event is fired, such that all listeners are aware of the new state of subscriptions.

In line with keeping the application responsive, the `unfollow` method immediately removes the feed. Simultaneously, Zeeguu will be requested to remove the feed on its server, such that client and server are in sync when the page reloads. When Zeeguu replies with a success state, we again fire a `changed`-event. On failure, we notify the user using the `Notifier`.

As discussed above, the SubscriptionList is meant to handle a `follow` request from the FeedSubscriber. This immediately calls the callback method used to add feeds, such that a subscription seems to be included in the list immediately when the user requests it: again promoting responsiveness. Concurrently a request for the subscription is sent to Zeeguu, and its reply shall trigger a `changed`-event on success. On failure, the user is again notified.

## 5.5 ARTICLELIST

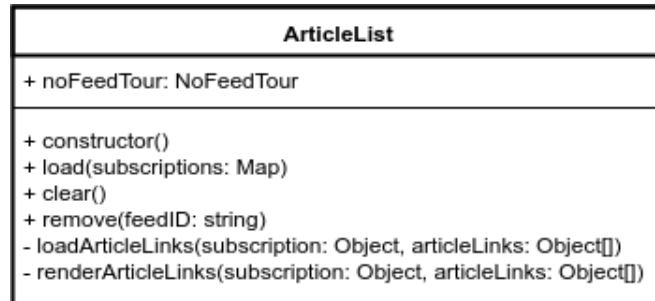


Figure 8: The ArticleList class.

The change-event fired by the SubscriptionList (see Listing E.2 in Appendix E) triggers the ArticleList, which will use the event's appended data to retrieve articles for each feed listed within it. In order to reduce redundant requests for articles retrieved previously within the same session, we make use of the Cache class which allows us to cache a hash-map linking feeds to a list of articles. When a feed has been retrieved before and there was enough space to store its articles (its key is present within the map and the map was smaller than the space allocated in Cache), we simply pass the hashed list to our private renderArticleLinks method such that they will be displayed in the document again.

Whenever our event lists a feed not listed within our cached hash-map, we are forced to request a list of articles from /get\_feed\_items\_with\_metrics. This endpoint will return a list of JSON elements describing each article for the given feed as follows:

- **title.**
- **url.**
- **content.**
- **published.**

This list is included in the hash-map, which is then renewed in cache, and then is passed to the same renderArticleLinks method to be displayed.

The renderArticleLinks method uses a template that defines the markup of an individual article link and the passed JSON to generate a document element describing it. Each element is appended to the list of article links, and a click-event listener is bound to animate a page-transition whenever the link is clicked.

## 5.6 CACHE

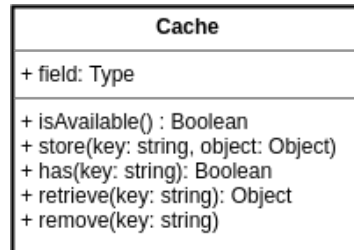


Figure 9: The Cache class.

Requests for articles are expensive, as it might take several seconds before Zeeguu responds. Users are expected to frequently leave and enter the system within the same session due to the application redirecting to readable articles. Requesting the same list of articles every time the user returns to our system is not only a waste of requests, it will severely test our user's patience by asking them to repeatedly wait for multiple seconds in order to continue normal usage.

As a solution, the retrieved articles need to be cached. Modern browsers support a feature called *local storage* where (in contrast to cookies) large amounts of data can be stored securely on the client device. The local storage API provides an object called `sessionStorage` that retains data until the tab containing our application is closed. Temporary storage is preferred due to the fact that the list of articles changes over time, and thus permanent caching will yield too much outdated content.

The Cache class provides a more convenient manner of caching information. Where `sessionStorage` only allows for key-string storage, our Cache class wraps around this instance and provides key-object storage. By use of the `store` method, an object is converted to its JSON string representation, and then stored in `sessionStorage`. Subsequently using `retrieve` with this key shall yield this string and convert it back to an Object instance before returning it to the caller. Only property information is preserved in this process, thus proper casting might be required afterwards.

Local storage is not always enabled on a browser, either due to it not being supported or due to a user actively disabling it. To prevent from cluttering the code with checks for its presence, the wrapper will simply not store information when support is absent. By calling the `has` method to check for an object's existence or `isAvailable` to explicitly verify support, proper behavior can easily be enforced.

## 5.7 STARREDARTICLELIST



Figure 10: The StarredArticleList class.

When a user saves an article for later reading, it is stored on a remote list at Zeeguu. Gaining access to these articles is a simple operation in respect to gaining access to the general list on a per-feed basis: we simply pass the user-session and are almost instantly given the JSON representation of starred articles in return.

The lightweight aspect of this request does not require us to implement caching. Not caching this list is beneficial, due to the user expecting this list to be up-to-date with their latest actions, where they are less aware of the activity generating the cached `ArticleList`'s content. Only updating the list after their session could cause confusion and frustration.

The information passed about each starred article is more elementary:

- **title.** The title of the starred article.
- **language.** The language the starred article is written in.
- **url.** The URL pointing to the content of the article.

Using this data and a template similar to the default article link's template, each starred article is generated as an element in the document's list of starred articles. The template also contains a clickable cross-symbol, to which we tie a click-event listener. This listener, when triggered, will remove the element from the list and sends an unstar-request to Zeeguu. All this actively reflects our previously discussed design choices.

## 5.8 NOFEEDTOUR

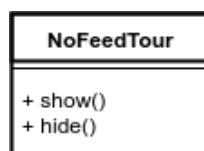


Figure 11: The NoFeedTour class.

New users are introduced to the system using the previously discussed 'wiggle tour'. The NoFeedTour class can collectively toggle all elements of the page dedicated to this tour, such that other instances need not to be concerned with the layout of the document.

## 5.9 USERACTIVITYLOGGER

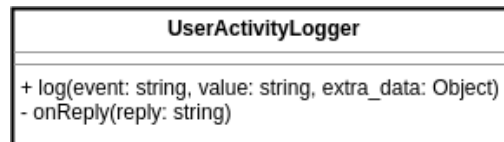


Figure 12: The UserActivityLogger class.

Recording user activity in order to improve the system and perform proper analysis is (amongst others) implemented by sending logs to a remote endpoint. Logging should be straightforward and minimal, in order to decrease the amount of code cluttered by it. Therefore The UserActivityLogger further simplifies the endpoint call provided by ZeeguuRequests and defined a method `log`.

The `log` method requires a string argument that defines the type of event being sent. In addition the caller can pass an associated value or even an Object to provide further detail.

## 5.10 NOTIFIER

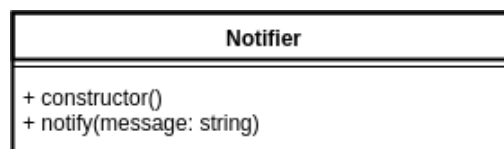


Figure 13: The Notifier class.

Notifying the user of short but relevant pieces of information is achieved by use of the MDL Snackbar component. This Snackbar can be requested to show messages through the `showSnackbar` method provided by the MDL framework. However, repeated messages are repeatedly shown by this method, even if the message is currently being displayed. For example: 10 "failed to connect" messages would be shown 10 times, one after another, for one second each.

This is not desired behavior, and thus we created the Notifier class to wrap around this method. This class remembers the previous message and whether or not it is currently being displayed. If we pass a message to `notify` that is currently in view, this class will ignore the request.

Detailed information about each class, down to the actual lines of code, can be found hosted on the easily accessible GitHub pages branch<sup>2</sup>.

---

<sup>2</sup><https://mircealungu.github.io/Unified-Multilanguage-Reader/>

## RESULTS AND EVALUATION

---

In order to evaluate whether or not our solution ensures all non-functional requirements (testing **RQ1**) and to see if it increases engagement with respect to traditional textbook material (testing **RQ2**), we initially deployed our system in an high-school environment<sup>1</sup> where 98 french-learning students were willing to be beta-testers. Additional beta-testers were gathered by the students themselves, as they advised the use of our systems to their friends and relatives. Which languages were actually studied is discussed in section 6.3.



Figure 1: Introducing the entire Zeeguu platform to a group of testers. At the moment of taking the picture, the exercise platform by M. Avagyan [1] was being demonstrated with words gathered by the reading tool.

Information has been gathered in multiple ways. We use a service called Hotjar to visually monitor activities such as clicks and mouse movement, and to send our user direct questions from within the same environment as the platform (a simple dialog window will pop-up from the bottom of the page to users whom have not yet answered the question it asks). Secondly, we log specific user interaction by sending these events to a dedicated Zeeguu endpoint (as discussed in section 5.9). All users logged to have used our system at least once have been send a form asking about their personal experience with the system. Finally, Zeeguu also maintains a valuable database describing all

---

<sup>1</sup> The Gomarius College, Groningen.



data relations the user might have created when using system, such as which feeds they are subscribed-to.

## 6.1 USER SATISFACTION

When our users were asked:

*"If you wanted to read something in the language you study, what would you reach out for first?"*

58% of those responding preferred to use our system as apposed to the 16.7% that prefer to read from a Textbook. The remaining respondents provided us with replies such as *"Internet"*, or *"Google"*, and thus seem to prefer searching content solely on their own.

Of course, certain beneficial changes can always be made, as indicated by the responses received when we asked for improvements. The primary conclusions that could be derived from these responses were:

- Some users desire to **order articles** differently than the way they are presented. For example, many requested a feature to order articles by topic.
- Many users would like to be able to **filter articles** based on content. A search feature or similar query tool is often given as an example.
- Some users wanted the ability to **add other sources** than the default list of sources presented to them.
- Some users are not always clear on which kind of content is provided by which feeds, leaving them incapable of properly choosing the correct feeds at first use. Some ask for **feed descriptions** or tags.
- Some users want **images** describing the content of an article.
- **Internet Explorer does not work** with the list of articles: an empty white page is shown.

In general, users are not explicitly disregarding the current way content is shown to them<sup>2</sup>, but they do desire to have alternatives and additional tools to further filter content to truly fit their personal interests. Furthermore, some still find it difficult to quickly understand the topic of foreign articles or even entire feeds.

A large part of this feedback had been used to determine potential future work still to be done for the system, as can be read in chapter 8.

---

<sup>2</sup> In fact, users have rated both usability and ease of use a solid 3.9 out of 5.

## 6.2 CLICK ACTIVITY

We recorded click activity on our domain to get an indication of how the tools of the system were being used. Figure 2 is a chart of accumulated user clicks that layers 1206 user clicks from 609 individual page views on top of themselves as a heat-map.

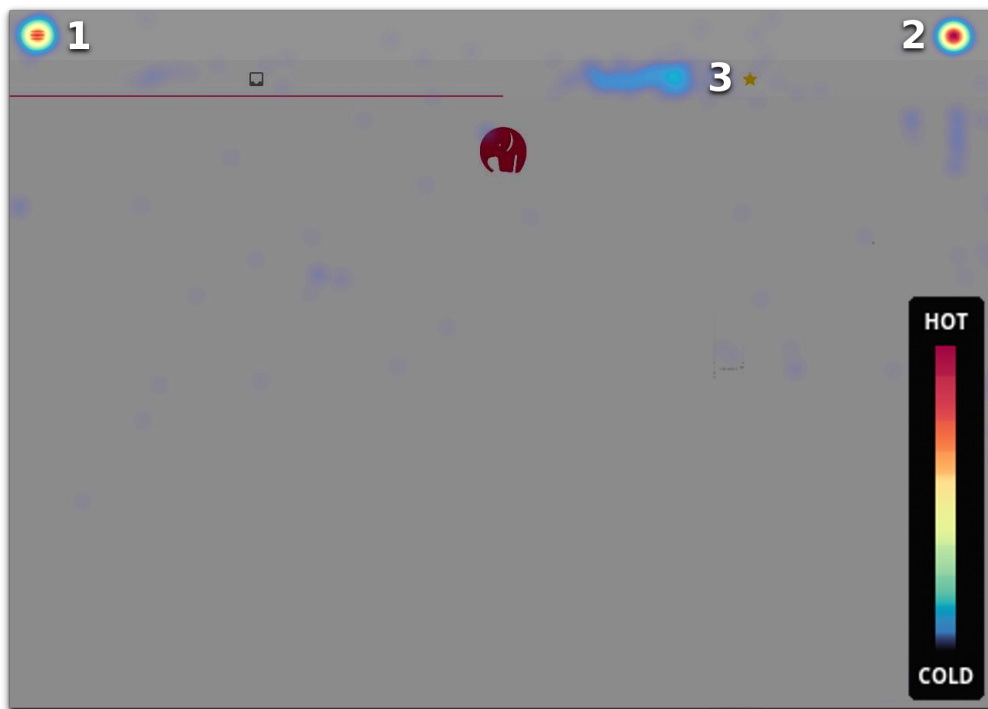


Figure 2: Recorded click activity.

In the heat-map we can identify 3 main hot-spots.

1. **The menu button** is clicked often, which is as we would expect when subscribing to a feed is required in order to use the system. It indicates that most of our users are finding the button successfully.
2. **The home button** is clicked often as well. The home button was (similarly to many components) added based on user-feedback, thus its heavy usage showcases the value user-feedback can give.
3. **The starred-articles tab** is clicked less often, and clicks are distributed to the left. This left aligned distribution can be explained with the fact that the tab decreased in size for a later version of the system.

This information is enforcing the validity of our user interface design, as it can suggest that these three core features have not been left unnoticed by our users.

The clustered nature of these clicks indicate that most of our users understand clearly *where* they need to click in order to interact with the components.

### 6.3 STATISTICAL ANALYSIS

Although our group of beta testers were supposed to improve their French, our system does not restrict the user to learning solely one language at a time. To objectively see which languages the users of our platform are learning, we can cross-reference the list of active users (those users whom have UMR user events logged), with their list of subscriptions, and compute a histogram of languages from the result.

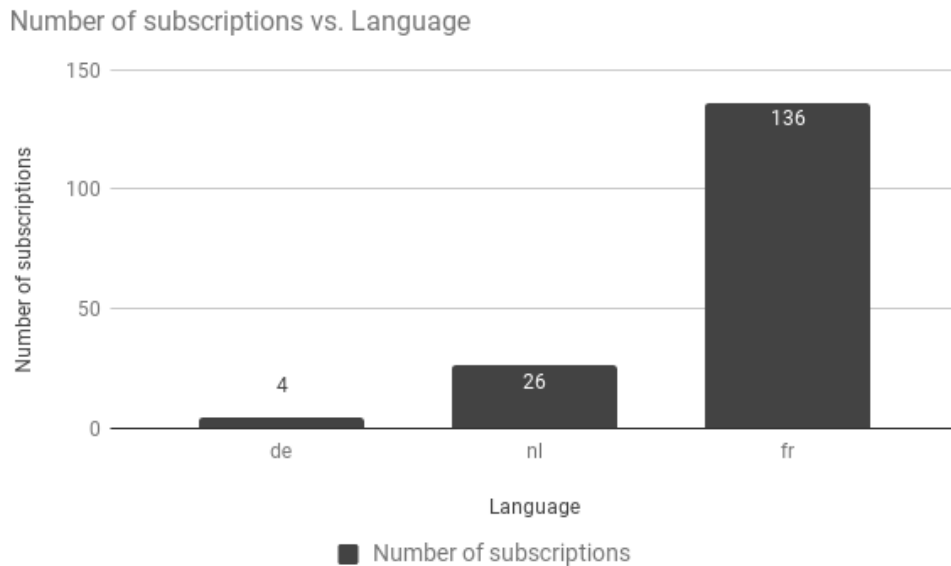


Figure 3: Frequency of read languages.

As we can see from this plot, French is still the primary subscribed-to language on our system, which we would expect when the majority of our sample set is studying this subject.

From the French reading sources, the subscription pattern of each user is visualized in the scatter-plot below. The horizontal axis represents our users, whom are subscribed to one or more of the feeds represented by the vertical axis. We chose to only sample from French sources, as this allows us to focus on subscription behavior without the need to take into account that different users might subscribe to feeds of different languages. In fact, on average our sample of individuals subscribe to only one language at a time, where only 4.3% are subscribed to other languages than French.

Active user subscription patterns of French sources

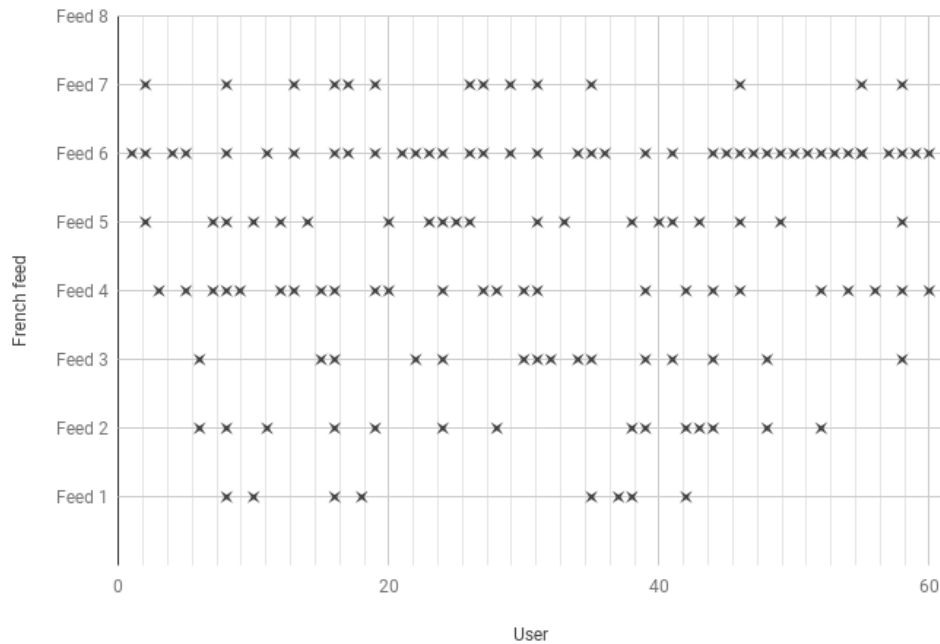


Figure 4: Scatter-plot of subscribed French reading sources.

We would expect to see fully continuous horizontal rows of data-points if every user subscribed to the same feed, and fully continuous vertical rows if every user subscribed to all of the feeds available. The notion that these patterns are largely absent in Figure 4 supports our assumption that different individuals prefer to subscribe to different reading sources.

Of course some feeds are more popular than others. Projecting the data-points into the vertical axis results in the histogram of Figure 5.

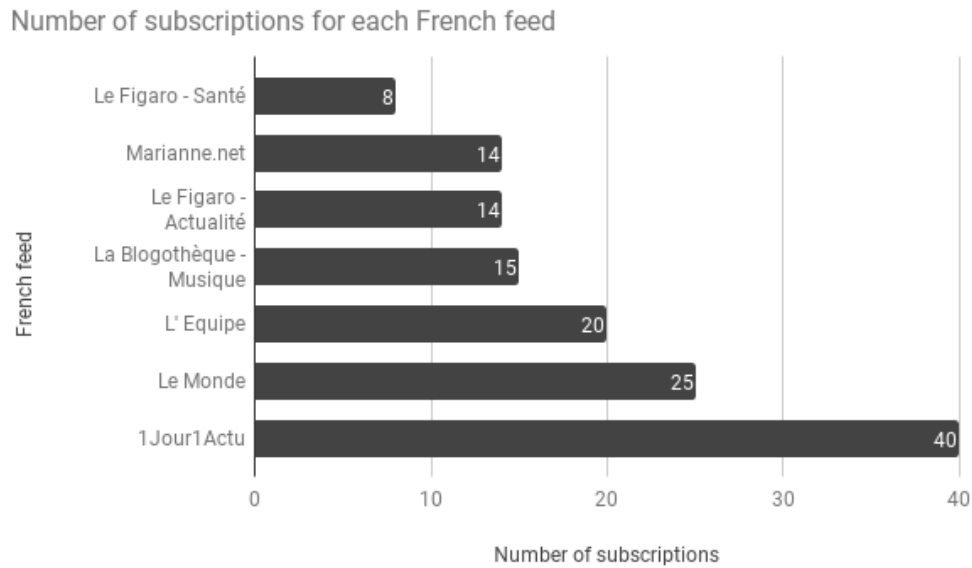


Figure 5: Frequency of subscriptions for each French feed.

Feed *1Jour1Actu* is the most popular French feed, and feed *Le Figaro - Santé* is the least popular French feed. In order to see whether or not this might be related to how they are presented in the dialog window of our system, we can compare the order of popularity with the order in which they are displayed.








Popularity	As listed to the user		
1		1Jour1Actu	+
3		L'Equipe	+
4		La Blogothèque - Musique	+
5		Le Figaro - Actualité	+
7		Le Figaro - Santé	+
2		Le Monde	+
6		Marianne.net	+

Figure 6: Popularity of feeds and how they are ordered in the menu, where 1 is most popular and 7 is least popular.

As we can see in Figure 6, feeds listed at the bottom are not necessarily less popular. *Le Monde* is the second most subscribed feed, although it is the second to last listed in the dialog window.

The content read by our users might differ, but their estimated difficulty is roughly the same when averaging the results of Figure 7.

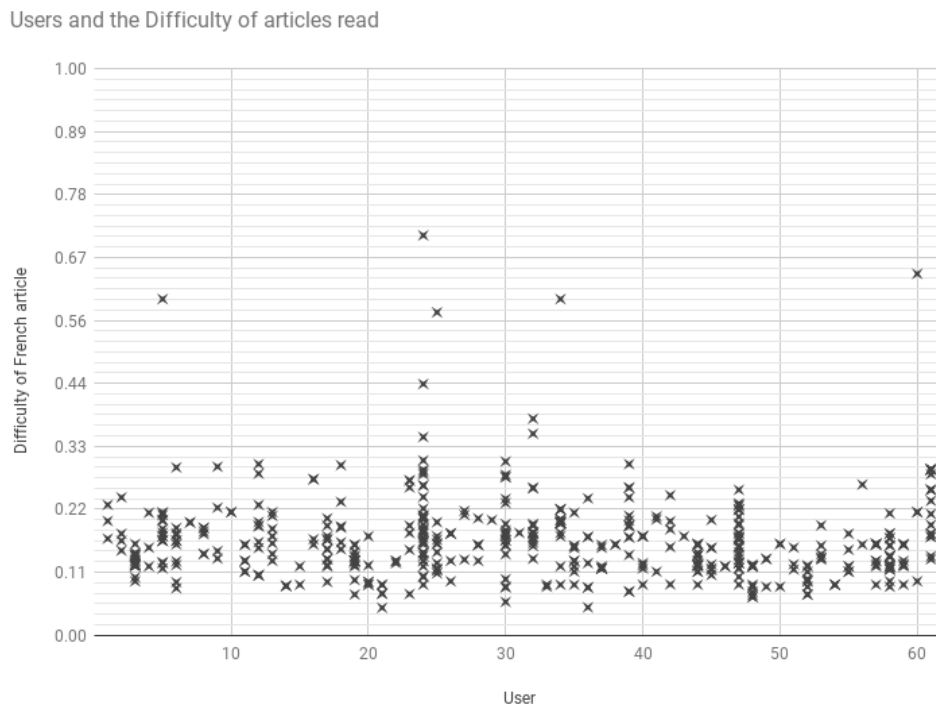


Figure 7: Scatter-plot of users reading articles of a particular calculated difficulty in relation to the user accessing it.

The average normalized difficulty of 0.2 is relatively easy and notably constant, which might have two causes:

1. Our users broadly share the same level of reading skill and strongly prefer to read 'easy' articles to 'intermediate' or 'hard' content.
2. The difficulty estimator lacks user data to properly predict a difficulty score and thus resorts to a base value for this relatively recent group of beta-testers.

Due to new users (with no recorded reading data) being presented with similarly scored articles, the latter explanation is more likely to be the cause of these results. Further research could prove the former to be true as well, as some arguments can still be made about how students might prefer to practice

with lighter material. Moreover, most of our French-learning students on which these results are based share the same educational level.

Finally, knowing the environment in which the articles provided by these feeds have been read is very important to us as well, since it will allow us to test the effectiveness of our efforts to support multiple platforms. We recorded the working environment of about 200 randomly selected interactions with the system, from which we shall discuss the Browser and OS distribution below.

Percentage of visiting browsers

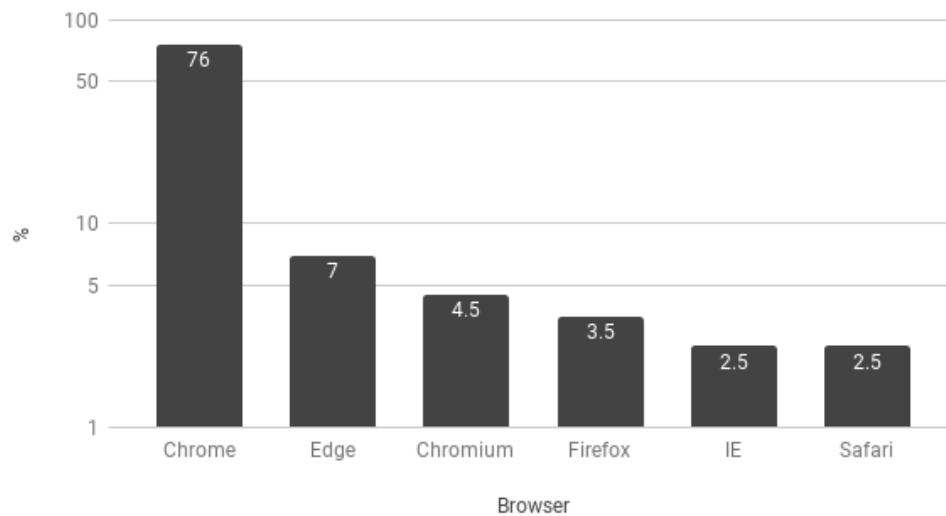


Figure 8: Distribution of browsers accessing our domain.

The most important piece of information that can be gathered here is the relatively low amount of users accessing our domain using Internet Explorer (2.5%). Internet explorer does not support many of the features used by our system, and our choice not to support it seems to effect only a small amount of users. We do support all other browsers listed in this graph, giving us a support coverage of 97.5%.

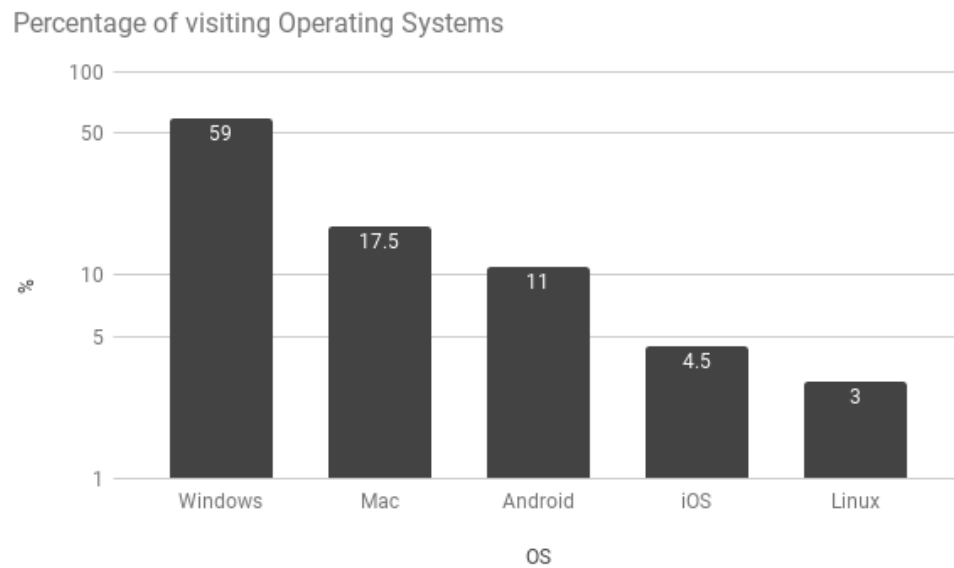


Figure 9: Distribution of operating systems accessing our domain.

The most popular system accessing our domain is Windows, followed by Mac. In total, 15.5% of the operating systems are mobile. In other words, the deployment of our single code-base is being used successfully on different devices and on different platforms, but the majority is using a desktop device.



## 6.4 DISCUSSION

Based on the results and intermediary conclusions found above, we can begin to find answers to some of the questions we posed earlier.

### **RQ1. What is the simplest design and minimal set of functionality for an environment that would support a language learning individual in easily finding appropriate material of personal interest and language of choice?**

Our heat-map shows that users are decisive when interacting with the system, showing that our design choices have intuitive qualities. None of the implemented tools have been left unused, indicating that our choice to continuously adapt the system to the feedback of real-time beta-testers allowed us to focus on the core set of tools users desire to use. When asking for feedback, no indication has been given that the interface is distracting. Following Material Design has clearly kept the interface simple and function-oriented.

Support is lacking for Internet Explorer, but all other browsers seem to work flawlessly and are used by at least some fraction of the sampled set of users. The same can be said for operating system support. This broad range of support is covered by a single set of sources, without any need for platform specific code.

### **RQ2. Would a system like the one we describe in this proposal be helpful for the learners? More specifically, how much do our users value and make use of the ability to personalize the provided content to their specific interests?**

Students indicated that they prefer to use our system with respect to traditional textbook materials. Analysis of the user's subscription patterns show how different individuals subscribe to different feeds. Users do not simply subscribe to the first few feeds listed to them in the dialog window, but actively decide to subscribe to a specific selection. This further proves the advantage our system can have over the static content given by less dynamic content providers.

## CONCLUSION

---

All functional requirements have been implemented, and the evaluation indicates that most (if not all) non-functional requirements are being met as well. We can conclude that the French learning students are actively finding personalized content, and that the interface is clear and intuitive to most of our recorded visitors. This gives a proper foundation to the positive feedback users have given us, as this system has become their preferred tool to find readable material in their language of choice.

Due to the simplicity and flexibility of HTML and CSS, we were able to go through many iterations of the user interface with changing little to no code (see Appendix F as an example). The benefits of developing reusable and maintainable code have become apparent multiple times throughout development, where changes in functionality rarely required large architectural changes. More often than not, a simple redefinition of methods or inclusions of additional listeners, constants or libraries allowed us to implement the additional requirements that any project develops over its lifetime. Our workflow (see Appendix A) mandated us to work in iterations, each iteration reevaluating our priorities and keeping the project focused on delivering a valuable and usable project to our end-users.

The code has been thoroughly documented, and is currently being hosted alongside the remote repository with a calculated documentation coverage of 90%. The only remote dependency our architecture relies on is the Zeeguu API itself, with which we interact only using asynchronous requests in order to make the reader feel as seamless as possible.

Negative feedback has been received as well, primarily focusing on the lack of tools allowing the user to order articles differently than currently implemented or filtering the list of available feeds to only list topics of their personal interest. These comments, and more, will be discussed in the following chapter.

In general, users are very positive when talking about the subscription platform. The high amount of satisfied users confirms most of the assumptions our system has been based on, motivating us with the reassuring knowledge that we have a strong foundation that easily allows for further improvements.

## FUTURE WORK

Based on our user-feedback, and some of our own insights, we shall suggest some improvements that would benefit the usability and durability of our system as a whole.

As our solution is open-source and follows the MIT<sup>1</sup> licence, those reading this thesis are free and invited to implement some of these discussed sections themselves. For more information regarding contributing to this project, see Appendix D.

### 8.1 ARTICLE ORDERING AND FILTERING



Figure 1: Articles are grouped together.

As can be seen in Figure 1, articles from different sources are grouped together. Although this allows for scanning over similar content more easily, it might be more natural to sort these articles in a different manner. Some options are:

<sup>1</sup> <https://opensource.org/licenses/MIT>

- Sorting **by date**, such that new content is easier to find.
- Sorting **by difficulty**, such that we can motivate users to read at a particular linguistic level.
- Sorting **by interest**, such that interesting articles (based on personal behavior, multiple-user behavior, or categories) are shown first.
- Sorting **by language**, such that it becomes easier to only search for articles in a particular language.

Each of these options has potential to be valuable to the user, including the method already implemented in the system. We would suggest to research the value of each option through one of two different means: Either through A/B testing, or (with a smaller user-base) by implementing a feature that allows users to choose a sorting method themselves. Both will allow for collecting usage statistics to decide on the effectiveness of each option, but the latter would introduce another potentially cluttering component to the system.

If those results indicate that sorting by source is used infrequently, an option would be to implement this feature in the more-hidden source list: clicking on a chip representing a source would cause the article list to only show articles related to it. This implementation allows more advanced users to select a particular source and *then* sort its articles, without introducing new components or confusing newcomers.

## 8.2 FEED DESCRIPTIONS

Some new users would not know what to subscribe-to when presented with a list of available sources. A user is assumed to know which of the feeds provide interesting content for them personally, and thus only icons and titles have been provided. To improve on this, we suggest two possible extensions to the current system:

1. By providing **feed descriptions**, such that the content of each feed becomes clear to newcomers. This content could be automatically generated by listing the most popular key-words found within its articles, or manually entered into the Zeeguu database.
2. By implementing a **category system**, such that users no longer subscribe to particular feeds, but to categories encapsulating a set of feeds related to them.

Although option 1 might provide more freedom and is easier to implement into the current platform's environment, option 2 is more likely to fit user behavior: when asked for feed suggestions, users would reply with keywords such as "games" and "horses", instead of listing concrete content providers.

### 8.3 ADDITIONAL CACHING

The weakest link in our application, before caching was introduced, used to be the delay in responsiveness occurring when articles were retrieved. Although our users have not indicated any remarkable points of improvement, some additional features can be cached to future-proof the system for heavy usage. The most prominent are listed below.

- Caching **feed retrieval** does not only increase loading speeds of the feed list, but also has the potential to decrease loading delay in the article list. As has been discussed in the architecture chapter, the article list can only update when the feed list is updated.
- Caching **retrieval of available feeds**, as currently the system will request the list of available options every time the dialog window is opened or a language is switched within this window.
- Smart caching of **starred articles**. Articles are starred outside of our system's environment, and thus changes in state (and thus information for whether or not the cache is still valid) should be requested from a remote service. This request should be significantly faster than the simple request for starred articles in order to make caching valuable.

### 8.4 VISUAL HINTS FOR ARTICLE CONTENT

Visual information about an article can provide further hints as to the contents of the text. Proper imagery has a higher potential to convey meaning when scanning through the list (when compared to foreign words which take longer to parse).

To incorporate these images into the design of the listed articles, one should be careful as to keep consistent with our original design goals: an intuitive and distraction-less environment.



Figure 2: One possible way to include visual content in the listed article.



## WORKFLOW

---

A general consensus is that any good engineer thinks before acting, be it sketching the blueprints for a house or defining the components of a system. Still, one must be careful not to think too deeply about a future he or she cannot possibly predict<sup>1</sup>, as this individual becomes prey to immobilizing guidelines and requirements predefined in a past that has become irrelevant to the present.

This does not imply that we should think *less*, it implies that we should think *more often*: a good software engineer strives for continuous improvement throughout the entirety of his work, not just at the start of a project definition.

This vision has become well spread in modern Software Engineering practices, partly due to the work of famous Software Engineer and author Jeff Sutherland. His book about Scrum [13] convincingly defines a manner of iteratively approaching projects that can be summarized as follows:

- The **Product Owner** understands the vision of the project, this individual keeps the project on track by evaluating the value of features and thus determining which features are important to implement. These features are listed in the **Product Backlog**, which lists all that needs to be implemented in a high-level perspective.
- The **team** is the group developers building the system. They should be cross-functional and small: three to eight members.
- A **Scrum Master** helps the team eliminate waste: obstacles that are blocking their development process. In addition, they make sure members continue to work in accordance to the Scrum framework.
- Development is done in **iterations**. An iteration is also called a **sprint** and should be short in duration (one to three weeks). Each sprint, the Product Owner, Scrum Master and the team gather and define the sprint's backlog.

The product backlog should be **refined and estimated**, and a Definition Of Done should be defined for each feature to be implemented. Estimation

---

<sup>1</sup> A good way of approaching life in general

is best done through ‘Planning Poker’, where a particular task is given a certain amount of points by each of the team members and the average is used to predict its complexity.

- Work should be **visible**. This can be achieved by measuring the amount of points completed each iteration (**velocity**) and managing a **Scrum Board** where team members assign themselves tasks and keep the status of their progress up-to-date. A scrum board is split into three columns: **To Do, Doing** and **Done**, where tasks move from the To Do column to the Done column as the sprint progresses. This increases communication and allows for obstacles in development to be spotted early-on.
- In a **Daily Stand-Up**, each team member answers three questions: what did you do to help the team this iteration, what will you do for them today, and is there anything blocking the team in finishing this iteration successfully?

The meeting should take no longer than 15 minutes, and is meant to force all team members to stay up-to-date about each individual’s progress.

- After each sprint, a **Sprint Demo** should be held, where the current version of the system is demonstrated to anybody who wishes to give feedback. Based on this feedback, the product backlog might be altered, appended-to or even detracted from. This vital part of a sprint keeps the project up-to-date with the *current* requirements instead of the *initial* requirements.
- Finally, before continuing-on to the next iteration, the team should have a **Sprint Retrospective**. During this retrospective, each member looks back to try to see what part of development could have gone better. Discussing improvements should be solution-oriented, and helps the team **continually improve**.

It is important to note that, by their definition, no model of reality fits it perfectly. Thus the environment under which we will develop our system does not entirely fit into the Scrum mold: our core team consists of two members, where there is a relatively strict division as to whom is supposed to work on which part of the system in order to prevent conflicts in predefined research guidelines. As such, development of Zeeguu UMR follows a slightly different (although heavily influenced) workflow with regards to what has been discussed above:

- Our primary supervisor functions as the **Product owner**, whom we meet at the end of each iteration in order to **demonstrate** the product and determine the (next sprint’s) **backlog**. In addition, each sprint is closed with a **Pull Request** to master (more on Git in the section below), where

the supervisor has the ability to review the quality of the most recent code produced.

- The **Scrum board** is available online, through Github's *Project Board* feature<sup>2</sup>. Tasks are assigned by assigning a contributor to a particular issue listed on the board, and are closed once they arrive in the **Done** column. This allows us to coordinate progress even if all members of the project are in a remote location.
- The team develops at the same location when possible, where the beginning of the day is used in order to discuss what feature we are working on and any problems we might face. When a particular problem seems to linger on, **Pair Programming** is applied in order to accelerate its solution. This is a method borrowed from the **Extreme Programming** philosophy, in order to accommodate the absence of a dedicated Scrum Master eliminating waste.

In addition, in order to further reduce time not allocated to actually implementing valuable features, we have tried to automate many tasks in our development life-cycle. Continuous Integration makes sure our code is tested every time we commit a change, Continuous Deployment allows a running version of our latest in-development code base to be available at all times, and an online hosted Documentation keeps all descriptive information on a publicly accessible location.

---

<sup>2</sup> <https://help.github.com/articles/about-project-boards/>



## PLATFORMS AND TOOLS

---

Any modern project has its dependencies. Following is a list of all notable platforms and tools used in this project, what they are, and why they were used.

- **Flask.** The Flask microframework allows us to run a RESTful server in Python, which was chosen for its comprehensibility, simplicity and its active community. Furthermore, it allows us to share and reuse code more easily between other services of Zeeguu which also implement most of their functionality using Flask. Originally created as an april-fools joke<sup>1</sup>, Flask finds itself to be the most popular Python web development framework on GitHub.
- **Webpack.** When developing in native Javascript (see listing B.1), cross-browser support and backwards compatibility tends to be a constant source of trouble. The many fallback methods and workarounds needed in order to develop a stable application is not only error-prone, but also tends to clutter code with statements diverging from whichever task the set of instructions is meant to perform. Similarly, the latest advances in EcmaScript (see listing B.2) are not supported by most modern-day browsers, so the use of them is highly restricted.

In order to write in relatively clean and maintainable EcmaScript code whilst benefiting from the broad range of compatibility native Javascript provides, we decided to make use of code transpilation<sup>2</sup>. Webpack is a transpilation tool that allows us to transpile our EcmaScript 2015 code into backwards-compatible Javascript whilst minifying and compiling the modules into neat and efficient files.

- **Documentation script.** To make sure our documentation remains up-to-date, we created a shell script that can automatically generate documentation using the in-code documentation strings and additionally provided

---

<sup>1</sup> <http://lucumr.pocoo.org/2010/4/3/april-1st-post-mortem/>

<sup>2</sup> Where one programming language is converted into another programming language whilst maintaining the same functionality.

manual files. This allowed us to easily include updated documentation with each code change requested to merge with the master branch. The script uses the ESDoc<sup>3</sup> tool to generate a nicely readable and web-hostable set of files.

- **Development scripts.** Similarly, we defined a few scripts that would make it easier for new developers to start using the system. A setup script can be called to automatically setup a virtual environment with all dependencies required to start working immediately.

---

<sup>3</sup><https://esdoc.org>

Listing B.1: Class inheritance in native JavaScript

---

```
var Snake = function (weight, height, isPoisonous) {  
    Animal.call(this, weight, height);  
    this.isPoisonous = isPoisonous;  
};  
Snake.prototype = Object.create(Animal.prototype);  
Snake.prototype.constructor = Snake;  
  
var Human = function (weight, height, hobby) {  
    Animal.call(this, weight, height);  
    this.hobby = hobby;  
};  
Human.prototype = Object.create(Animal.prototype);  
Human.prototype.constructor = Human;
```

---

Listing B.2: Class inheritance in EcmaScript 2015

---

```
class Snake extends Animal {  
    constructor (weight, height, isPoisonous) {  
        super(weight, height)  
        this.isPoisonous = isPoisonous  
    }  
}  
  
class Human extends Animal {  
    constructor (weight, height, hobby) {  
        super(weight, height)  
        this.hobby = hobby  
    }  
}
```

---

# WATCHMEN

---

As a side project, we created the Python library called Watchmen. It is a simple wrapper for the Readability library that caches parsed articles.

## C.1 PROBLEM

When retrieving results from cache fails to occur on the client-side we are forced to request a list of each non-cached-feed's articles from Zeeguu. Zeeguu has many users, and before Watchmen it simply performed the same operation for each user without reusing previous results. As parsing articles can be a relatively slow procedure, article requests send to Zeeguu tended to take multiple seconds before a reply was finally received. To remedy this delay, a Python library was proposed that would:

- Provide the ability to efficiently and effectively filter clutter from online articles, extracting their content and meta-data.
- Caching these results in memory and simply returning these stored values when requested for identical articles to be parsed.

## C.2 SOLUTION

To use Watchmen, simply include it wherever you desire to use it. Watchmen defines an `article_parser` instance, on which you can call the `get_article` method and pass the URL of the article in question. This method will return an `Article` object, a class the used Readability library documents very well. In general, the `Article` object will contain all information about the parsed article.

## C.3 ARCHITECTURE

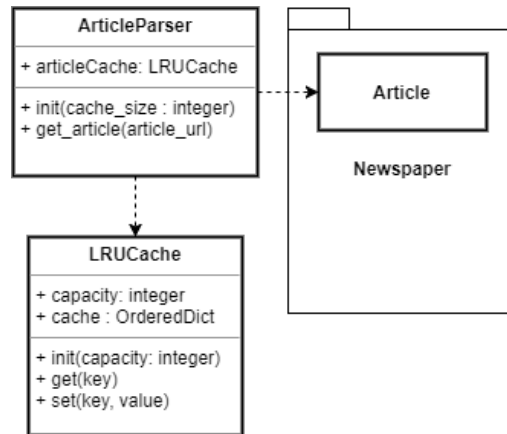


Figure 1: The watchmen library is a simple construction of three classes.

Article parsing and filtering was easily implemented by wrapping around the Readability library, which provides all the functionality we need.

Caching is done by using a simple map to bind keys to their values and implementing a Least-Recently-Used algorithm to manage its limited space in memory.

## C.4 RESULTS

After being used in Zeeguu-core, the efficiency of the `/get_feed_items_with_metrics` endpoint greatly increased. The delay is still large when both client and server suffer from an unfortunate cache miss, but any other case results in near instantaneous article retrieval.

## C.5 WHERE TO FIND IT

If you are interested in using Watchmen for your own project, or simply are curious to see the sources, you can check out our public repository<sup>1</sup> on GitHub.

<sup>1</sup> <https://github.com/mircealungu/watchmen>

## HOW TO CONTRIBUTE

---

Contributing to Zeeguu UMR is free and openly encouraged. Reading this thesis should have given you some general insight into our architecture, but we advise you to look into our online documentation<sup>1</sup> as well.

After doing so, try to fork our repository<sup>2</sup> on GitHub. You can send pull-requests to our repository once you are done developing a new feature or improvement, and we will try to review it.

Keep in mind that your alterations will need to be in line with our style guidelines (both code and UI), and our current road map. To ensure you are in sync with *Team Zeeguu* when developing your next groundbreaking piece of code, please make an issue on our GitHub page describing the feature you wish to implement. We might even provide some friendly advice!

---

<sup>1</sup> <https://mircealungu.github.io/Unified-Multilanguage-Reader>

<sup>2</sup> <https://github.com/mircealungu/Unified-Multilanguage-Reader>

## NOTABLE CODE-SNIPPETS

---

In this appendix we listed some interesting code snippets taken from the UMR source code. These snippets are meant as supplementary information to support some of the discussed Architecture, but is not intended to cover the entire code base.

Listing E.1: Method in FeedSubscriber that opens the dialog menu.

---

```
open() {
    UserActivityLogger.log(USER_EVENT_OPENED_FEEDSUBSCRIBER,
        this.currentLanguage);
    let template = $(HTML_ID_DIALOG_TEMPLATE).html();
    swal({
        title: 'Available Sources',
        text: template,
        html: true,
        allowOutsideClick: true,
        showConfirmButton: false,
        showCancelButton: true,
        cancelButtonText: 'Close',
    });

    this.languageMenu.load(this);
    this.load();
}
```

---

Listing E.2: Method in SubscriptionList to be called when the internal state has changed. Fires a changed-event.

---

```
_changed() {
    document.dispatchEvent(new CustomEvent(
        config.EVENT_SUBSCRIPTION,
        {"detail": this.feedList}));
}
```

---

---

Listing E.3: Registering a listener for the changed-event of SubscriptionList.

---

```
document.addEventListener ( config.EVENT_SUBSCRIPTION,  
                           function (e) {  
                               articleList.clear ();  
                               articleList.load (e.detail );  
                           });
```

---



## EXAMPLE EXTENSION

---

In order to showcase the power of some of our design choices, this appendix will demonstrate (in abstract) how one could implement a new requirement without changing a single line of code.

In chapter 8, we discuss the possibility of including images in the listed article. Below we list the 4 general but required steps that would allow us to make this potential feature a reality once Zeeguu decides to send hyper-linked images along with each article.

1. **Add the corresponding image parameter to the article template.** Due to our system simply constructing templates based using the JSON data returned by Zeeguu, additionally provided data can immediately be included into the template.
2. **Modify CSS to fit the desired styling.** A simple square image might not be the preferred layout.
3. **Increase version in `package.json`.** In order to prevent users not being able to see your changes due to file-caching, increase the minor version by one digit. This will result in differently named files, so make sure the HTML documents still refer to the proper files. Update `setup.py` as well for consistency.
4. **Build deployment files with Webpack.** The differently named files will need to be generated first. Simply use the webpack command (or use the automated deployment script).

These simple additions can result in a significantly altered design, turning Figure 5 from chapter 4 into Figure 2 from chapter 8.

# BIBLIOGRAPHY

---

- [1] M. Avagyan. An Open Adaptive Online Language Practice Platform. 2017.
- [2] Timothy Bell. Extensive reading: Why? and how? *The Internet TESL Journal*, 4(12):1–6, 1999.
- [3] D. Chirtoaca. Apollo. 2017.
- [4] P. Giehl. Zeeguu Translate Application. 2015.
- [5] William Grabe and Robert B Kaplan. *Theory and practice of writing: An applied linguistic perspective*. Routledge, 2014.
- [6] Stephen Krashen. We learn to write by reading, but writing can make you smarter. *Ilha do Desterro A Journal of English Language, Literatures in English and Cultural Studies*, (29):027–038, 1993.
- [7] Mircea F. Lungu. Bootstrapping an ubiquitous monitoring ecosystem for accelerating vocabulary acquisition. In *Proceedings of the 10th European Conference on Software Architecture Workshops, ECSAW '16*, pages 28:1–28:4, New York, NY, USA, 2016. ACM.
- [8] C. McCarthy. Reading Theory as a Microcosm of the Four Skills. *The Internet TESL Journal*, 5(5), 1999.
- [9] S. Namhee. The Effects of Extensive Reading on Reading Comprehension, Reading Rate, and Vocabulary Acquisition. *Reading Research Quarterly*, 52(1):73–89, 2016.
- [10] J. Oosterhof. Making reading in a second language more enjoyable. 2016.
- [11] Willy A. Renandya. The power of extensive reading. *RELC Journal*, 38(2):133–149, 2007.
- [12] L. Schwab. Using rss Feeds to Support Second Language Acquisition. 2016.
- [13] Jeff Sutherland. *Scrum: The Art of Doing Twice the Work in Half the Time*. Crown Business, New York, NY, USA, 1st edition, 2014.