An Application for Automatically Translating Dynamic Web Content

by

Katrina Michelle Ellison

Submitted to the Department of Mechanical Engineering
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Mechanical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2011

Author .....
Department of Mechanical Engineering
January 14, 2010

Certified by .....
Anjali Sastry
Senior Lecturer
Thesis Supervisor

Accepted by .....
John H. Lienhard V
Chairman, Department Committee on Undergraduate Theses

An Application for Automatically Translating Dynamic Web Content

by

Katrina Michelle Ellison

Submitted to the Department of Mechanical Engineering
on January 14, 2010, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Mechanical Engineering

Abstract

This thesis describes an application, AutoLex, for translating dynamic content on websites that use the Django web framework. AutoLex retrieves translations from the Google Translate service, stores them in a database using a single table, and serves them via a user-defined accessor. In doing so, AutoLex offers website owners a fast, cheap way to translate large amounts of content and to enable multilingual communication between users. Future improvements will include automated accessors, hooks for integration with caching applications, and improved translation generation and display.

Thesis Supervisor: Anjali Sastry
Title: Senior Lecturer

Acknowledgments

Many thanks to Aaron Beals, for overseeing this project and giving me the opportunity to stretch far beyond my proven capabilities; to Anjali Sastry, for making the time to supervise a thesis outside of her main area of interest; to Aaron VanDerlip, for his invaluable technical support; and to everyone at the Global Health Delivery Project, especially its intrepid leader, Dr. Rebecca Weintraub.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is a design document for an application, referred to here as AutoLex, that automatically translates *dynamic* (user-generated) content on websites built using the Django web framework. The application was developed for integration with GHDonline, a forum-type website serving the international medical community.

1.1 Project Context: Global Health Delivery Project

The Global Health Delivery Project (GHD) is a group of researchers, doctors, and other professionals whose goal is to improve healthcare delivery in resource-limited settings through research, education, and collaboration. In such settings, doctors frequently treat an immense variety of conditions while working in isolation. GHD's executive director, Rebecca Weintraub, explains that the project was begun to ameliorate "the loneliness of being a clinician without the backbone," of mentors and colleagues found in medical institutions in the developed world[1].

GHD's main vehicle for spurring collaboration is GHDonline (http://www.ghdonline.org), a collection of online resources whose centerpiece is a network of virtual communities. The communities are a place for healthcare professionals worldwide to ask questions and share best practices. Some communities, like Global Surgery & Anesthesia, are directed mainly at clinicians, while others, like Global Health Nursing & Midwifery

[1]Weintraub, Karen. "Second Opinions, Anywhere", 3 March 2010. *Boston Globe*.

and Health Information Technology, have a broader audience. Community members contribute by posting *resources* (including files and links) and starting *discussions* or posting *comments* on existing discussions.

Since its inception in 2008, GHDonline has attracted over 3,600 members from 135 countries. The GHD team has worked to attract non-English-speaking members to its site, but the language barrier impedes truly international communication: members typically interact only with others who speak their own language. The site currently offers translations through the Google Translate widget, but using it decreases page-loading speeds too much for it to be a viable long-term solution.

This thesis project was begun in September 2010 in order to address this issue. Its long-term goal is to integrate translated content into GHDonline itself. The thesis addresses the core technical aspects of the most ambitious part of this endeavor: real-time translations of user-posted content. A pilot version of the feature will be introduced to GHDonline in the spring of 2011.

1.2 Technical Overview

GHDonline has was built using Django, a web framework for rapid development written in Python. Django was first released in 2005 and has an active community behind it. As of this writing, there were 3,750 websites registered as being powered by the framework[2] and 18,812 members of the Django users Google group[3], Django's main email listserv.

Django's structure is based on the Model-View-Controller philosophy, but it does not follow it strictly. It is more often described as MTV: models, templates, and views. A project's *models* provide definitions for and accessors to underlying data, which is typically stored in a database. Its *views* include the logic for processing user requests and presenting data. The final formatting of that data is performed by HTML *templates*, which typically have a one-to-one mapping with views. While this

[2] *Django Sites*, 2008-2011. http://www.djangosites.org. Accessed 5 January 2011.
[3] *Django users*, 2011. http://groups.google.com/group/django-users. Accessed 5 January 2011.

description leaves out the controller-like function performed by the framework itself, it does enumerate the three main areas controlled by a Django developer.

A single project usually pulls together models, views, and templates defined in multiple places. To facilitate code re-use, most Django applications provide only a small set of features, and a Django-powered website uses a combination of applications to achieve its goals. The modularization of Django applications allows for rapid development of new websites and is an accepted best practice in the Django community. A single application almost always includes models, and sometimes includes views and templates as well.

1.2.1 The GHDonline Website

GHDonline consists of one main application, Community, and several smaller applications, some of which were built in-house and others which were downloaded from third-party developers. The Community application contains models connected to GHDonline's underlying database, as well as views and forms to process requests to display or change information. Separate directories of CSS and HTML templates are used to format that information and display it to users.

1.3 Project Objectives

The overarching goal for this project is to internationalize GHDonline by building a platform for translating users' contributions into multiple languages. This platform will:

1. Enable automatic translation of user-created dynamic content.
2. Maintain acceptable performance.
3. Have a clean API.
4. Be easily extensible into new languages and item types.
5. Avoid massive changes to existing code.

6. Be easily turned on and off.

1.4 Report Guide

Chapter Two of this report contains background information on the current state of internationalization and localization on the web and existing approaches to translating content.

Chapter Three describes the design requirements considered in the development of this solution, while Chapter Four discusses the specifics of its implementation. Performance evaluations and other metrics can be found in Chapter Five.

The report closes with Chapter Six, which includes a conclusion and recommendations for further work.

Chapter 2

Background

Understanding the requirements for translating of GHDonline's dynamic web content requires a general understanding of the meaning of localization and internationalization, knowledge of relational databases, and a brief overview of GHDonline's architecture.

2.1 Localization and Internationalization

The two-part process of readying a website to serve a multilingual audience and translating all of its content is known as internationalization and localization.

The need to ready websites for translation is large and growing. In 2010, only 27.3% of web users spoke English, the most widely spoken internet language. Their numbers increased 281.2% between 2000 and 2010. Meanwhile, the other 72.7% of the population, internet users speaking other languages, grew at more than one and a half times that rate, by an average of 444.8% per language[1]. As access to the web grows, its population of users will grow even more diverse.

[1] "Top Ten Languages Used in the Web". *Internet World Stats: Usage and Population Statistics*, June 2010. http://www.internetworldstats.com/stats7.htm. Accessed 5 January 2010.

2.1.1 Localization

Localization is the process of adapting a website's user interface for a particular *locale* (user location), which typically includes some combination of translating content, choosing appropriate colors and images, and modifying layout orientation. This thesis is mainly about internationalization (see below), but a brief introduction to localization is helpful for understanding the current state of multilingual content on the web.

.po and .mo Files

The current web standard for serving localized content is through the use of `.po` and `.mo` files. Every language that a website is translated into has its own `.po` file, a dictionary that connects each of the site's original text strings to its translated equivalent. `.po` files are human-readable and -editable; a site owner wishing to translate his website from English to French would type all of his website's text into a `.po` file and send it to a translator, who would fill in the French version of each phrase. The `.po` file is compiled into a machine-readable `.mo` file, which is used to actually serve the translated versions to site visitors.

While this approach works well for static content, it is less successful for dynamic content because each `.mo` file must be re-compiled every time any text on the entire website is updated. It is not feasible to do this on a production server for a website whose content changes multiple times per day.

For dynamic content, an alternate approach is to store each translation in a database along with the original text. When a user requests to view the text, the version that closest matches his language preferences is selected from the database and presented to him.

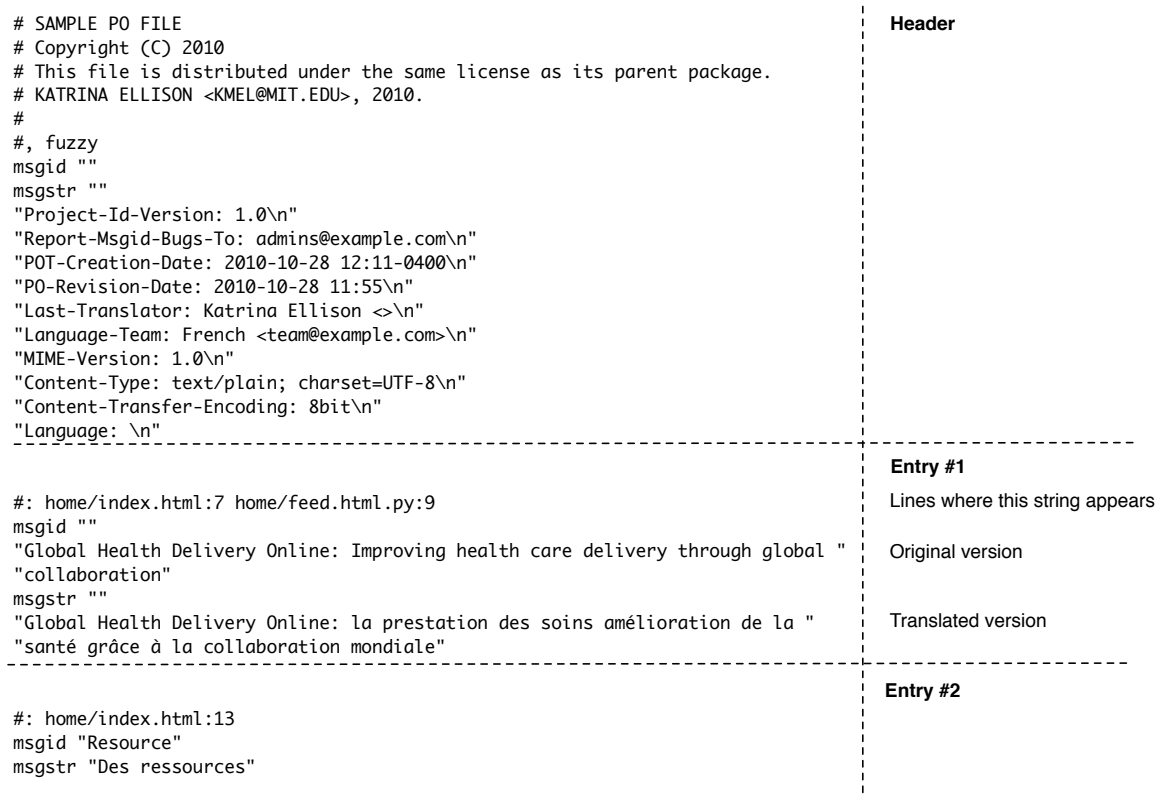

Figure 2-1: Each .po file begins with a header with information about its parent package, its creators, and its content. Translations are stored in entries that specify the original string, the translated version, and where on the website the original can be found.

Localization in the Django Framework

The Django Rosetta application[2] is designed to aid with localization by translating static text strings. It is one of the only Django applications to integrate with a machine translation service, such as Google Translate or Microsoft Translator. In spite of this strength, it is not appropriate for translating GHDonline's dynamic content because it is designed for use with `.po` and `.mo` files, not with text stored in a database.

2.1.2 Internationalization

A site's ability to be localized rests on its successful *internationalization*. Internationalization is the process of building the underlying infrastructure that will support multiple languages on the site. The internationalization process is especially involved when using dynamic content. Because such content changes frequently, its translations cannot be stored in `.po` and `.mo` files; instead, custom-built infrastructure for storing and serving translations is required.

Internationalization in the Django Framework

While no existing applications handle the actual translation of dynamic content, there are numerous apps for storing translations in a database and serving them to visitors, including Multilingual-Model, Transmeta, ModelTranslations, and MotherTongue. Most existing applications store translated content in one of two ways: either in the same database table as the original content, or in a new table created specifically to store translations. Fig. 2-2 shows the differences between these two options.

Storing Translated Content

Storing the new translation columns directly in the original table (Scheme 1 in the figure) is generally the better approach from a performance standpoint: accessing

[2] *Django-Rosetta*, 2010. http://code.google.com/p/django-rosetta/. Accessed 05 October 2010.

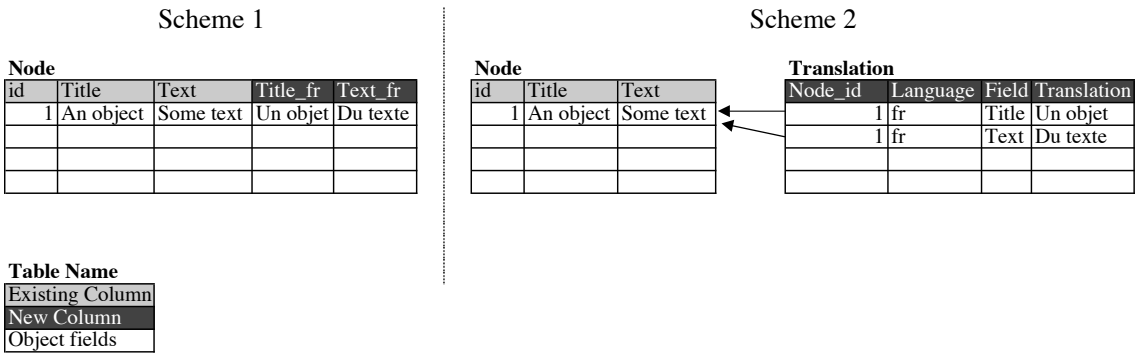

Figure 2-2: Two separate database schemes for storing translated content. Scheme 1 shows translations held in the original table, which typically improves lookup speed. Scheme 2 shows the fields in their own database table.

translations need not result in additional database queries, or in the more complex multi-table lookups. However, storing the translations in their own table (Scheme 2) makes it easier to save additional information about each translation (such as who created it and when it was created), add or change which objects are translated, and support new languages. Deciding between these two approaches requires careful consideration of an individual project's requirements, which is why no one method is dominant.

Serving Translated Content

While a variety of approaches to storage exist, nearly all of these applications serve the translated content by overwriting Python's default property accessor, the `__getattribute__` function. When a function asks for `object.text`, `__getattribute__` is what actually grabs the text and returns it. If it is overwritten, asking for `object.text` will not return the original version; it will return whatever the new `__getattribute__` function specifies. Many Django translation apps serve translated content by overwriting `__getattribute__` to return an appropriate translation instead of the original content.

Existing Applications

There are a variety of existing applications to assist with internationalization in Django. Below are two that were considered for use with GHDonline.

ModelTranslations ModelTranslations stores translated content using a the first scheme described above. At runtime, the application creates a set of Python descriptors for model fields that have been translated. For example, if the original field is called 'title', and the languages defined in the project's settings file are English and French, ModelTranslations creates descriptors named `title_en` and `title_fr`. The first time these descriptors are used, ModelTranslations automatically adds columns to the original database table to store the translations.

This approach prevents the performance degradation that would occur if translations were stored in a separate database table. At the same time, it sidesteps the need to update an application's models every time new fields or languages are added for translation. Because of this, ModelTranslations is currently popular within the Django community. However, the project's database tables still need to be updated, even though the models do not.

The ModelTranslations application was considered unsuitable because its extensibility is limited: adding new columns to existing database tables requires updating a project's database schema each time a new language or type of object is supported.

MotherTongue The MotherTongue application addresses the problem of serving translated content, while leaving it up to the user to determine where to store it in the database. MotherTongue overwrites Python's `__getattribute__` function to return translated content instead of the original. The site's URL schema is modified to automatically redirect users to different URLs based on their browser settings. For example, French-speaking visitors to an English-language site see pages with '/fr/' prepended to all URLs.

MotherTongue could be used to effectively serve translations, but it provides an incomplete solution because it does not provide a way to create or store new trans-

lations; nor does it address the performance concerns of retrieving translations from a large database. Additionally, because it overwrites `__getattribute__`, MotherTongue requires defining an additional property to access the original version of a translated field, limiting the clarity of its API. Therefore it was not used.

Each of these applications is used to translate text held in a relational database, the current standard for storing large amounts of content. A basic understanding of their functionality is helpful for evaluating potential translation solutions and choosing an appropriate solution in a given context.

2.2 Relational Databases

Data stored in a relational database is accessed through *queries* and held in memory before being displayed to users. Databases are arranged into *tables*, with each table containing one type of object. The values of each of the object's fields are held in *columns* in that table. Each entry has an identifying number, called its *primary key*, that is stored in one of the table's columns.

2.2.1 Performance

Querying the database is relatively slow compared to manipulating objects in memory, so it is good practice to keep the number of database lookups to a minimum. Additionally, more complex queries take longer than simple ones. In particular, lookup time scales with the number of tables that must be searched to find the required data; retrieving data from two or more tables is faster than retrieving it from a single table. Links between tables are made using *foreign keys*, described in Sec. 2.2.2 below.

An *index* can be added to a table to improve lookup performance. An index is a copy of a column that is used in lookups; it improves performance by creating a more organized structure. Indexes are helpful when created on columns that are frequently used as parameters in data selection queries[3]. On the other hand, using an index

[3]2011, PostgreSQL Development Corp. "11.1 Introduction". *PostgreSQL 8.2.19 Documentation: Chapter 11, Indices.* http://www.postgresql.org/docs/8.2/static/indexes-intro.html. Accessed on 23 December 2010

increases the time needed to change a database's entries because the index must be updated along with the original information.

The performance of relational databases can also be increased through the use of *caching*. Broadly speaking, caching is the minimization of database lookups through the storage of frequently-used objects in memory. This practice increases performance because lookups are frequently the slowest part of an application's processing time. It can be implemented in a variety of ways, and there exists a multiplicity of web caching applications.

In addition to taking overall database performance into consideration, AutoLex makes use of a special type of database constraint called generic foreign keys.

2.2.2 Generic Foreign Key Relationships

Foreign keys are a type of database constraint that allow information in different entries to be linked together. A foreign key constraint is applied to an individual table. For each entry in that table, the value in the foreign key's column is the primary key of something stored elsewhere in the database: it is a *key* to some *foreign* object.

In a regular foreign key relationship, the constraint on the original table specifies which table holds the foreign objects. All of the foreign objects must be in the specified table.

In contrast, a generic foreign key can point to an object stored anywhere in the database. Instead of a single constraint that specifies which table the foreign key points at, an additional column stores that information for each individual object.

2.3 GHDonline Website Architecture

Most of GHDonline is contained in one application, Community. Community holds all of the mechanisms for storing and serving GHDonline's dynamic content.

2.3.1 Storing Dynamic Content

GHDonline's user-generated content is stored in models called Discussion and Comment (for text), Artifact (for files and links), and UserProfile (for user information). Each of those models inherits from the Node model, whose database table holds the majority of these classes' fields, including the two fields that require translating: 'title' and 'text'. Internationalizing Node is the main focus of this project.

2.3.2 Serving Dynamic Content

The two views used most frequently to display user-generated content are the *community homepage* and the *detail page*. On each community homepage, a *feed* of featured and recently created items gives users a brief preview of the content available in that community. The feed length varies by community, but is generally 5-10 objects. Each object's title and a short snippet of its text is displayed on the community homepage.

Each object in the feed is linked to a detail page that displays the object's full text, its author, and other relevant information, including follow-up comments if the item is a discussion. Most detail pages contain information about only one object, but some discussions have many comments attached to them.

In general, displaying a feed requires retrieving and manipulating a large number of database entries, while displaying a detail page only requires using a small number of them.

Chapter 3

Design

3.1 Design Requirements

Specific considerations for fulfilling the goals described in the Introduction are listed below.

Integrate with a Machine Translation Service The main goal of the project is to enable multi-lingual user interactions. Translations must be obtained quickly and cheaply in order to achieve this. Since contracting human translators requires both time and money, this project makes use of a third-party machine translation service. Machine translation is inexpensive and fast, and its quality is improving.

Preserve Acceptable Performance GHDonline's mission is to connect users from around the world, including those from areas without high-bandwidth internet access. Minimizing page-loading times is necessary in these areas and appreciated in others. This feature can be integrated without increasing the loading times to unacceptable levels. 'Acceptable' is roughly defined as keeping a typical page's loading time to within one second or within 30% of its original loading time, whichever is higher. Performance can be further improved by using the module in tandem with other techniques, such as caching and database optimization.

Performance tests and metrics are focused on GHDonline's community homepages,

its slowest-loading pages containing the most translated items.

Accommodate Future Extensions Although most user-generated content is stored in one database table, Node, models that are part of the user interface must be translated as well. The application is easily extended to new models in order to accommodate these additional needs. It also makes it easy to add new languages so that GHDonline can support additional locales as its reach grows.

Use a Clear API for All Content Machine translation technology is improving, but is still far from perfect. Because this application relies heavily on machine translation, it is important to display the original text alongside the translated version. That way, users can read most of the text in their native language, but still refer to the original when the translation is not precise enough.

The standard Python accessor for getting the content stored in a particular field is the `__getattribute__` function. Many existing translation applications for Django overwrite this accessor so that it retrieves a translated version of that content instead of the original. Displaying the original content would require writing a different accessor. This configuration - using the original accessor for the translation and a new accessor for the original - is less intuitive than simply preserving the original accessor for the original content. To preserve API clarity, the application does not overwrite `__getattribute__`, instead relying on a different accessor to retrieve translated content.

Maintain Integrity of Existing Codebase Following the principles of modular design, internationalizing user-generated content should not require significant changes to a project's existing codebase.

Be Easily Enabled and Disabled Translations should be easy to turn on and off.

3.2 Solution Architecture

There are three major parts of the described solution: the translation service chosen to create translations; the database architecture chosen to store them, and the API chosen to access them.

3.2.1 Choice of Translation Service

Google Translate was chosen because it was free, well-documented, and easy to use. Its translations are relatively reliable, and their quality has been increasing over time.

Google Translate's Terms of Service include several clauses with implications for the module's design[1]. In particular, HTTP requests must be 5,000 characters or less, and results must be retrieved on behalf of a human end user. Google strongly recommends including the end user's IP address as a parameter in the request to make it easy to verify that the request was initiated by a human, not a robot or spider[2]. If the IP address is not included, it is more likely that Google will reject the request.

3.2.2 Database Architecture

A database schema of storing translations in a separate table was chosen due to the simplicity and flexibility of this approach. A series of pilot performance tests was conducted to ensure that choosing this architecture would not unduly harm performance. The results, shown in Fig. 3-1 show that while storing translations in the Node table is indeed faster, storing them in a Translation table increases loading times by only 25-30% when an index is added to the table's primary key column.

[1] "Google Translate API Terms of Use", November 2010. http://code.google.com/apis/language/translate/terms.html. Accessed on 13 December 2010.

[2] "Google Translate API: Developer's Guide", 2010. http://code.google.com/apis/language/translate/v1/getting_started.html. Accessed 13 December 2010.

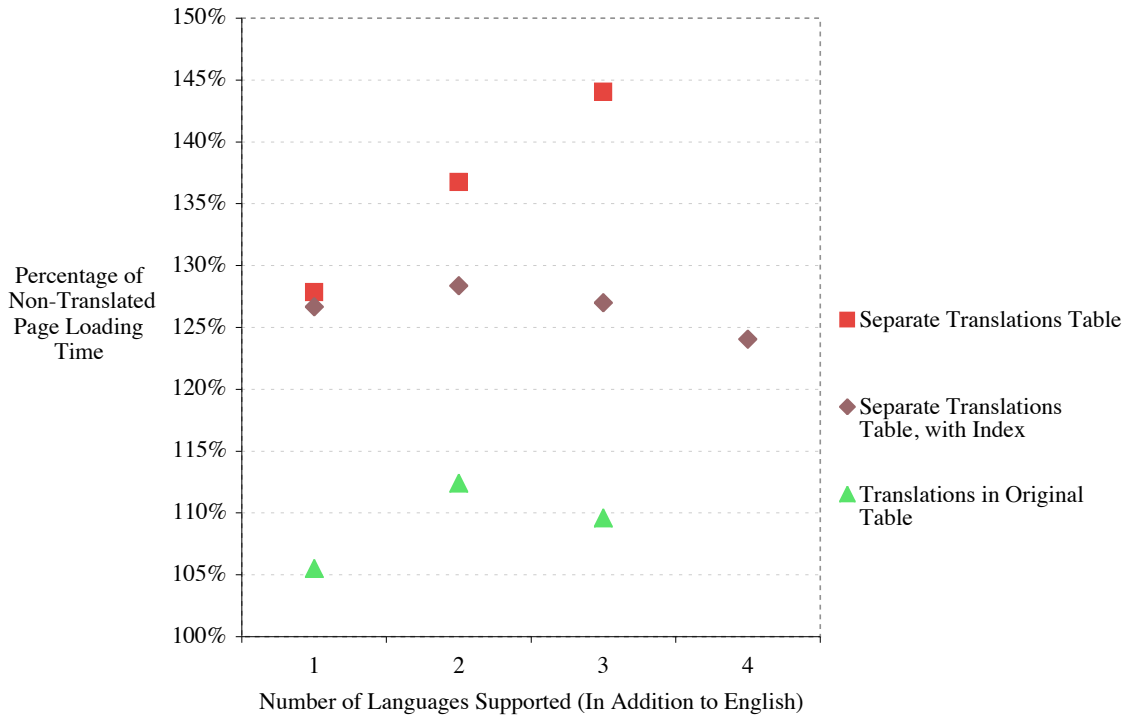

Figure 3-1: Time, shown as a percentage of reference loading time, to load a community homepage requiring retrieval of ten translations from the database for display. Storing the translations in the same table as the original text (triangles) increases page loading time by 5-15% over the reference. The performance impact of storing translations in a separate database begins at approximately 28% and increases linearly at a rate of approximately 8% per language added (squares), unless an index is used. The addition of an index on the primary key of the translations table (diamonds) prevents performance from decreasing as more translations are added. In these tests, each language supported adds approximately 11,000 objects to the database.

3.2.3 API

The application's *API* (Application Programming Interface) consists of all the methods that access its functionality, and what a developer would use to incorporate it into an existing website. The API has three methods, described below. Note that none of them require a language code as an input parameter; language detection is performed automatically within each method.

1. `make_translation` is used to translate one object, as for GHDonline's detail page.

   Input parameters: a translated object and an IP address.

   Modifications: adds one translation to the database for each field in the given object.

   Output parameters: none.

2. `make_multiple_translations` is used to translate many objects at once, as for GHDonline's community homepage.

   Input parameters: a dictionary of translated objects and an IP address.

   Modifications: adds one translation to the database for each field in all of the given objects.

   Output parameters: none.

3. `get_translated_version` is used to access the translated version of a particular field.

   Input parameters: a translated object and a field name.

   Modifications: none.

   Output parameter: a translated version of that field in the appropriate language.

Below is usage example showing code that could be added to Models, Views, and Template files to enable translation of a discussion object.

```
# In Models:
import translate
class Discussion(models.Model, translate.TranslatedItem):
      title = models.CharField()
      text = models.TextField()

      def get_title(self):
            return translate.get_translated_version(self, ‘title’)

      def get_text(self):
            return translate.get_translated_version(self, ‘text’)

# In Views:
# (Add the following to any view that will
# display the discussion’s title and/or text)
import translate
user_ip = request.META[‘REMOTE_ADDR’]
translate.make_translation(discussion, user_ip)

# In Templates:
{{ discussion.get_title }}
<br>
{{ discussion.get_text }}
```

Chapter 4

Implementation

The solution described in Chapter 3 is implemented as a separate Django application, AutoLex. In deployment on GHDonline, AutoLex is integrated with existing mechanisms for storing and serving dynamic content.

4.1 Overview

The AutoLex module has two functions: retrieving new translations from Google, and displaying stored translations from the database. These two steps are de-coupled so that each occurs at a different point in GHDonline's processing routine, as shown in Fig. 4-1. New translations from Google are added to the database before an HTML template is rendered, and translations are retrieved from the database when the translated fields are accessed in the template.

4.2 Making Translations

There are two different algorithms for making translations, the `make_translation` algorithm, and the `make_multiple_translations` algorithm. Below are use cases for each. In both cases, translations are displayed by calling `get_translated_version` from the template for each translated field.

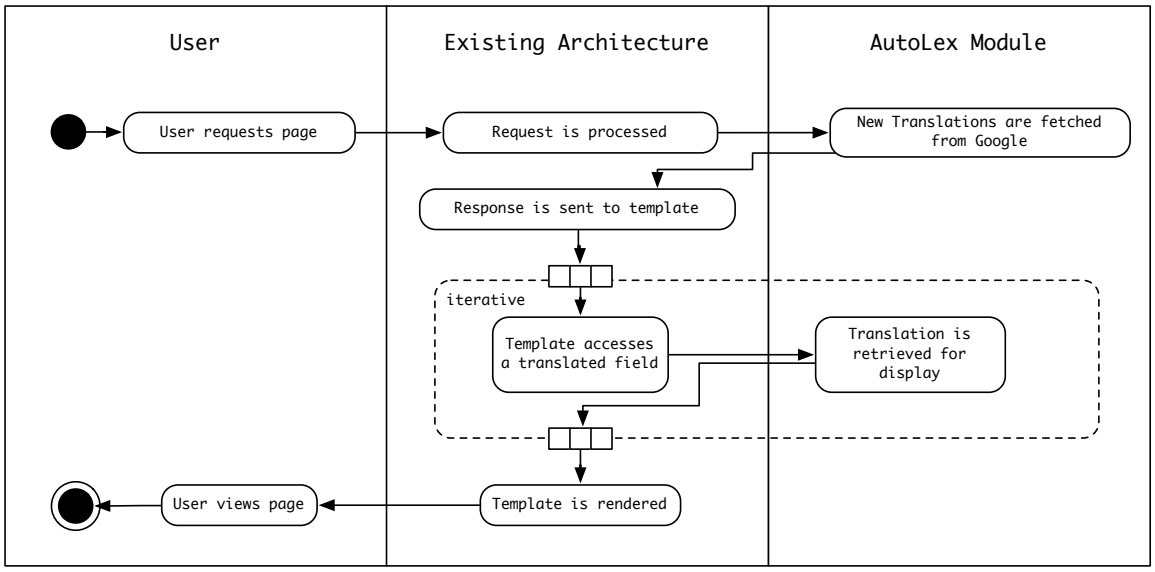

Figure 4-1: A high-level description of how using AutoLex affects a Django website's request-processing routine.

4.2.1 Case 1: Page with Few Translated Objects

For pages where only a small number of translated objects are displayed, such as GHDonline's detail pages, the translations are retrieved from the database and made, if necessary, one object at a time using the `make_translation` algorithm.

Algorithm

`make_translation` uses the following algorithm:

1. Check if translations are enabled. If not, return.
2. Get the user's language.
3. Determine which object fields need to be translated.
4. Get all translations for this object in this language from the database. Put most recent ones first.
5. Create a new unique `translation_set` id. All translations created right now will have this `translation_set` id, making it easy to differentiate between translations created at different times or by different users.

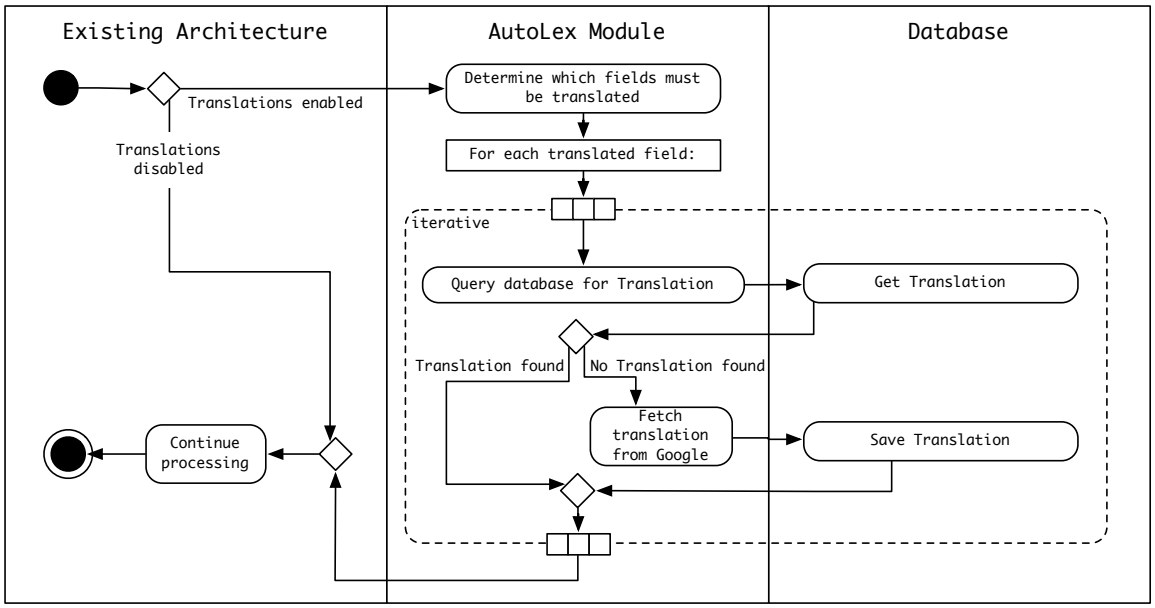

Figure 4-2: Make_translation is AutoLex's standard algorithm for translating a single object's translated fields.

6. For each translated field, see if any of the translations retrieved from the database are for this field. If not, get a translation from Google Translate.

7. Once all fields have been checked and updated, return.

A diagram of this algorithm is shown in Fig. 4-2.

4.2.2 Case 2: Page with Many Translated Objects

For pages with a large number of translated objects, such as GHDonline's community feed, using `make_translation` significantly increases loading time, as the number of database queries scales linearly with the number of objects translated. AutoLex includes a separate method, `make_multiple_translations`, for use in this situation. `make_multiple_translations` retrieves translations from the database with only a single query, no matter how many translated objects it is given.

Figure 4-3 shows a more detailed explanation of the process.

Algorithm

1. Check if translations are enabled. If not, return.

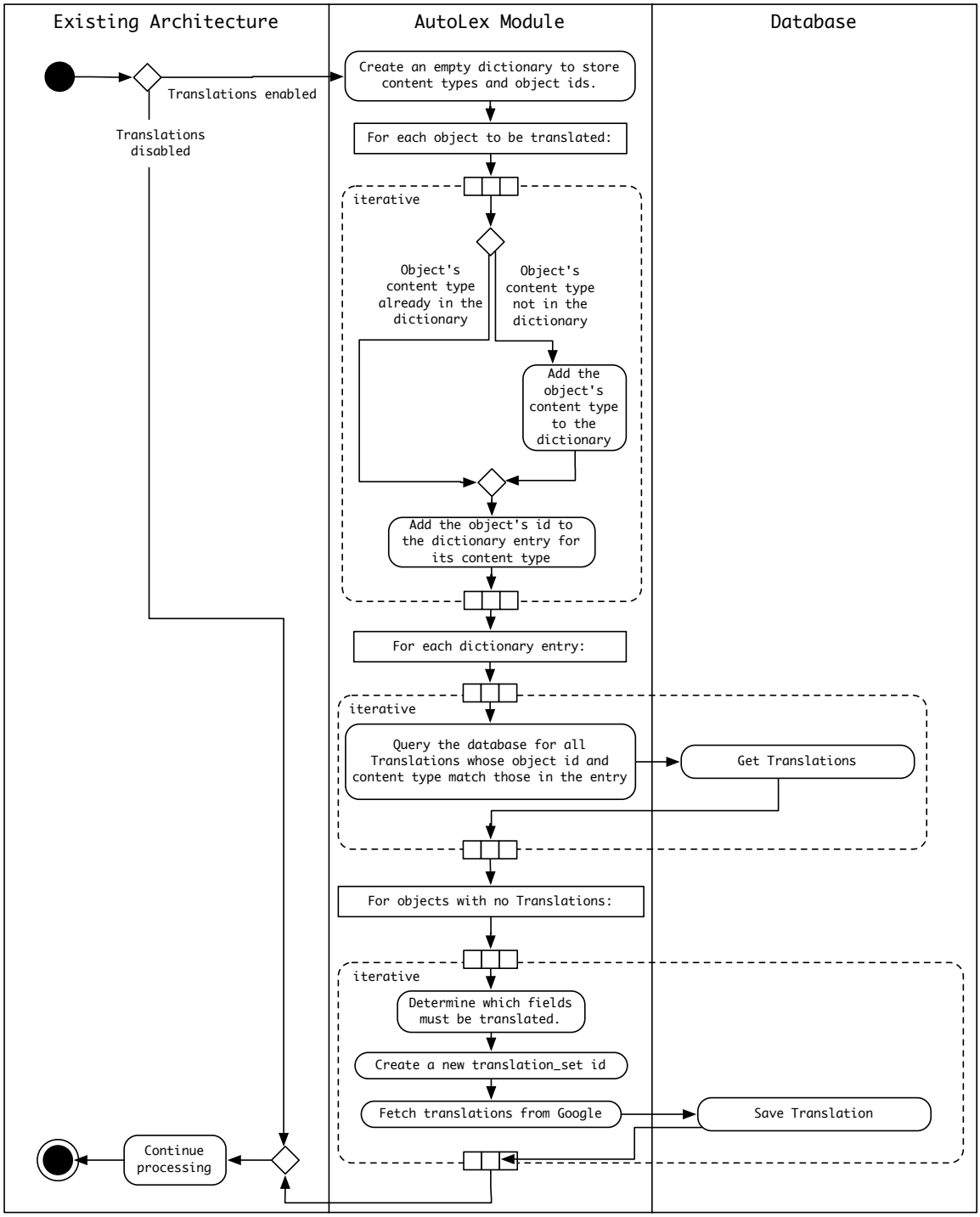

Figure 4-3: The make_multiple_translations algorithm is used to update items in a community's homepage feed. Using this algorithm instead of the simpler update_translation algorithm improves performance by getting all of the translations through one database query instead of querying for each individual object.

2. Get the user's language.

3. Get all Translations from the database whose `object_id` and `content_type` fields match those in the input dictionary.

4. For objects without Translations:

   (a) Create a new `translation_set` id. All translations created for this object at this time will have this `translation_set` id.

   (b) For each translated field, get and save a translation from Google.

4.3 Displaying Translations

Regardless of whether there are few or many translated objects on a page, translations are rendered for display one at a time through `get_translated_version`. This allows translations to be displayed without defining an additional template function, known within Django as a *template tag*.

However, it does carry performance risks, since each translation is retrieved from the database individually. As when making translations, the biggest effects are seen on pages with many items displayed. Indexes are used on the Translation table's `object_id` and `content_type` columns to decrease the speed of Translation lookups, especially within `get_translated_version`.

Algorithm

1. Check if translations are enabled. If not, return the original text.

2. Check the user's language.

3. Look in the database for a translation that corresponds with this object, language, and field.

   (a) If one is found, return it.

   (b) If no translation is found, return the original text.

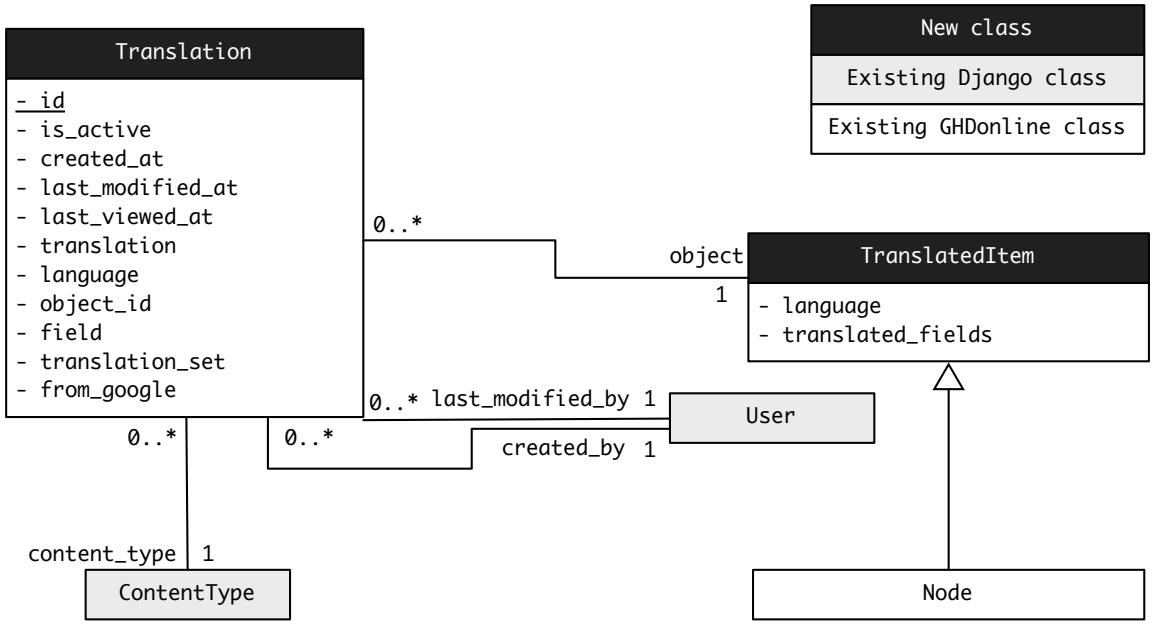

Figure 4-4: Class diagram of the AutoLex module.

4.4 Important Components of the AutoLex Application

The *TranslatedItem* and Translation classes and the `fetch_google_translation` method are also integral parts of the module. Objects that are translated must inherit from *TranslatedItem*; the translations themselves are stored as Translation objects. `fetch_google_translation` is the method that retrieves translations from Google and saves them as in the Translation database table.

4.4.1 Classes

There are two classes in the AutoLex module, *TranslatedItem* and Translation. Figure 4-4 shows AutoLex's classes and how they are integrated with GHDonline's existing classes.

TranslatedItem Classes that may be translated (for example, GHDonoline's Node) must inherit from the abstract class *TranslatedItem*, which requires the implementation of a `translated_fields` property that returns a list of fields to be translated

(for example, ['text', 'title']).

Translation The translations themselves are stored as Translation objects. The most important fields, `language` and `translation`, hold the translation's language and the translated text.

The Translation class contains a generic foreign key that specifies which object each translation belongs to. The two columns that hold this information are `object_id` and `content_type`. `object_id` is the primary key of the object that the translation belongs to. It is similar to a foreign key, but without an actual foreign key constraint on the table. `content_type` is a regular foreign key to Django's built-in ContentType table. For a detailed explanation of generic foreign keys, see 2.2.2.

A `translation_set` field connects translations of an object's different fields that are in the same language and were made at the same time.

4.4.2 Method for Fetching Translations From Google

The process of requesting translations from Google is handled by a utility method in the AutoLex module. This method, `fetch_google_translation`, breaks a text string into short chunks (to comply with Google's Terms of Service) and sends them to Google one by one for translation. If all of the chunks are translated successfully, then a Translation of the original object is created or updated in the database. Figure 4-5 shows an activity diagram of the `fetch_google_translation` method.

There are several risks with contacting the Google service: the process takes too long, and the method times out; or Google returns an error message instead of a translated string. AutoLex handles these errors in the following manner:

1. HTTP 400 (Bad Request): Google has blocked its translation service. If this happens, `fetch_google_translation` is immediately disabled site-wide and the administrators are notified so that the problem can be looked into.

2. HTTP 414 (Request Too Long): The request was over Google's character limit and has been rejected. While the original text is broken up into short pieces,

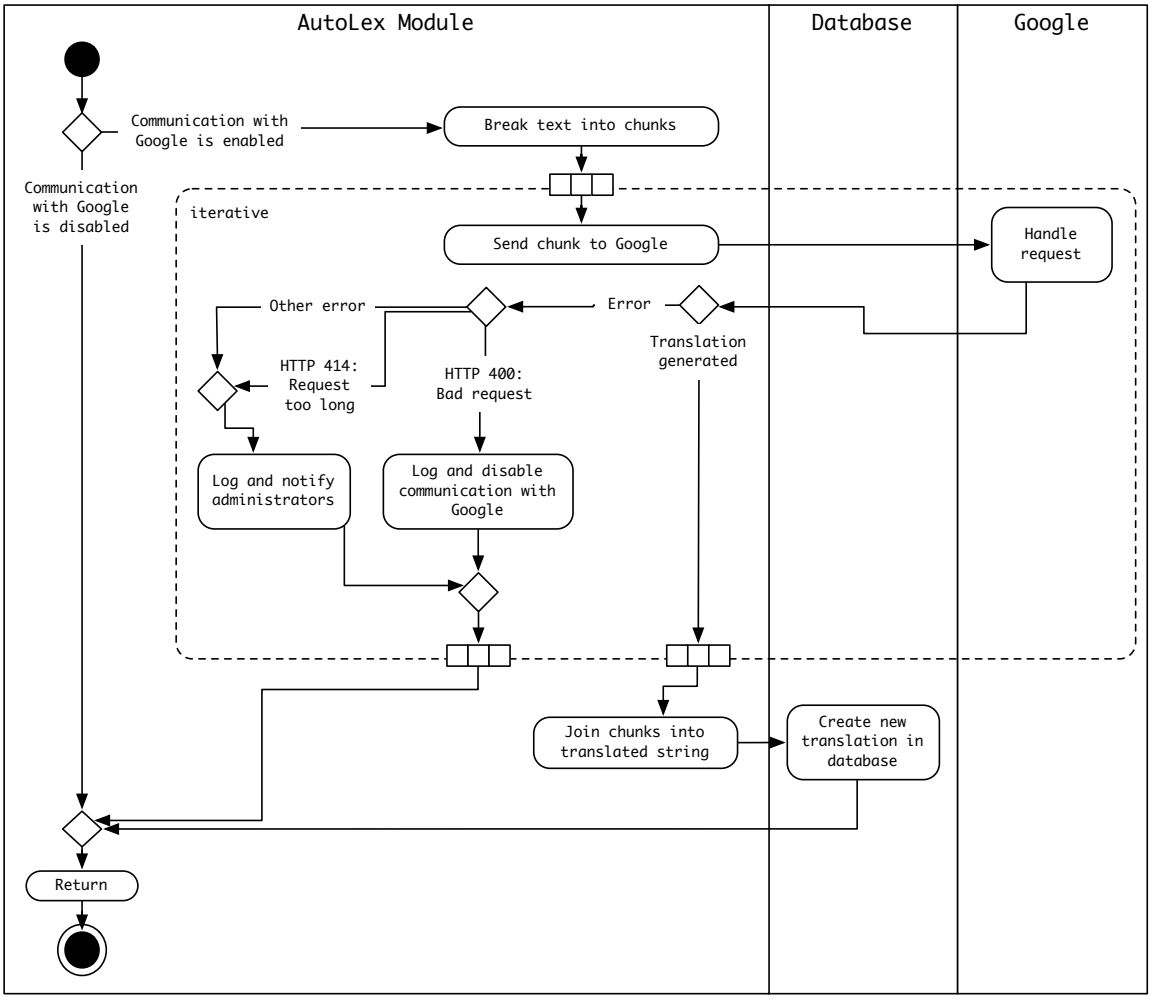

Figure 4-5: fetch_google_translation breaks a string into chunks, sends each chunk to Google for translation, and then re-joins the chunks into a translated string and saves it in the database. It handles errors by logging them and taking other appropriate measures as necessary.

this error may occur if other parts of the request are unusually long. If it does, the site administrators are notified, but `fetch_google_translation` is not disabled since the long request is likely an anomaly.

3. Other Errors: The translation is not updated and the error is logged.

The function returns without updating anything; users viewing that item will see the original version, not a translated version.

Chapter 5

Results and Evaluation

The AutoLex module satisfies all of the objectives laid out in the Introduction. It automatically generates machine translations and stores them in a database; performs well; has a clean API; and is extensible, modular, and easy to enable and disable.

5.1 Integration with Machine Translation Service

The AutoLex module is fully integrated with Google's machine translation service, providing a platform for translation into any of the 57 languages Google supports. Using AutoLex as an internationalization platform enables comprehensive, automatic localization of dynamic content.

5.2 Preservation of Acceptable Performance

There are three major variables that can affect page-loading time: the number of translations made; the number displayed on the page; and the total number of translations already in the database. Performance tests were conducted to isolate the effects of each.

The number of translations that must be created has the greatest effect on page-loading time. As can be seen in Fig. 5.2, loading time increases linearly with the number of translations that must be created. In one particularly bad scenario, the

translated version of a community homepage increases from 1.95 seconds to 8.41 seconds, approximately 331%. This corresponds to a situation in which 48 translations must be checked, fetched from Google, saved to the database, and displayed.

Fortunately, the likelihood of encountering this scenario is small. Very few pages on GHDonline display 48 different translated fields. If an earlier visitor had viewed even some of them in the current user's language, their translations would already be stored in the database and new translations would not be requested from Google. Because making new translations is the greatest source of processing delay in the AutoLex module, avoiding their creation saves significant amounts of time.

Most user requests do not require the creation of new translations, and when they do, the time required to make the new translations depends mostly on the speed of Google Translate. As a result, these results are neither an accurate measure of AutoLex's effects on page-loading times nor an entirely fair assessment of the module's algorithms. An additional set of tests, performed on community homepages whose items have been pre-translated, provides a fuller picture.

Community homepages were chosen as the reference because they are both GHDonline's slowest-loading pages and those most impacted by the addition of translations. For AutoLex to be a viable solution to GHDonline's internationalization needs, it cannot increase the average page-loading time of community homepages by more than approximately 30%. Each community controls the number of items it displays on its homepage; homepages with more items are significantly more impacted because they require generating and displaying more translations.

Two instances of this test show how AutoLex affects community homepage loading time. The first measures the loading time of a typical community homepage, with 22 translated items; the second measures that of an exceptionally large community homepage, with 58 translated items. This number was chosen to represent an upper limit to the increase in loading times caused by deploying AutoLex on GHDonline.

In addition to the number of items displayed on the page, the number of translations in the database also affects loading times, as queries to large database tables can take longer than queries to small tables. To measure these effects, each test

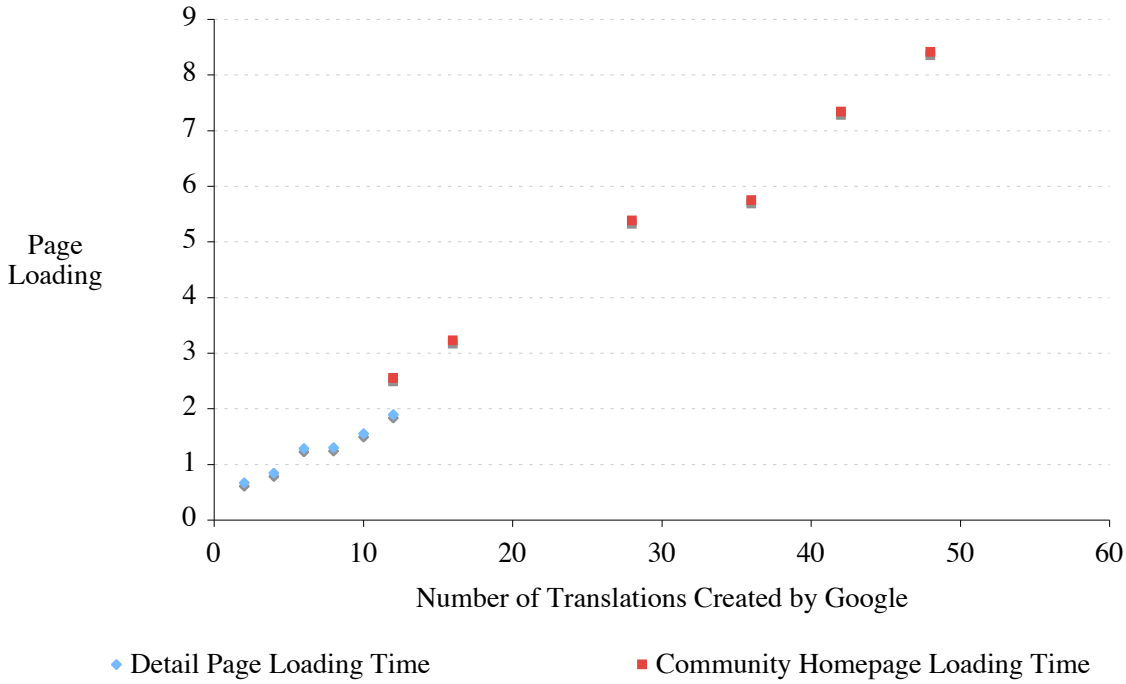

Figure 5-1: This chart shows how fetching translations from Google affects page loading time. For both the detail page (which uses make_translation) and the community homepage (which uses make_multiple_translations), loading time increases linearly with the number of translations created, due to the fact that each additional item must be sent to Google individually; there are no economies of scale. Displaying an additional translated field on the page increases loading time by approximately 0.1185 seconds on the detail page and 0.157 seconds on the community homepage, depending on the amount of text that must be translated. Although make_multiple_translations appears to be slower than make_translation, the performance drop is because GHDonline's community homepage algorithm does not scale as well as its detail page algorithm; it is not caused by the AutoLex module. Using make_translation for the community homepage would increase its loading time even more.

was conducted at five different translation table sizes. The smallest, depicted in Fig. 5.2 as "Zero Languages Supported", was performed with virtually no translations in the database, only those displayed on the page tested. The largest, "4 Languages Supported", was performed with a translation table holding four sets of translation objects for each piece of user-generated content on GHDonline. Supporting each additional language adds approximately 11,000 entries to the Translation table.

The loading time of the typical homepage increased by less than 26% in all tests, satisfying the specification laid out in the requirements.

5.3 API

With only three methods, `make_translation`, `make_multiple_translations`, and `get_translated_version`, all of which require only two input parameters, AutoLex's API is straightforward and simple.

5.4 Extensibility

Any language supported by Google Translate can be used with AutoLex, and because all translations are stored in the same database table, it is easy to add new types of objects for translation. No database schema changes are required for adding new languages, new fields, or new object types. This is an important benefit, as it would be unwieldy to modify the database each time a change occurred.

With integration into Django's built-in admin site, AutoLex can easily be extended to allow human translators to overwrite the machine-generated translations improving overall quality.

5.5 Modularity

Installing AutoLex requires making few changes to GHDonline's existing code. The largest change is in its feed generation algorithm, which is altered to construct the

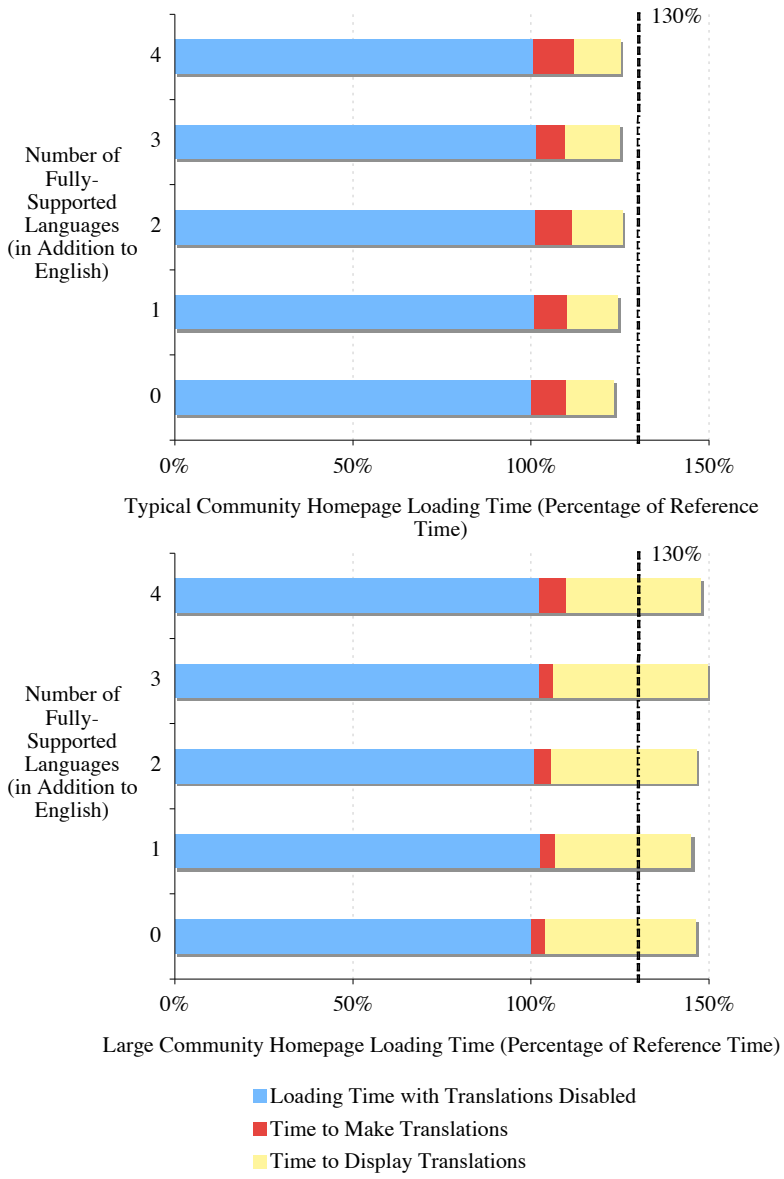

Figure 5-2: The two charts above show how the AutoLex module affects page loading times if all of the required translations are already in the database. This scenario represents the majority of page views, since each item only has to be translated once into any given language; after that, it can be accessed from the database. The top figure shows the loading time of a typical GHDonline community homepage, with 22 translated items. The bottom shows the loading time of an extremely large community homepage, with 58 translated items. Performance is acceptable; the loading time of the typical homepage increases by approximately 25%. Loading times do not degrade significantly with increasing translation table size because of the use of indexes on thranslations' object_id and content_type columns. Caching (saving the translations in memory) could be used to improve performance further; see Sec. 6.2 for more details. Tables A.1 and A.2 hold the data used to construct these charts.

object dictionary for `make_multiple_translations`. In addition, templates are updated to display translated and original versions side-by-side. Within the Community application, calls to `make_translation` and `make_multiple_translations` are added to processing routines, and calls to `get_translated_version` have been added to Node's `get_title` and `get_text` accessor methods.

5.6 Ease of Enabling and Disabling

Using the AutoLex application requires the addition of two boolean variables to a project's settings file. `ENABLE_TRANSLATE` turns the feature on and off entirely. `ENABLE_GOOGLE_TRANSLATE` can be used to control communication with Google; if it is disabled, but `ENABLE_TRANSLATE` is enabled, translations from the database will be displayed, but new translations will not be generated.

Chapter 6

Conclusions and Further Work

The AutoLex application provides a clear way forward for offering multilingual communication to GHDonline users. With its introduction, healthcare professionals around the world will be able to collaborate with others in their field regardless of what language they speak, eliminating linguistic isolation as an obstacle to improving care. This is a major success, and GHDonline will be the one of the first websites to offer this feature.

The module is a profound addition to the wider Django community as well. It is the first application to automatically translate content in Django models, greatly reducing the barriers to translating large amounts of new and changing content. It does so in a way that is clear and easily extensible without repeated database modifications, and it can be integrated into an existing website without making major modifications to its code base. As an addition to the growing ecosystem of Django translation applications, it stands out for offering functionality beyond that of any other module.

These accomplishments aside, more work remains to deploy translations on GHDonline and to improve AutoLex itself. The application described in this document is similar to an alpha release: functional, but not quite ready for widespread adoption. A pilot version will be incorporated into GHDonline in spring 2011. Below is an outline of the work that is required to ready AutoLex for widespread use.

6.1 Automatically Generate Accessors for Translated Fields

AutoLex should automatically create accessors for translated fields. A widely-used method is to append the language to the end of the field name, resulting in accessors like 'title_fr'.

This was not strictly necessary for internationalizing GHDonline's Node model, which already had accessor methods, but providing them makes installation less unwieldy and usage more uniform.

6.2 Enable Integration with Caching Backends

AutoLex should make it easy to use caching for increased performance. In this context, *caching* is the storage of database objects in memory, and *caching backends* are software that perform this purpose. This practice reduces the need for costly database calls, since the objects can be retrieved more quickly from memory.

As seen in Sec. 5.2, database calls are one of AutoLex's greatest performance costs. With caching, performance could be improved significantly, making AutoLex even more attractive. Developers using AutoLex would need to set up the caching backend themselves, but AutoLex should take steps for easy integration.

6.3 Improve Chunking Algorithm

The current version of `fetch_google_translation` creates short chunks from long strings by splitting them every 1,000 characters, without taking into account natural breaks in the text. This algorithm results in relatively low-quality translations, as the writing could be truncated in the middle of a sentence or even in the middle of a word. It should be updated to a more sophisticated version that splits chunks at logical breakpoints.

6.4 Improve Translation Display

The current version of Google Translate (version 1) does not preserve line breaks, which greatly reduces the readability of long comments and discussions. Version 2 corrects this problem, but is not yet complete. Once version 2 is ready, AutoLex should move to incorporate it.

6.5 Build a Comprehensive Test Suite

More thorough testing must be conducted before the module can be considered reliable enough for adoption.

Appendix A

Tables

Table A.1: Loading Time of a Typical Community Homepage

Reference Loading Time (s): 0.8007

Translations neither made nor displayed				
Number of Languages Supported	**Approximate Number of Translation Objects in Database**	**Loading Time (s)**	**Uncertainty (s)**	**Increase Over Reference (%)**
0	18	0.8007	0.0166	n/a
1	11,000	0.8083	0.0138	0.95%
2	22,000	0.8117	0.0128	1.37%
3	34,000	0.8140	0.0094	1.66%
4	45,000	0.8074	0.0112	0.83%

*Translations made**				
Number of Languages Supported	**Approximate Number of Translation Objects in Database**	**Loading Time (s)**	**Uncertainty (s)**	**Increase Over Reference (%)**
0	18	0.8804	0.0110	9.95%
1	11,000	0.8837	0.0105	10.36%
2	22,000	0.8931	0.0090	11.54%
3	34,000	0.8776	0.0134	9.60%
4	45,000	0.8911	0.0140	11.29%

* These data reflect only the time to run the module's `make_multiple_translations` code, not the time to retrieve translations from Google Translate

Translations made and displayed				
Number of Languages Supported	**Approximate Number of Translation Objects in Database**	**Loading Time (s)**	**Uncertainty (s)**	**Increase Over Reference (%)**
0	18	0.9864	0.0111	23.18%
1	11,000	0.9965	0.0072	24.44%
2	22,000	1.0059	0.0133	25.62%
3	34,000	0.9996	0.0145	24.84%
4	45,000	0.9946	0.0075	24.22%

Table A.2: Loading Time of a Large Community Homepage

Reference Loading Time (s): 0.8604

Translations neither made nor displayed				
Number of Languages Supported	**Approximate Number of Translation Objects in Database**	**Loading Time (s)**	**Uncertainty (s)**	**Increase Over Reference (%)**
0	18	0.8604	0.0079	n/a
1	11,000	0.8818	0.0123	2.48%
2	22,000	0.8695	0.0126	1.05%
3	34,000	0.8817	0.0081	2.47%
4	45,000	0.8805	0.0076	2.33%

*Translations made**				
Number of Languages Supported	**Approximate Number of Translation Objects in Database**	**Loading Time (s)**	**Uncertainty (s)**	**Increase Over Reference (%)**
0	18	0.8960	0.0093	4.13%
1	11,000	0.9190	0.0141	6.81%
2	22,000	0.9091	0.0106	5.65%
3	34,000	0.9137	0.0106	6.19%
4	45,000	0.9466	0.0465	10.02%

* These data reflect only the time to run the module's `make_multiple_translations` code, not the time to retrieve translations from Google Translate

Translations made and displayed				
Number of Languages Supported	**Approximate Number of Translation Objects in Database**	**Loading Time (s)**	**Uncertainty (s)**	**Increase Over Reference (%)**
0	18	1.2584	0.0226	46.25%
1	11,000	1.2480	0.0136	45.04%
2	22,000	1.2602	0.0188	46.46%
3	34,000	1.2879	0.0201	49.68%
4	45,000	1.2706	0.0129	47.67%