Easyling tech manual Documentation

Release crest-documentation

Sep 23, 2021

Contents

CHAPTER 1

Architecture

1.1 Modularity

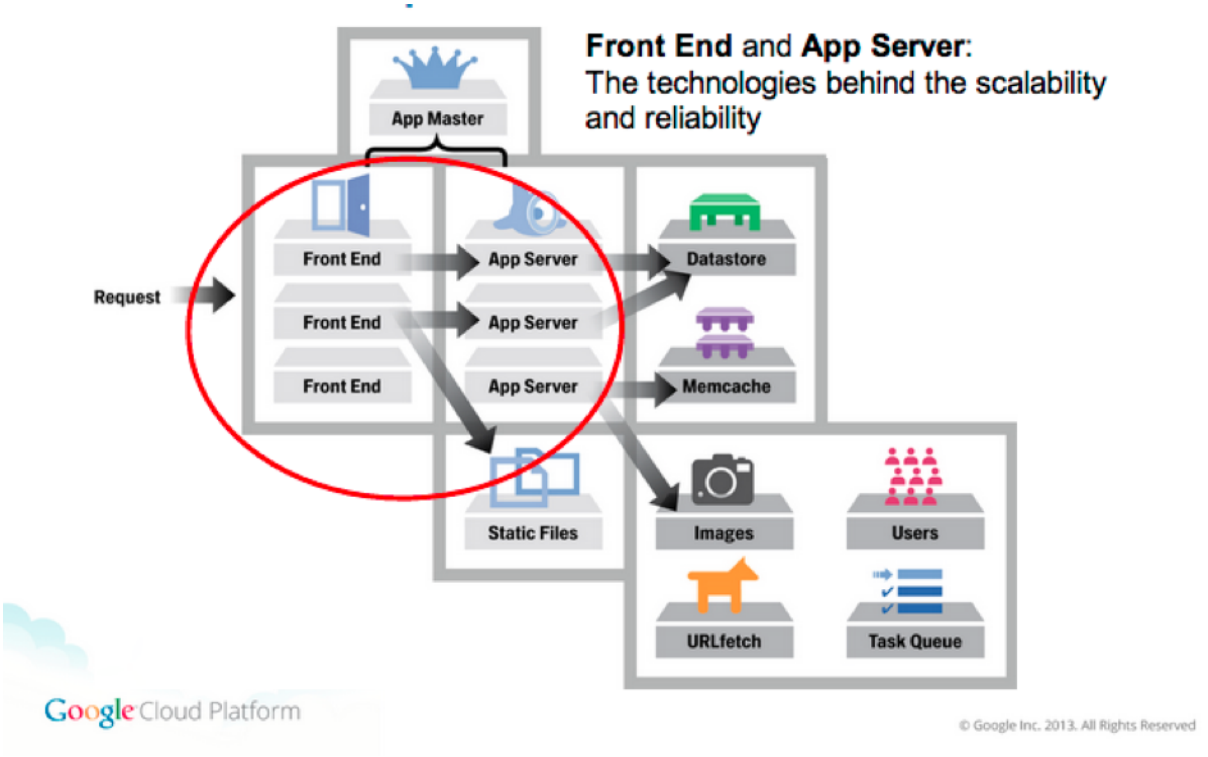
The Translation Proxy is based over Google's AppEngine infrastructure, split into frontend and backend modules. Each module encompasses variable numbers of instances, scaling automatically in response to demand. Modules are versioned and deployed separately, and can be switched independently, if needed.

Frontend instances serve requests from visitors to translated pages (in addition to serving the Dashboard and providing user-facing functionality).Requests are routed to the Proxy Application via the CNAME records created during the publishing process.

Backend modules are responsible for billing, statistics aggregation, and handling potentially long-running tasks, like XML import-export. Backend instances are not directly addressable, and provide no user-facing interaction.

1.2 Underlying technologies

Immediately underlying the Proxy Application is the AppEngine infrastructure, responsible for rapidly scaling the deployed application. AppEngine also handles communication with the Google Cloud Datastore, a high-replication NoSQL database acting as the main persistent storage; as well as the Google Cloud Storage system, an also-distributed long-term storage. Logging is provided by Google Cloud Logging, while BigQuery provides rapid search ability in the saved logs on request.



AppEngine Architecture

Encompassing the entire application is the Google EdgeCache network, proactively caching content in various data centers located regionally to the request originator. Any content bearing the appropriate headers (`Cache-control:public; max-age=/\d+/` and `Pragma:public` - both are required) is cached by the EdgeCache for as long as needed, for requests originating in the same geographic area.

The current instance of the Proxy Application is hosted in the US central region, as a multi-tenant application (serving multiple users, but dedicated to proxying). However, single-tenant deployments (dedicated to a single user), special deployments to other regions (EU or Asia), or internal systems fulfilling the AppScale system requirements can be discussed on a per-request basis.

CHAPTER 2

Request Handling

In the Translation Proxy, frontend instances are responsible for serving translated pages. Thanks to AppEngine's quick-reaction scaling, the number of frontend instances aggressively follows (and somewhat predicts) demand, keeping latency low. The general life cycle of a proxy request can be described as follows.

- The incoming requests, based on the domain name, reach the Google Cloud (rerouted via DNS record CNAME, pointing to `ghs.domainverify.net`.
- Based on the domain name and the deployed Proxy application, AppEngine decides that this specific request should be routed to the Proxy AppEngine deployment.
- The request reaches the Proxy Application internally; the application does a lookup against the domain for the associated project. There are special domain names, and the final serving domain, for which caching is activated.
- Based on the URL, the Proxy application determines the matching Page in the Proxy database. The database has a list of segments, pointing to our internal Translation Memory (TM). We retrieve all these existing Database entries, including the translations for the given target language.
- The Proxy application processes the incoming URL request, and transforms it to point back to the original site's domain. Then, the source content of the translation is sourced, according to cache settings in effect on the project.
  - If source caching is *disabled*, the application issues a request, and retrieves the result from the original web server, which is hosting the original website language.
  - If source caching is *enabled*, a local copy (a previously stored version of the source HTML) is used, instead of issuing a request to the original web server.
- Depending on the `Content-type` of the response, the appropriate `Translator` is selected, and the response is passed to an instance of the `Translator` as a document. The behavior of the `Translator` can be affected by cache settings as well.
  - If binary caching is *disabled*, the application then builds the Document Object Model (DOM) tree of the result, finally iterates through all the block level elements, and matches them against the segments loaded from the database. If there's a match, we replace the text with the translation. If not, we 'report' it as a missing translation.

  - If binary caching is *enabled* and the hash of the source HTML *matches* the one stored in the cache, a previously prepared and stored translated HTML is served.
  - If binary caching and keep cache are *both enabled*, and the hash of the source HTML *doesn't match* the one stored in the cache, the proxy translates the page using the TM. If the number of translated segments is higher than the previously prepared and stored translated HTML, the new version is served; otherwise the old one. (Keep cache can be thought of as a "poor man's staging server").

- Hyperlinks in the translated content are altered by the `LinkMapper` to point to the proxied domain instead of the original. This affects all `href` or `src` attributes in the document equally, unless the element is given the `__ptNoRemap` class. At this point, resources may be replaced by their localized counterparts on a string-replacement basis.
- The application serializes the translated DOM tree, and writes it to the response object.
- Before final transmission takes place, the Proxy may rewrite or add any additional headers, such as `Cache-control` or `Pragma`.
- Finally, the Proxy serves the document as an HTML5 stream, as a response to the original request. AppEngine *must* close the connection once the response is transmitted, so proxying streaming services is not possible in this fashion!

CHAPTER 3

Classification of Content

The Translation Proxy distinguishes two main types of content: text content and resources. The key difference is that text content may be translated, while resource content is treated opaquely, and only references can be replaced as resource localization. It is possible to reclassify entities from Resource to Text, but not the other way around.

During proxying, resources are enumerated, and any already-localized references are replaced, while text content is passed to an applicable `Translator` implementation for segmented translation.

3.1 Text content

By default, the Proxy only handles responses with `Content-Type:text/html` as translatable. To process HTML content, the source response's content is parsed into a Document, then text content is extracted from the DOM-nodes. Additionally, various attribute values are processed (without additional configuration, `title` and `alt`).

The content is then transformed into `SourceEntry` entities server-side. Each block element comprises one source entry, with a globally unique key. If segmentation is enabled on the project, the appropriate rules are loaded (either using the default segmentation or by loading the custom SRX file attached to the project), and the content is segmented accordingly, with the resulting token bounds stored in the `SourceEntry`.Along with the `SourceEntry` entities, the corresponding `TargetEntry` and `SourceEntryTarget` entities are created. `TargetEntry` entities, as the name suggests, hold the translations; `SourceEntryTarget`s act as the bridge between the two, and hold the segment status indicators for both.

The content of source entries is analyzed in the context of the project, and statistics are computed. These statistics include the amount of repeated content at different confidence levels based on the similarity of the segment - The Translation Proxy differentiates five levels of similarity:

1. 102%: Strong contextual matches: every segment in the block level element (~paragraph) is a 101% match, where all the tags are identical. These matches do not result in the creation of new `SourceEntry` entities, thus changes in one place are propagated instantly to all occurrences.
2. 101%: Contextual matches: both tags in the segment, and contexts (segments immediately before and after) match.
3. 100%: Regular matches: the segment is repeated exactly, including all tags.

4. 99%: Strong fuzzy matches: tags from the ends are stripped out, words lowercased, numbers ignored.
5. 98%: Weak fuzzy matches: all tags are stripped out (may have to be adjusted manually afterwards!), words lowercased, numbers ignored. If the Proxy cannot match the tags between the translation and the source due to excessive differences, all tags are placed at the end of the segment, requiring manual review!

These classifications are reused during memory-powered pre-translation in order to select the best applicable translation or to propagate existing translations.

3.2 Resource content

By default, any content with content types other than `text/html` are treated as a resource, and is not a candidate for translation, only replacement en bloc. This mainly includes `application/javascript`, `text/xml`, and various `image/*` content types. Every resource can be given different replacements per target language, and if required, certain resources (`application/javascript` and `text/xml`) can be made translatable after pre-configuration is done. In this case, instead of references being replaced, the appropriate `Translator` will be instantiated and the content passed to it. This can enable partial or complete translation of dynamic content transmitted as `JSON` or `XML`.

CHAPTER 4

Translation Memories

The Translation Proxy can be configured to maintain and leverage internal translation memories. These memories can contain more than one target locale allowing leveraging them for any pair of locales contained within.

As opposed to project dictionaries, translation memories are keyed to the *user* creating them, and can be assigned to any project the user has Backup Owner privileges or higher.Any project can contain an arbitrary number of memories, but one must always be designated the default: only this memory will be utilized when segments are being written; while pre-translation and suggestions are fed from *all* memories assigned to the project with applicable locale configurations.

CHAPTER 5

Using TMs

Translation memories are initialized empty, and must be first configured with locales. After the target locales are defined, the memory can be populated. There are three ways a segment can be injected into the memory:

- TMX-import: The Proxy can digest a standard `TMX` (Translation Memory eXchange) file and populate a designated memory based on its contents. The memory must be configured with at least one of the target locales of the `TMX` file. Duplicate segments are either merged (if for different locales) or discarded during import.
- Project population: The Proxy can populate the memory from the project it is currently assigned to. The memory must be configured with at least one of the *project's* target locales for this to work. If there are several locales assigned to the memory, the UI will treat them as a set, and offer the *intersection* of the memory and the project's locales as the default. This set can be further restricted by removing locales from the population task before committing it. This action is logged in the project's Audit Log.
- Individual Injection: If a memory is assigned to the project with at least one locale present on both, it will be available on the Workbench for use. Confirming one or more segments will trigger the `saveToMemory` action, injecting the segment in its current form into the memory.

Memories are used for two tasks on the UI:

- Pre-translation tasks can leverage any memories assigned to the project, provided the memory is configured with the correct locale. This applies to user-triggered Pre-translation, as well as Automatic Pre-translation triggered by new content. Only content with confidence levels above the user-configured threshold will be used, matches with lower percentages are discarded.
- The Workbench automatically leverages any memories with the appropriate locales on segment selection. Matches are displayed in the Suggestions tab of the sidebar, along with their match percentages. Additionally, all memories on the project with the applicable target segments can be queried at will by entering a search term.

5.1 Confidence levels

The Proxy differentiates five levels of similarity between individual segments/entries (see here). Memory application yields the best results between 101% and 99% - 98% matches disregard tagging, and may need manual adjustment.

However, searching below 98% is also possible, using the Google Search API, but these matches should be used with caution, as there is no guarantee regarding their accuracy due to the Search API's word stemming.

CHAPTER 6

Page modifiers

Due to the way the Proxy Application operates, it becomes fairly easy to modify the pages as they are being served. Because the datastream must pass through the proxy to have the translation embedded, the Proxy Application can insert JavaScript modifiers, modify style sheets, and even embed entire pages that do not exist on the original.

- CSS Editor: the Proxy Application can be used to insert locale-specific CSS rules into the site being served. The rules are inserted as the last element of the `head` on every page served through the proxy. The most common use of this feature is to alter the writing direction for non-Latin scripts, such as Arabic.
- JavaScript Editor: the JavaScript edited here is inserted into the `head` element of every page being served through the Proxy Application. As the last element of the `head`, it has access to any global variables added by scripts before it.
- Content Override: the Proxy Application can create a "virtual" page in the site or override an existing one with custom code. For any requests to an overridden page, the corresponding remote server request is not sent, and the override contents are used as the basis of the translation. The source is not required to be HTML, custom content-types can be entered, along with customized cache headers, and status codes (HTTP status codes are restricted to those permitted by the Java Servlet class!) - note that the 300-family of status codes requires the `Location` header to be defined as well.

Both the CSS and JavaScript injectors can use already-existing files for injection instead of copied content. The injected files must be handled by the project in some way (either by being in the project domain, or in the domain of a linked project), or be created by a content override. The order of definition for these entries matters, as they will be inserted into the document in the order they are displayed on the UI, which may cause dependency or concurrency issues!

CHAPTER 7

Client-Side Translator

7.1 Operations

7.1.1 Overview

The *Client-Side Translator*, codenamed *CREST*, is an alternative publishing mode. Instead of operating in proxy mode, the system generates a Javascript stub that needs to be referenced in the site, and it will translate the page in real time using a dictionary downloaded from the cloud service. Language choice is persisted in the browser's *Local Storage*, enabling automatic translation of any page in the site instantly on landing.

7.1.2 Setup

Content is collected and translated the same way as normal. Once publishing is needed, content is exported by selecting the *Client-side translation* file format, then publishing the latest export (or the one selected for production) from the *Previous Exports* screen and clicking the context menu.

The translation loader script can be inserted with a one-liner script element, which is available from the *Global Settings* screen of the *Publish* section in the sidebar. The website owner needs to insert this script element into pages requiring translation.Once complete, the translations can be requested by adding a query parameter to the URL, with the name `__ptLanguage` and the chosen locale as the value (for example `https://example.com/path/to/page?__ptLanguage=ja-JP`).

7.2 Integrators' Guide

7.2.1 Elements

CREST is controlled by the loader script, inserted into every page requiring translation. The script element should be inserted as high in the `head` as possible in order to begin translation at the earliest possible point.The loader script has a number of query parameters that may be used to manipulate its operation. Any number of these can be

combined to customize the loader's behavior from the default settings (existence of said defaults also means *none* of these parameters are mandatory to supply).

- `languageParameter`: This parameter can be used to change the language selector key from its default of `__ptLanguage`.
- `storageParameter`: This parameter can be used to change the LOcalSTorage key used to store a previous selection from its default of `ptLanguage`.
- `noXDefault`: if set to `true`, suppress placing an `x-default` link element in the `head` if a translated langauge is loaded. This may have SEO implications!
- `rewriteUrl`: if set to `true`, use `history.replaceState` to rewrite the URL shown to the user so that it always displays the selected language.
- `scriptUrlIsBase`: if set to `true`, the loader will search for the translator script based on its own URL. **CAUTION: This is not supported under Internet Explorer!**
- `disableSelector`: if set to `true`, the stub will not inject its own language selector in the sidebar. In this case, it is up to the website to provide links to the various language versions.

Language selection is possible via the sidebar inserted on the right by default, or custom `a` elements that manipulate the value of the `__ptLanguage` query parameter. Note that once a language is selected, the choice is persisted into the browser *Local Storage*, so further links need not be annotated with the query to maintain translation.

On selecting a language, the loader script will insert a new script element referencing the exported dictionary. This download the translations necessary for display and the translator algorithm that processes the available DOM to replace content with the available translations.The translator will also attach a `MutationObserver` to the document being displayed that allows it to react to DOM manipulation or newly-appearing elements in real time.

7.2.2 Interop

In order to provide a seamless user experience, *CREST* exposes a number of events at key points in the process that allow the containing page to react to the translation process and take action to enhance the experience. The following events are dispatched at various points:

- `crestDictionaryLoadingStart`: Dispatched when a language is selected and download of the corresponding dictionary begins. As the dictionary can be sizeable, this event can be used to display a notification to the visitor advising them that the language is about to change.
- `crestDictionaryLoadingEnd`: Dispatched on completion of the dictionary download. Firing this event means translations are available, and they will be applied to the DOM momentarily. If a notification was displayed on download start, it should be removed on this event.
- `crestDocumentTranslationStart`: Dispatched when the initial translation of the document begins. Firing this event means translations are currently being applied to the entire page, and the displayed language is about to change. In case translation takes significant time, the user may benefit from an overlay or other message notifying them of the process and that the displayed language will change soon.
- `crestDocumentTranslationEnd`: Dispatched when the initial translation of the DOM is complete, and all available content has been replaced. At this point, the page is translated to the best of *CREST*'s abilities, and if an overlay or notification was displayed, it should be removed.
- `crestMutationTranslationStart`: Dispatched when a mutation of the DOM is detected by the attached observer and translation of the new/changed elements begins. This event is unique in that it includes a payload, an array of the `MutationRecord`s that are being processed. These include information about the element name, DOM path, and other data that may be used by the page to react to changes.

- `crestMutationTranslationEnd`: Dispatched when the mutation observer completes its run and designates all mutated elements translated. If a notification was displayed on the preceding event, it should be removed now.

Example

```
<script type="application/javascript">
    document.addEventListener("crestDocumentTranslationStart", () => console.log(
↪"Document translation started"));
    document.addEventListener("crestDocumentTranslationEnd", () => console.log(
↪"Document translation ended"));
    // e.detail.targets contains the array of MutationRecord objects that are being␣
↪processed in the current run. For more information, see https://developer.mozilla.
↪org/en-US/docs/Web/API/MutationRecord
    document.addEventListener("crestMutationTranslationStart", (e) => console.
↪log(`Mutation translation started. Mutated records: ${e.detail.targets}`));
    document.addEventListener("crestMutationTranslationEnd", () => console.log(
↪"Mutation translation ended"));
    document.addEventListener("crestDictionaryLoadingStart", () => console.log(
↪"Dictionary download started"));
    document.addEventListener("crestDictionaryLoadingEnd", () => console.log(
↪"Dictionary download ended"));

    console.log("Event listeners ready...");
</script>
```