Implementing WebLinDa

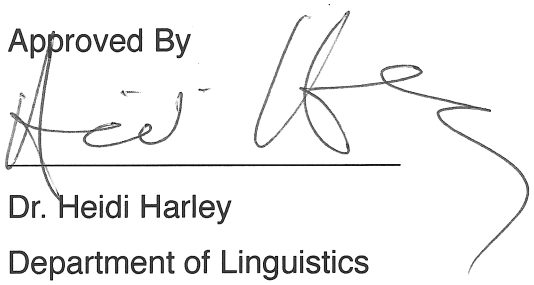
By Michael Tremel

A Thesis Submitted to the Honors College

In Partial Fulfillment of the Bachelors degree
With Honors in
Linguistics

THE UNIVERSITY OF ARIZONA
MAY 2011

Approved By



Dr. Heidi Harley
Department of Linguistics

WebLinDa

Abstract

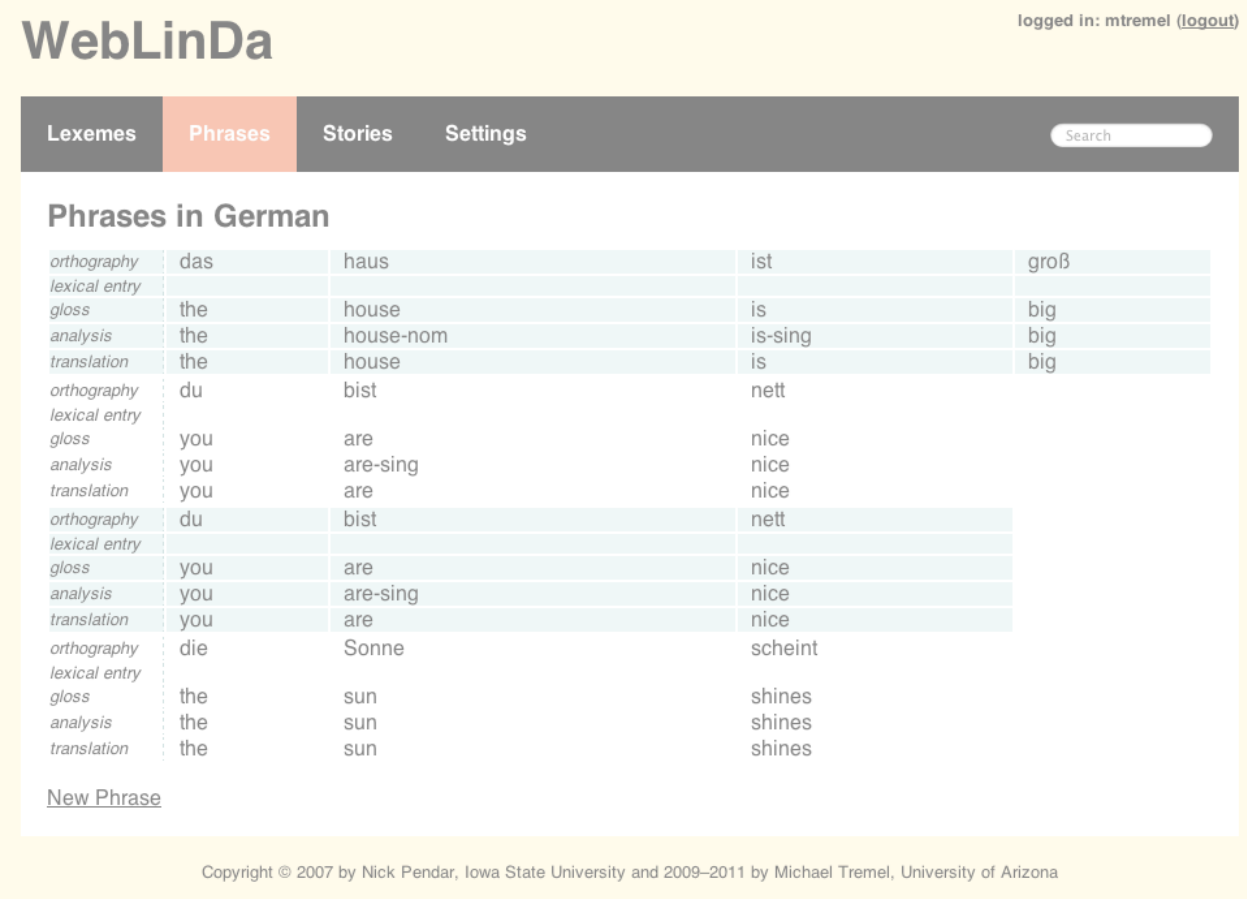
This honors project set out with the goal of replacing a user-end software for linguists, known as WebLinDa. The software is an inter-linear gloss database and is available as a web application, freely available to linguists around the world, but also freely available to the public as means of documenting language samples. The web application was written with the popular Ruby on Rails framework, was designed from the ground up with a new, more compact data structure, and enables very complex queries of the dataset. Special attention to making using WebLinDa the best experience possible was given, with a focus on speed, work-flow, and maintainability.

Contents

Sources and Documentation 13

Introduction

WebLinDa (*Web Lin*guistics *Da*tabase) is an inter-linear gloss database web application, currently accessible freely online[1]. It is designed from the ground up to support linguists, with both data recording and data-manipulation and querying in mind. Interlinearly glossed texts are the standard accepted by linguists to store, represent, and otherwise work with texts in different languages.



Such glosses offer linguists a way to store and work with annotations about an utterance, and in this way can be seen as metadata to that utterance.

Die Sonne scheint.
the sun shines
the sun shines

[1] http://weblinda.heroku.com

The sun is shining.

The above is an example of an inter-linearly glossed utterance of German, providing a lot of information about each element and a final translation into English. Traditionally, they consist of four lines of text: the first line is the original utterance, i.e., the source language orthography or phonetic transcription; the second is a word-by-word translation, known as the gloss; the third a breakdown of each word in the gloss to the morphemic level; and finally, a more free translation to the target language. Linguists use the format to represent and study a language's structure. WebLinDa enables this, providing a database structure to encode these levels, and further enables complex searches at any of these levels.

WebLinDa was initially conceived and developed only as a web application. The primary goals for this design decision were accessibility and ease of development. Starting with the latter, I was already familiar with web applications, including their design and deployment. This allowed getting from design to a working product very quickly. A more important deciding factor was the consideration of application accessibility, both physically around the world and also from a software perspective. WebLinDa is inherently a multiuser system, enabling multiple linguists to work on a single language project and benefit from other's work. A unified, server-client relationship eliminated many difficult syncing problems between these different users by providing a single, globally accessible server-based application. An exciting benefit of this model is that the application is just that, globally accessible by any computing device with a web browser, from normal computers to pocketable smart phones. All of these devices can use the system with little or no modification of the application. End users also benefit from having a constantly up-to-date application with full backups, and because of this design, enabling these benefits for endusers is trivial. This design decision was truly a win-win, both for users and developers. This was the consensus of my clients in the linguistics department, but also of the original author of WebLinDa, and me.

WebLinDa was originally written by Nick Pendar at University of Iowa. It is originally written in Python with a custom web framework and strongly attached to MySQL, a SQL database server. The project was donated to the University of Arizona

where it was already used in a variety of linguistic projects. I was originally commissioned to setup and deploy the existing application, do basic testing and user support, as well as add a few missing features. I had some success in doing this, but I encountered a number of limitations, not the least of which was the difficulty of dramatically changing the program while there were active users of the application. It was clear with later feature requests like more advanced search and visualizations of the data and its relations that the existing system would have a hard time being adapted to these constraints. The existing HTML user interface also had notable problems with mobile devices and even some desktop browsers. For these reasons, I decided to rewrite the application while concurrently maintaing the existing system, with the hope of replacing it.

To complete the honors requirement at the University of Arizona, one must undertake an honors project of one's choice. At the university, I've studied both linguistics and computer science, and hope to one day work on linguistic tooling. Considering that I had already worked on the project and was already familiar with its needs, the project was a perfect fit. With my honors project advisor, Heidi Harley, I set the following goals: determine and plan what a more modern WebLinDa would look like, chose a development language and framework, rewrite WebLinDa, and identify and help with an appropriate deployment plan. Further, since I won't be able to work on this project forever, extensive documentation regarding the design and deployment decisions, as well as documentation of the source code itself were a high priority, since this was a major change from the original version of WebLinDa. The written portion of the documentation serves to fulfill the paper requirement of the honors project.

Interlinear Gloss Database

As stated, an interlinear gloss (ILG) is a indispensable tool of linguistics. It is used to document, preserve, and analyze speech data. A gloss is composed of the original source data, say a sentence, from a source language, a translation into a target language for ease of understanding, and a number of lines of metadata known as the gloss. The gloss is a word-by-word, literal translation of the source language into the target language. It differs from that final translation in that it is not a free translation, expressing the content of the the utterance in the best way for the target language—

indeed, it is usually ungrammatical when taken as a separate utterance of the target language. Further lines in the interlinear gloss narrow the unit-by-unit translation, to, for example, a gloss that takes into account individual morphemes, and gives the appropriate translations for those smaller units (Maeda and Bird 2). Importantly, the first two or three lines are lined up vertically (one visual improvement I made during my rewrite of WebLinDa).

orthography	das	haus	ist	groß
lexical entry				
gloss	the	house	is	big
analysis	the	house-nom	is-sing	big
translation	the	house	is	big

This vertical grouping represents a relationship indicating meaning of a unit of speech from the source language. Linguists also use these glosses to store more complicated metadata, depending on the needs of the project, e.g.: prosodic or semantic information and even grammaticality judgments—a feature that was added in my subsequent revisions to WebLinDa. The database aspect is just that, a collection of these ILGs in a computer database, that is both easily transmittable and searchable. Computerizing ILGs brings a number of obvious benefits to linguists, and WebLinDa has so far proven quite successful.

CRUD

At its very simplest, WebLinDa is a CRUD (Create Update Read Delete) application. Utterances are stored as discrete units, or records, and are each assigned an identification number. WebLinDa enables four basic types of operations on this data: *Create*, *Read*, *Update*, and *Delete* (Martin 381). These four supported operations, or in web parlance, *verbs* are deceptively simple, and enable all database options. WebLinDa stores its data permanently in a SQL server, and these supported actions translate nicely into SQL (though they are certainly not limited to this: they could, instead, for example, be implemented with objects in memory, normal text files, etc.): In SQL, *Create* becomes *INSERT, Read* becomes *SELECT, Update* becomes *UPDATE*, and *Delete* becomes *DELETE*. All WebLinDa operations of the database eventually are translated into one of these four actions.

Database and Relations

At its most basic level, WebLinDa operates on its *Lexicon,* a collection of words, more or less, that contain all of the glossed information: the original orthography transcription (the source), a word-by-word gloss, a finer, perhaps morphemic level gloss, and a final translation to the target language. All in all, the database consists of these records, each tagged by an identification number. Utterances, and then the larger Texts are built using these Lexicon records and are linked together by the representative type: Utterances are ordered lists of Lexeme items, and a Text, an ordered list of Utterance items. These links are what are known as relations, and are easily implemented with SQL databases. This compactly stores the data by avoiding needless duplication, but also affords complex querying of the database without seriously straining resources. For example, consider the following query: “Locate every utterance that a certain Lexeme is used in.” This type of query is simple with said system.

This is all implemented behind the scenes efficiently in the SQL layer with what is known as a Join Table. This separate table stores records linking, for example, the Lexeme model with the Utterance model, and the Utterance model with the Text model. This correctly and efficiently enables the many-to-many relationship desired. For example, an Utterance model (representing a sentence), has many Lexeme models associated with it, and just the opposite is also true, a single Lexeme model (a single word, for example) can have a relationship, or in this case, be a member of, several Utterance models, or sentences. This simple, but well-known design is one of the major design changes in my rewrite of WebLinDa. The older version of WebLinDa needlessly copied data between the different models, while the new version stores it more compactly. More importantly, this new model enables much more advanced queries on the dataset than the old model allowed: for example, finding all sentences in which a single word appears is a trivial SELECT on the appropriate Join Table.

User Interface

As stated, WebLinDa is a web service and web application, which means that its user interface was created using web technologies. In this case, that means HTML, CSS, and JavaScript. Each resource provided by WebLinDa is transformed by the

application into an HTML document and transmitted back to the the client. Using web technologies means the enduser only needs a web browser to access WebLinDa. I decided early on though, that this would have to be a modern browser. At the time of writing, there are numerous supported and tested platforms: iOS 4.3 (iPad, iPhone, etc.), Safari 5, Chrome 10, Opera 11, Firefox 4, and Internet Explorer 9. Though in truth other and older web browsers will probably work without a hitch (except for the Public Access application[2], which requires a very modern browser), I feel confident that I have captured the vast majority of devices and users.

A major goal for the user interface of the rewrite of WebLinDa, was to improve navigation of the application and beautify the existing look. The old version of WebLinDa uses a complicated frames system, which breaks navigation on many browsers, and makes the URL system useless, e.g., you cannot copy a link to, say, a specific utterance in the database and send it to a friend with the existing system. The rewrite from day one addressed this, conforming to the way resources ought to be accessed on the web. Though subjective, a second goal was to improve the look of WebLinDa, and hopefully use some newer CSS techniques to provide things like nice background gradients and curves. Hopefully I have come near to accomplishing this goal, but failing that, it could be easily redesigned. WebLinDa is completely styled using CSS, and all HTML documents are separated from application code in template files. This all makes it much easier to change than the original ever afforded.

API

One thing very nicely afforded by the traditional CRUD model is the ability to quickly and simply draw up an externally accessible API, or *Application Program Interface.* By building WebLinDa as a web application, we automatically gained an interesting and useful property of the web: everything accessed from the web is considered to be a Web Resource and is identified by its unique URL, or Uniform Resource Locators, and these resources are naturally accessed and modified using a set of actions or supported "verbs": GET, PUT, POST, DELETE, all of which are implemented over standard HTTP, the transmission protocol a web browser uses to

---

[2] See section “Public Access Application”

communicate with the server. This forms the basis of RESTful (Representational State Transfer) communication, and WebLinDa implements that, becoming a RESTful web service (Tomayko). WebLinDa was designed as a web service that is able to provide support for these verbs on its resources, say an example glossed utterance. These RESTful operations are very easily implemented using CRUD operations, making it very easy to grant very flexible access to the data WebLinDa contains. In response to a RESTful request to the datastore, WebLinDa returns a JSON (JavaScript Object notation—a very terse, easily parsed and human readable data format) document that can be consumed by another application or service. This provides a very powerful and flexible framework for producing applications that can be used to access stored data. Across the Internet, this technique is widely successful, but perhaps is best seen in Twitter, which really took off by granting access to its data to third-party implementations or clients for mobile devices, computers, etc.. This API of sorts for WebLinDa will hopefully one day enable all kinds of varied applications, for example a native application for iOS (iPhone, iPad, etc.) or Android. As stated, one of the original design goals of WebLinDa was to provide the public access to the data in a clean and efficient way. This public access application is actually a separate application from WebLinDa, one that uses WebLinDa's RESTful API.

The Public Access Application

Part of the point of rewrite was to provide the public, specifically the speech community of a language documented by linguists in WebLinDa, complete access to its stored data. This will be the Public Access application. In this way, any work done by linguists on a language can directly benefit that speech community. Because most of these people will not be linguists, effort was made to make it as simple and easy to use as possible, though it provides much of the main reading and search functionality of WebLinDa proper. It too is implemented as a web application, though differs slightly in that it is not a web application run on a server—it does not produce any data. Instead, it simply consumes data from WebLinDa. Although this application could have been quite easily baked into WebLinDa's existing code, the application instead consumes the API published by WebLinDa. This has several benefits: first, it makes WebLinDa a primary consumer of its own API, ensuring it is up-to-date, complete, and functional; second, it

clearly separates code and projects, making maintenance of both systems that much easier.

The new application is a good example of how an external, third-party application might look and operate. It is written using only web technologies, HTML and JavaScript, and has no other dependencies on the traditional server model, outside an Internet connection to WebLinDa's API. This makes the application a statically served HTML document that is quickly served and efficiently cached. It underlines the viability of such a system, so much so that it seems possible to write WebLinDa itself in such a way. It also scales normally server-intensive operations with each user, which is of course linear, perfectly. This has the benefit of being cheap to deploy and run, as normally server intensive operations are performed locally on the client. It can also leverage many of the new exciting web technologies, like local storage for efficient caching and offline read-only access of the dataset. Of course, it also accomplishes its original goal of being an easy way for the public to access WebLinDa's data, but it also a shining example of using the API and the crowning jewel for the technology stack used in this project.

User Usage

A primary goal of WebLinDa was to make common user interactions, or usage scenarios, function as simply, quickly, and efficiently as possible. To better accomplish this, I took time to analyze my use and how others use WebLinDa. Through this approach, I identified many optimizations for the user interaction:

- Smart tabs and appropriate key bindings: users seem to love their keyboard and abhor using the mouse. I've found the more a user has to use a mouse, more specifically switch between a mouse and a keyboard—and data entry for WebLinDa is inherently keyboard bound—is frustrating. To this end, all user-facing forms are correctly written to enable the cursor flow of the cursor using the tab key, and appropriate key bindings, like enter, trigger actions, like form submission. These bindings are documented in the user help section of WebLinDa.
- Sensible default actions: the flow of entering user data into the system can become repetitive. To this end, default actions point the user in the right direction. When adding lots of new entries, the form for adding becomes easy to access at the top

of the page. When the user first logs in, the system assumes the user wants to view the data he was last remembered as using, and is so redirected.

- Inline editing: editing data in any application can be a pain, but a lot of this can be avoided by speeding up the process. To edit a piece of information, more often than not, the edit can be performed inline, without reloading any other pages.
- Fast website: perhaps the most important optimization of all, and it is not immediately obvious, is to ensure the webpages are delivered to the user as quickly as possible. A good rule of thumb is to note that a user notices a delay when anything takes longer than a couple hundred milliseconds. To that end, pains were taken to make sure WebLinDa can serve pages under that time, even under load. In most cases, this means caching of data and compression of downloaded elements, like the CSS, HTML, and JavaScript. Google's Page Speed[3] tool was used to this end and was tremendously helpful. In the cases where delivering a quick response is not possible, the task becomes asynchronous, and is reprinted to the user as a progress dialog, in order to distract him into thinking it is working faster than it actually is (audio file uploads are a good example thereof). Getting things as fast as possible helps users speed along through the system, but also helps squeeze the very utmost out of the servers that run WebLinDa.

Administration

WebLinDa supports an unlimited number of languages, and by its very nature is multiuser. Initially, support to accomplish this automatically was nonexistent, and relied heavily on the system administrator to add new languages to the system, grant users access to said language projects, and so forth, all manually. This was both time consuming and error prone. To help, I wrote several administrative tools to help streamline the process and bring down the difficulty level of actually doing so. These tools enable user management, including the ability to edit all user-related items, and even reset passwords, as well as add new users, and to modify and add language projects. This system is, however, an independent program with its own coding style,

---

[3] http://code.google.com/speed/page-speed/

documentation, and even deployment. This administrative functionality is now part of WebLinDa proper, integrating perfectly in all mentioned areas.

Implementation Details

WebLinda was originally written in Python[4], leveraging standard parts of the Python toolkit, but using a custom framework to do so. This led to some understandable difficulty in becoming first acquainted with the code-base, and I would imagine it would be a problem for new maintainers of the project as well. This, along with gaining experience of a new, well-known framework were the primary motivations for selecting a new, established framework to write WebLinDa with. I looked toward using Ruby[5] and Ruby on Rails[6] for the rewrite. Ruby on Rails is a very famous, opinionated web framework that has a strong emphasis on security. Another strong attraction was its powerful Object Relational Mapping (ORM) system, that helped me quickly model the new database layout.

Successes

The decision to use Ruby on Rails and its associated technologies turned out to be very successful. In a short time, I was able to rewrite the majority of WebLinDa's existing functionality, while fixing a few outstanding issues with the original codebase. In addition to this, I was able to dramatically improve the end-user experience, bringing an entirely new interface, by testing for the usage patterns of linguists on the program, which allowed me to focus my efforts of optimization. Administrative use of WebLinDa was also improved with the integrated administrative tools. Finally, the deployment of WebLinDa was dramatically simplified from a multi- to a single-step process, allowing a single person to easily grasp how WebLinDa works and what technology stack it requires. WebLinDa will hopefully go on to benefit linguists and speakers of studied languages in the future with these new features and simplified interface. I have also benefited greatly though, expanding my knowledge of Ruby on Rails, databases, as well as how linguists best work with and store collected data.

---

[4] http://www.python.org/

[5] http://www.ruby-lang.org/en/

[6] http://rubyonrails.org/

Conclusion

As stated, WebLinDa is a web-application, inter-linear gloss database for linguistic data. As a web-application, WebLinDa serves the needs of teams of linguists working on the same project, but also provides access to its data to the general public. At a technical level, this rewrite of WebLinDa simplifies the data structure somewhat, by relying on advanced features of the underlying datastore, the SQL database. More importantly though, storing the data like this enables more complex search queries against the database that were before impossible. Finally, special attention was given toward optimizing the completely rewritten user interface. Rewriting WebLinDa using Ruby on Rails and its associated technologies turned out to be very successful. Hopefully WebLinDa will go on to live a successful life with linguists and speakers alike.

Sources and Documentation

WebLinDa is an opensource application. The source[7] and documentation[8] can be found online.

---

[7] https://github.com/weblinda/weblinda

[8] http://weblinda.github.com/weblinda/

Works Cited

Maeda, Kazuaki, and Steven Bird. "A Formal Framework For Interlinear Text." (2000): 1-17. Print.

Martin, James. Managing the data-base environment. 2, illustrated. Upper Saddle River, NJ: Prentice-Hall, 1983. eBook.

Tomayko, Ryan. "How I explained REST to my wife." The various writings and linkings of Ryan Tomayko. N.p., 12 Dec 2004. Web. 03 Mar 2011. <http://tomayko.com/writings/rest-to-my-wife>.