UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escuela Técnica Superior de Ingeniería Informática

Universidad Politécnica de Valencia

DESARROLLO DE UN EXTRACTOR DE INFORMACIÓN DE LA WEB PARA FIREFOX

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: ADELANTADO ROMERO, LUIS

Tutor: SILVA GALIANA, JOSEP FRANCESC

Septiembre 2018

Resumen

La extracción del contenido web comprende un conjunto de técnicas que le permiten a un programa localizar los diversos componentes de una página web y extraer aquellos que puedan ser de utilidad u ocultar los que sean innecesarios. Hay diversas herramientas *software* que permiten hacer esto, como por ejemplo los bloqueadores de publicidad que le ocultan al usuario todos los mensajes publicitarios que le puedan resultar molestos.

En la tienda de aplicaciones de *Firefox* hay varios *add-ons* que permiten realizar estas tareas. Lo que difiere de unas herramientas dedicadas a la extracción del contenido web a otras es, fundamentalmente, la manera de extraer el contenido. Hay herramientas que analizan el documento *HTML* en busca de patrones (*HTML parsing)* e incluso herramientas que intentan hacer uso de *machine learning* para interpretar una página web como lo hace un ser humano. En el caso de este proyecto trabajaremos con herramientas que analizan la estructura interna de una página accediendo al árbol *DOM.*

A finales del año 2017 la fundación Mozilla lanzó una nueva versión del navegador que cambiaba completamente la manera de desarrollar estas extensiones. El objetivo de este proyecto consiste en la migración a esta nueva arquitectura de tres herramientas diferentes que extraen el contenido principal, el menú y la plantilla de una página web.

Palabras clave: página web, Firefox, plugin, HTML, JavaScript, árbol DOM, extracción de contenido

Abstract

The extraction of web content includes a set of techniques that allow a program to locate the components of a web page and extract those that may be useful or hide those that are unnecessary. There are several software tools that allow the user to do it, such as advertising blockers that hide all advertising messages that may be annoying to the user.

In the Firefox application store there are several add-ons that allow you to perform these tasks. What differs from some tools dedicated to the extraction of web content to others is, fundamentally, the way to extract the content. There are tools that analyze the HTML document in search of patterns (HTML parsing) and even tools that try to use machine learning to interpret a web page as a human being does. In the case of this project we will work with tools that analyze the internal structure of a page by accessing the DOM tree.

At the end of 2017 the Mozilla Foundation launched a new version of the browser that completely changed the way to develop these extensions. The objective of this project consists of the migration to this new architecture of three different tools that extract the main content, the menu and the template of a web page.

Keywords: web page, Firefox, plugin, HTML, JavaScript, tree DOM, content extraction

Índice

Índice de figuras

1. Introducción

Los *addons* [1] son pequeños programas que se instalan en un navegador con el objetivo de ampliar sus funcionalidades. Un *addon* nos permite hacer una amplia variedad de cosas, por ejemplo podemos cambiar el aspecto de una página web, modificar la interfaz o añadirle características nuevas al navegador. Se desarrollan utilizando las tecnologías estándar del desarrollo web; es decir, *JavaScript, HTML* y *CSS*. Hasta el año 2017 si se quería crear una extensión para Firefox existían diversas herramientas y posibilidades al alcance de cualquier desarrollador. Una de las más utilizadas la tecnología *XUL/XPCOM* [2], con la cual se desarrollaron las extensiones que vamos a migrar. El lenguaje *XUL* es un dialecto de *XML* desarrollado por la fundación Mozilla que se utilizaba para el desarrollo de interfaces. Estas interfaces se desarrollaban de manera similar a como se desarrollan las páginas web. Por otro lado *XPCOM* es un entorno de desarrollo de Mozilla que ofrece diversos servicios: gestión de la memoria, una interfaz para el paso de mensajes y abstracción de archivos.

Imagen 1: AdBlock es una de las extensiones de navegador más populares.

Todo esto cambió cuando estas arquitecturas son sustituidas por *Webextensions* [3]. Este cambio de arquitectura se fundamenta en varias razones: compatibilidad con las arquitecturas del resto de navegadores del mercado, mayor seguridad puesto que sus capacidades son más limitadas y mayor simplicidad a la hora de desarrollar. La principal motivación de este proyecto es migrar tres extensiones realizadas con la antigua arquitectura *XUL/XPCOM* a la más reciente *Webextensions*, así como ampliar y mejorar sus funcionalidades. A un nivel muy elemental, ambas arquitecturas comparten muchos elementos, pero también hay una serie importante de diferencias. Empezamos hablando de sus similitudes.

Ambas arquitecturas incluyen las siguiente características:

- Archivos *manifest* que definen los metadatos de la extensión tales como el lenguaje por defecto, el nombre y la descripción. También se definen algunos de los aspectos de su funcionamiento.

- Código *JavaScript* que permite el acceso a un conjunto de APIs privilegiadas que permanecen cargadas mientras la extensión permanezca habilitada.

- La capacidad de añadir a la interfaz del navegador botones y diversos elementos.

Las diferencias más importantes entre ambas son las siguientes:

- A diferencia de las extensiones *XUL/XPCOM*, en *Webextensions* las interfaces son mucho más limitadas y el acceso a las APIs de *JavaScript* es mucho más restringido.

- En *Webextensions* tan solo se puede acceder al contenido de la web inyectando *scripts* individuales en cada una de las páginas web. Estos *scripts* se comunican mediante una API de paso de mensajes.

A continuación, vamos a desarrollar un poco más estas similitudes y diferencias.

Archivos *manifest*

Las extensiones *XUL/XPCOM* utilizan dos archivos *manifest; install.rdf* y *chrome.manifest.* El primero de estos archivos incluye los metadatos que le permiten al Firefox identificar a los *add-ons* y obtener información durante el proceso de instalación. Incluye información sobre los desarrolladores y sobre las versiones con las que son compatibles. El segundo archivo le da la información al navegador sobre dónde se encuentran los componentes esenciales de la extensión, como por ejemplo en qué directorio se encuentran los archivos que definen la interfaz así como dónde se encuentran los *strings* localizados en los diversos idiomas. Estos archivos *manifest* desaparecen en *Webextensions* y en su lugar son sustituidos por un único archivo *JSON* llamado *manifest.json* [4]. La función de este es idéntica la de los dos anteriores.

Interfaz de usuario

En la antigua arquitectura las interfaces de usuario se construían directamente mediante el lenguaje *XUL* que ya de por sí Firefox utilizaba para especificar su propia interfaz. En el caso de las extensiones *XUL/XPCOM* con las que trabajamos nosotros se podían utilizar *overlays* escritos en *CSS*.

La extensiones desarrolladas con *Webextensions* no obtienen un acceso tan directo a los elementos del navegador. Utilizan una combinación de elementos definidos en el *manifest.json* y de APIs en *JavaScript* que pueden modificar un conjunto limitado de componentes de la interfaz. Los únicos componentes disponibles aparecen en la siguiente tabla.

Nombre	Descripción	Especificado
Browser action	Botón en la barra de tareas	*Manifest:* browser_action API: browserAction
Page action	Botón en la barra de las URL	*Manifest:* page_action pageAction: API
Commands	Atajos del teclado	commands en *manifest* y en la API
Context menu	Añade elementos y submenús al menú contextual del navegador	API: contextMenu

Las APIs de *JavaScript*

Ambas arquitecturas contienen *scripts* que permanecen cargados tanto tiempo como la extensión permanezca habilitada. Con *XUL/XPCOM* se tiene acceso a un conjunto bastante grande de APIs y además acceso directo a los componentes internos del propio navegador. En *Webextensions* el equivalente a estas funciones se consigue mediante los *background-scripts* [5]. Estos *scripts* permiten el acceso a un conjunto de librerías utilizando la directiva *browser,* pero como ya hemos comentado estas librerías quedan muy reducidas en *Webextensions*. De más de 200 accesibles desde *XUL/XPCOM* nos quedamos con un total de 43, lo cual significa que no siempre vamos a tener una API que sustituya a la que usábamos en nuestras extensiones *legacy.*

Interacción con el contenido web

Con *XUL/XPCOM* las extensiones podían acceder al contenido web de manera directa. Con una simple directiva *gBrowser* se podía acceder al árbol *DOM* de la página y manipularlo. Esto, sin embargo, solamente era posible hacerlo de manera secuencial. Es una diferencia sustancial con Webextensions, puesto que esta nueva arquitectura trabaja por defecto de manera concurrente. Todo el código que interactúa con el contenido de la web se escribe en *scripts* separados llamados *content-scripts* [6], que se comunican entre ellos o con los *background-scripts* utilizando la API de mensajería.

Localización

En *XUL/XPCOM* los *strings* en los distintos idiomas se definen en documentos *DTD* [7] por cada uno de ellos. Cada uno de estos documentos se guarda en un directorio propio dentro del directorio *locale.* Para posteriormente acceder a estas cadenas de texto se declaran unas instancias en el *chrome.manifest.* En *Webextensions* la aproximación es similar, pero hay algunas diferencias. Por cada idioma se define un directorio dentro de *_locales,* pero en lugar de utilizar documentos *DTD* se utilizan documentos JSON. Para acceder los *strings* definidos en estos documentos haremos uso de la API *i18n.*

Una diferencia importante y que debemos destacar entre ambas arquitecturas en cuanto a la localización se refiere es que *Webextensions* no ofrece ninguna librería para traducir *strings* que puedan aparecer en *HTML*. Por tanto, si el desarrollador decide diseñar un menú para la extensión en *HTML* tendrá que utilizar JavaScript para poder localizar estas cadenas y sustituirlas por otras.

Ajustes y directorios

En *XUL/XPCOM* las extensiones guardan los ajustes utilizando las interfaces de *XPCOM* o bien utilizando los archivos *options.xul.* En *Webextensions* estas opciones se escriben en un documento HTML que incluye un *script* mediante el cual se tiene acceso a todas las APIs. Entre otras se dispone de una API *storage* en la que se almacenarán todos los ajustes. Para poder utilizar este *HTML* será necesario que su *URL* sea declarada como la clave de *options_ui* en el *manifest.json.* Es importante destacar que no es posible desde *Webextensions* acceder a las preferencias del propio navegador como sí que se podía hacer antiguamente. La organización de los directorios en ambas arquitecturas es bastante diferente. Como ya se ha comentado anteriormente en *Webextensions* se trabaja con dos tipos diferentes de *scripts: content-scripts* y *background-scripts.* Por tanto, necesitaremos dos directorios diferentes para colocar dentro de ellos los *scripts* que correspondan. Necesitaremos también un directorio *_locales* para la localización de las cadenas de texto que vayamos a utilizar y por último un directorio específico para guardar los iconos que se mostrarán en la barra de tareas.

Webextensions y los estándares de la web

De momento *Webextensions* no es un estándar [8]. Actualmente *Mozilla* está trabajando con *Opera*, *Microsoft* y el consorcio *W3C* para definir una *API* que funcione en diversos navegadores, pero no se ha definido aun como un estándar. Hay una especificación preliminar que coincide con la que tiene implementada *Google*, *Opera* y *Microsoft* y por eso las extensiones desarrolladas con *Webextensions* para Firefox pueden funcionar en *Google Chrome*, *Opera* y *Microsoft Edge* de la misma manera. A pesar de esto aun existen algunas diferencias entre el funcionamiento de las extensiones entre todos estos navegadores y hay que tenerlas en cuenta a la hora de desarrollar.

Pese a que no se haya definido este estándar, *Webextensions* presenta una *API* bastante estable. En nuestro caso al trabajar con *Firefox* la documentación la encontraremos en *MDN* (*Mozilla Development Network*). No todos los navegadores van a implementar *Webextensions* de la misma manera, sin embargo, tener una *API* común entre la mayor parte de los navegadores existentes en el mercado presenta una serie de ventajas que debemos destacar:

- Si los desarrolladores utilizan una *API* común van a poder migrar sus extensiones de un navegador a otro sin apenas realizar modificaciones en su código.

- *Webextensions* incorpora una serie de tecnologías que son bastante útiles e interesantes para el desarrollo web, como por ejemplo el soporte que ofrece al bloqueo de contenido. Característica muy utilizada tanto por extensiones que bloquean la publicidad como por aquellas que buscan ofrecerle al usuario algo más de privacidad en las búsquedas que realiza en la web.

- Actualmente *Google Chrome* tiene una alta cuota de mercado y ofrecer una plataforma conjunta permite que muchas de las extensiones desarrolladas para el navegador de *Google* se puedan migrar también en la tienda de *Firefox* ampliando así la cuota de mercado en ambos sentidos.

- Se mejoran diversos aspectos relacionados con la seguridad al limitar el acceso a recursos como la interfaz del propio navegador.

Seguridad

Las versiones *legacy* de *Firefox* eran muy poco restrictivas en comparación con *Webextensions*, no obstante esto conllevaba una serie de problemas que trataremos en este apartado [9]. La diferencia más significativa es que al comienzo la firma digital de las extensiones no era un requisito para que se pudiera publicar en la tienda oficial, por lo tanto publicar *addons* maliciosos a la página oficial de *Mozilla* no era demasiado complicado. A partir de *Firefox* 41 todas las extensiones tienen que ser validadas y firmadas por *Mozilla* para poder ser publicadas. La librería oficial de extensiones se encuentra en addon.mozilla.org (*AMO*). El proceso de evaluación de las extensiones *legacy* era bastante largo puesto que todo el código se revisaba manualmente en busca de alguna potencial amenaza. En *Webextensions* el proceso de revisión ha sido modificado y se requiere que las extensiones pasen por una serie de pruebas automatizadas en un entorno específicamente preparado para esto. Después de superar esta prueba un revisor hará una revisión manual de la extensión.

En cuanto al modelo de seguridad interno ambas arquitecturas utilizan un mecanismo conocido como *sandbox.* Este proceso consiste en aislar procesos del resto del sistema operativo y que se ejecuten de manera controlada evitando que accedan a otra información que no es necesaria para su funcionamiento. *Firefox* tiene un proceso padre y una serie de procesos hijo que descienden de él. Este proceso padre tiene acceso a los recursos del propio sistema operativo, pero los hijos se encuentran aislados en estos pequeños entornos llamados *sandbox.* Cada uno de estos entornos tiene su propio conjunto de permisos que limitan su acceso a componentes del navegador, a otras *APIs* o a otros procesos.

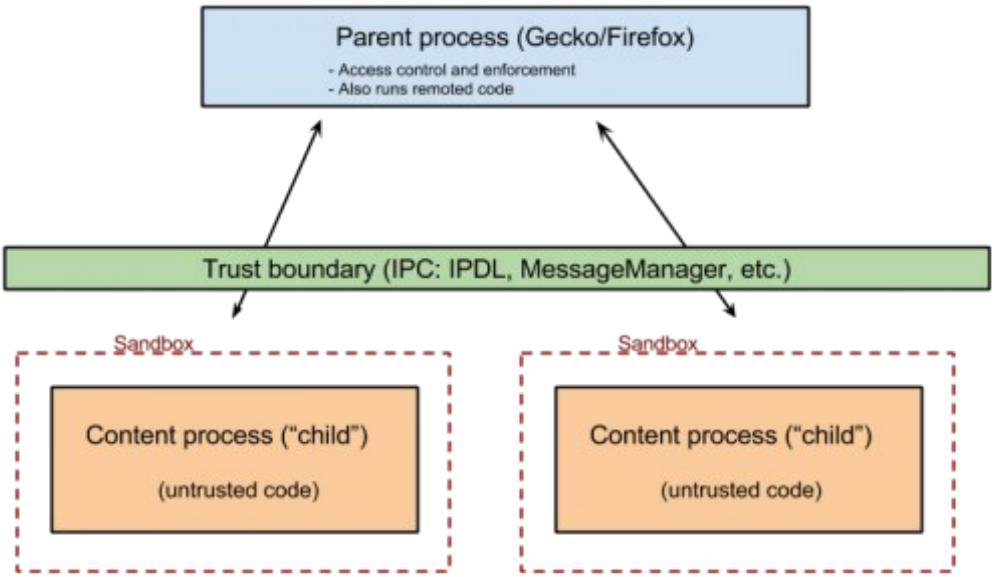

Imagen 2: Modelo de seguridad en Firefox.

2. Objetivo del proyecto

El principal objetivo de este proyecto es la migración de tres extensiones de *Firefox* realizadas con la antigua arquitectura *XUL/XPCOM* a la nueva *Webextensions*. Cada una de estas extensiones tiene un propósito diferente, pero el objetivo del desarrollo de las tres es extraer aquel contenido que pueda ser relevante de una página web. A continuación, describimos brevemente cada una de las tres extensiones y mostramos ejemplos de su funcionamiento con la web https://linuxmint.com/

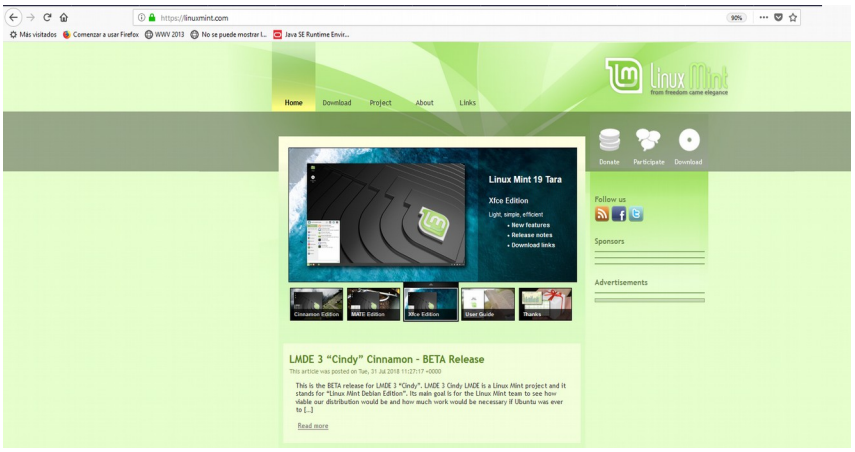

Imagen 3: La página con el aspecto inicial.

1.- MenEx (Menu Extractor): Localiza y extrae el menú de una página.

El menú es uno de los componentes esenciales en cualquier página web. Le permite al usuario conocer las distintas secciones que la componen así como navegar entre ellas.

La mayoría de estas páginas se escriben utilizando el lenguaje de marcado *HTML*. Es un lenguaje compuesto por etiquetas que describe la estructura de un sitio web. Estos documentos los recibe el navegador y con ellos construye la página que visualiza el usuario. Utilizando el modelo *DOM* (*Document Object Model*), una página web puede verse como un conjunto de nodos estructurados en forma de árbol. Por tanto, el menú de una página es un conjunto de nodos que forman parte de este árbol. Con *MenEx* tratamos de descubrir qué nodos pueden ser candidatos a formar parte del menú y nos quedamos con aquellos que tiene una probabilidad más alta de serlo y se le devuelve este resultado al usuario.

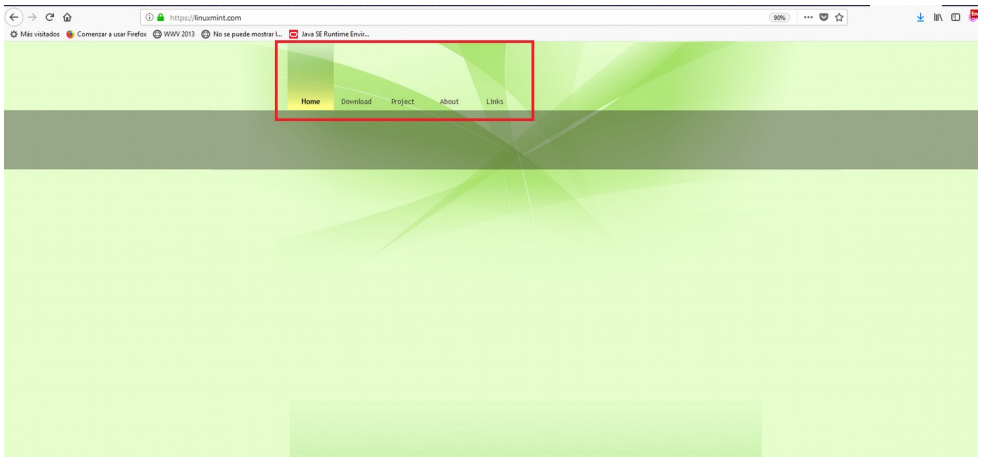

Imagen 4: Página con el menú principal extraído.

2.- TemEx (Template Extractor): Localiza y extrae la plantilla de un sitio web.

Una plantilla es un documento *HTML* con una estructura definida así como una serie de componentes visuales que están preparados para que se inserte contenido. Esta estructura es común para todas las páginas que componen el sitio web. Son importantes porque los diseñadores y los desarrolladores web las utilizan para crear sitios web que comparten una temática o incluso un estilo. Agilizan el trabajo y facilitan la navegación al usuario puesto que un diseño uniforme y con unos colores adecuados le permite al usuario distinguir bien unas opciones de otras y encontrar rápidamente la información que se busca.

El algoritmo que emplea esta extensión también se basa en el análisis de los nodos del árbol *DOM*. En este caso la extensión busca un conjunto de páginas candidatas a contener la plantilla y a continuación analiza sus respectivos árboles. Se buscan aquellas partes comunes entre estos árboles, ya que muy probablemente se traten de nodos que forman parte de la plantilla. Una vez se encuentran estos nodos se le devuelven al usuario.

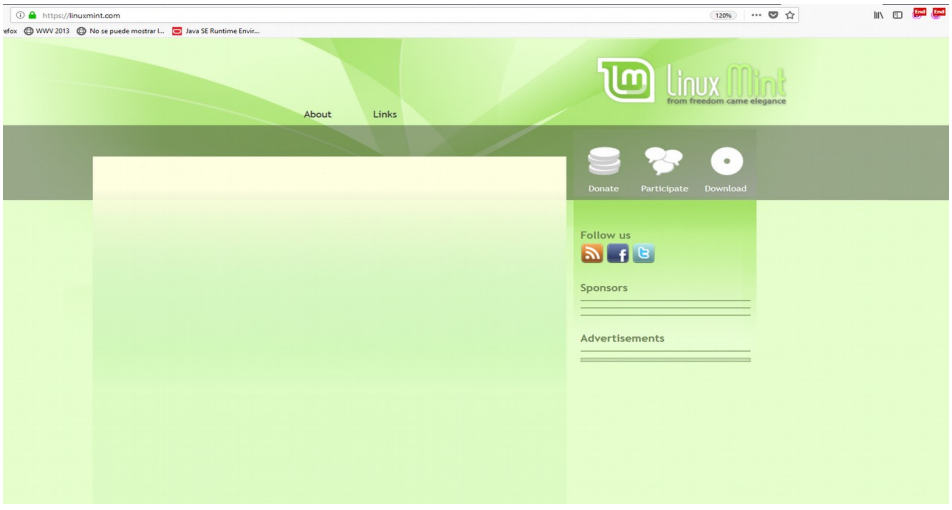

Imagen 5: Plantilla extraída con TemEx.

3. ConEx (Content Extractor): Extrae el contenido principal de un sitio web.

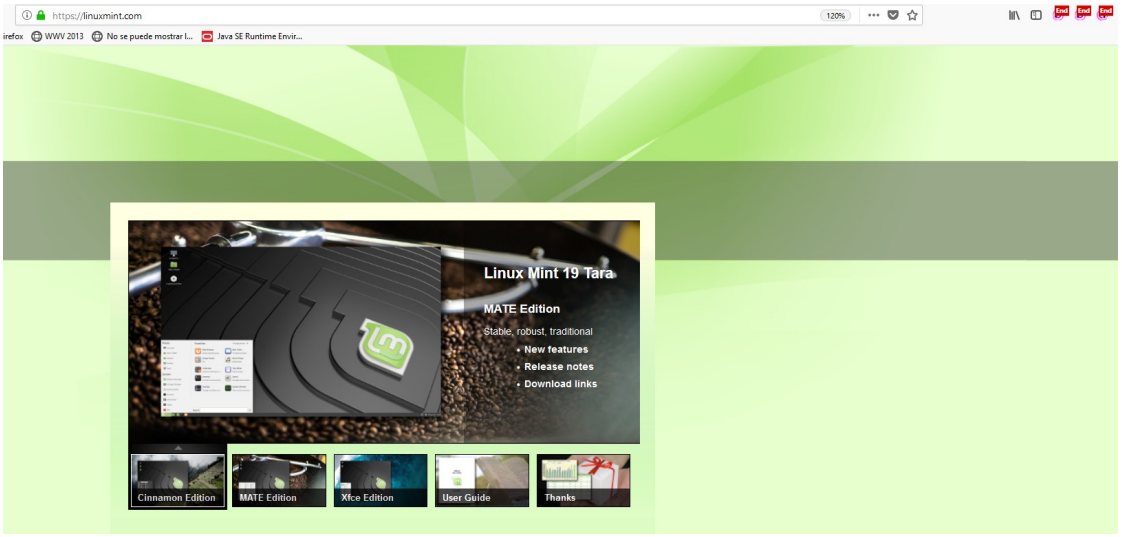

Imagen 6: Contenido principal extraído con ConEx.

El contenido principal es la información más importante de una página web. Es la información que los usuarios buscan cuando acceden a la página. Por tanto, es importante que se fácilmente visible y que el resto de elementos de la página molesten lo menos posible. La localización de este contenido resulta fundamental para el funcionamiento de extensiones que buscan eliminar los *banners* publicitarios (*AdBlocks*) y cualquier elemento visual que dificulte la navegación por la página. Con esta extensión nos vamos a centrar en aquellas páginas realizadas puramente con *HTML*. Como ya hemos comentado con anterioridad en las dos extensiones anteriores, podemos ver una página web como una colección de nodos *DOM*. Por tanto, el contenido principal de una página será un subconjunto de estos nodos. Las páginas web que se pueden encontrar a día de hoy navegando por Internet presentan estructuras muy variadas. Muchas veces aunque dos páginas se hayan realizado con la misma tecnología tienen una estructura interna completamente diferente. Esto hace que la localización del contenido principal de una página sea una tarea complicada y en ocasiones costosa.

Para obtener el contenido principal de un sitio web dado, *ConEx* selecciona un conjunto de páginas que pertenecen al mismo sitio web y una página que será la página clave de la cual queremos obtener su contenido. De este conjunto se examinan sus respectivos árboles *DOM* y se comparan con el árbol de la página clave buscando aquella parte del árbol de la página clave que no aparece en el resto de árboles del conjunto. Este conjunto de nodos que obtenemos de la realización del análisis forman parte del contenido principal de la página.

Para la realización de las migraciones de este proyecto se han seguido las siguientes fases:

- Especificación de requisitos.

- Fase de análisis y pruebas.

- Fase de diseño.

- Fase de implementación.

- Fase de testeo.

- Propuesta e implementación de mejoras.

En la **especificación de requisitos** se describe el comportamiento general que tiene que tener un *addon* en el navegador *Firefox*. Se definen tanto los requisitos funcionales como los complementarios.

En la fase de **análisis y pruebas** se analiza el funcionamiento y el comportamiento general de las extensiones *legacy* en las versiones antiguas de *Firefox*. Para ello se utilizará el depurador incluído en *Firefox* y se añadirán funciones en el código que ayudarán a trazar el recorrido del código y a detectar errores. Se averigua cómo se implementó la antigua interfaz y cómo se interactuaba con el usuario. Obtenidos estos resultados se estudiará cómo funciona Webextensions con la finalidad de obtener la misma funcionalidad. De las diversas maneras que puedan haber para implementar esta funcionalidad se buscará implementar la más ágil posible.

En la **fase de diseño** se van a tomar decisiones en función del diseño anterior. Se analiza la interfaz antigua y se realizan los cambios necesarios en los iconos para poder utilizarlos en *Webextensions*. También se busca mejorar el diseño anterior e implementar algunas mejoras visuales que puedan facilitarle el uso y la comprensión del funcionamiento de la extensión a los usuarios.

La **fase de implementación** es una de las más extensas de todas. Se desarrollan tanto los *background-scripts* como los *content-scripts* así como las comunicaciones necesarias entre los dos. Se implementan las interfaces y su localización. Esta fase requiere se un nivel importante de depuración, ya que cada una de estas extensiones tiene una cantidad importante de código que requiere de la interacción con el contenido interno de cada página web. Como ya se ha comentado en la introducción, uno de los objetivos de Mozilla al cambiar a esta arquitectura es aumentar el nivel de seguridad del navegador limitando lo que las extensiones puedan realizar, así que hay funciones que no tienen ninguna homóloga en *Webextensions* y esto implica que en varias ocasiones habrá que encontrar maneras alternativas de llevar a cabo nuestro desarrollo.

En la **fase de testeo** se comprueba el funcionamiento de las nuevas extensiones y se busca solucionar aquellos problemas que puedan surgir con el uso habitual que podría darle un usuario. Se busca también ver si hay alguna diferencia de funcionamiento con el mismo banco de páginas web, tanto como en los resultados que devuelven como en el rendimiento. En vista de estos resultados se podrán proponer mejoras en la última fase.

Para finalizar en la fase de **propuesta e implementación de mejoras** se analizan todos los resultados obtenidos y se proponen posibles mejoras. Especialmente en el campo de la optimización ya que ayuda mucho a la valoración de le extensión que el tiempo de ejecución sea el menor posible.

3. Especificación de requisitos

3.1 Introducción

En un proyecto de *software* la especificación de requisitos es el proceso en el que se determinan las necesidades del producto que se quiere desarrollar. En esta fase estas necesidades han de ser debidamente detectadas y documentadas.

3.2 Objetivos de la *ERS*

- Incluir información coherente relacionada con las necesidades reales del usuario que hay que satisfacer. Es por esto que es importante que el cliente participe activamente en esta fase.
- Ayudar a los desarrolladores a entender qué es exactamente lo que está pidiendo el cliente.
- Servir de ayuda para el desarrollo de estándares *ERS* de otras organizaciones.

3.3 Propósito de la *ERS*

En este apartado se especifican aquellos requisitos que deben cumplir las extensiones con el fin de que sean publicadas en la tienda oficial de *Mozilla Firefox*. Se desarrollan aquellas condiciones necesarias para el correcto funcionamiento de estas extensiones en el navegador.

De manera general los desarrolladores son libres de mantener y ofrecer soporte a sus extensiones como ellos elijan. Sin embargo, con el fin de que la extensión no suponga ningún perjuicio para los usuarios y puedan ser analizados por Mozilla de manera apropiada, hay una serie de criterios que todas las extensiones tienen que cumplir.

- Las extensiones han de pedir solamente aquellos permisos necesarios para su funcionamiento.
- Todo el código necesario para el funcionamiento de la extensión tiene que estar incluido en el paquete y no ser cargado desde otra fuente externa.
- Si la extensión va a enviar información personal es necesario que lo haga a través de canales cifrados y seguros.
- Es preciso evitar archivos repetidos así como innecesarios.
- El código debe estar escrito de manera clara y legible puesto que hay una parte del proceso de publicación en la que se hace una revisión del código manual.
- Ninguna extensión debe afectar de manera negativa al rendimiento del navegador.
- Tan solo se permite el uso de librerías y/o *frameworks* de terceros sin modificar.

3.4 Ámbito de la *ERS* en este proyecto

El objetivo principal del proyecto es migrar tres extensiones *legacy* a la nueva arquitectura *Webextensions* para que funcionen con las nuevas versiones de *Firefox Quantum* (a partir de la versión 47.0). Sin embargo, dado que hay varios navegadores en el mercado que implementan especificaciones similares de *Webextensions* se intentará también que funcionen también en otras plataformas como *Google Chrome* y *Opera*.

3.5 Requisitos funcionales

Aquí se especificarán cuáles son las funciones que deben realizar los tres complementos.

3.5.1 *MenEx*

- **Extracción del menú:** permitirá a los usuarios extraer el menú de la página web en la que se encuentran en ese momento.
- **Posibilidad de recuperar la página original:** una vez que el menú se ha extraído, el complemento debe de ser capaz de recuperar el aspecto original de la página.
- **Posibilidad de volver a visualizar el menú extraído:** si el usuario ha decidido recuperar el aspecto original de la página, el complemento le ofrecerá la posibilidad de volver a visualizar el menú extraído sin necesidad de que el algoritmo de extracción se vuelve a ejecutar.

3.5.2 *TemEx*

- **Extracción de la plantilla:** permitirá extraer la plantilla de la página web en la que el usuario se encuentra en ese momento.
- **Posibilidad de recuperar la página original:** el usuario podrá recuperar el aspecto original de la página.
- **Posibilidad de volver a visualizar la plantilla extraída:** se podrá recuperar la plantilla previamente extraída sin necesidad de volver a ejecutar todo el proceso de extracción.

3.5.3 *ConEx*

- **Extracción del contenido principal:** permitirá localizar y mostrar el contenido principal de la página en la que se encuentra el usuario.
- **Posibilidad de recuperar la página original:** el usuario podrá recuperar el aspecto original de la página.
- **Posibilidad de visualizar el contenido extraído:** se podrá recuperar el contenido previamente extraído

Las tres extensiones mostrarán la interfaz en el idioma por defecto que esté el navegador siempre que dicho idioma sea uno de los cuatro disponibles en el complemento.

3.6 Requisitos de eficiencia

Al tratarse de una arquitectura nueva, *Mozilla* no ha especificado unos requisitos mínimos de eficiencia, por tanto en este caso los requisitos los pondrán los usuarios. Se ha decidido marcar como máximo 10 segundos ya que es el tiempo medio máximo que el usuario aguanta sin cambiar de página de acuerdo con las 10 normas de usabilidad web de Nielsen [10]. No se espera que esta media se cumpla en todas las páginas por el momento, especialmente con el complemento *ConEx* ya que hace operaciones bastante costosas, no obstante se va a intentar que se acerque lo máximo posible.

Una vez extraído el contenido cambiar de la vista de la página inicial al contenido extraído debe ser un proceso instantáneo y no se espera que en ningún caso supere los dos segundos de tiempo.

4. Análisis del sistema

4.1 Estructura y funcionamiento de una extensión XUL/XPCOM

Una extensión *XUL/XPCOM* se desarrolla utilizando distintos tipos de tecnologías cada una de ellas con una funcionalidad bien diferenciada.

Imagen 7: Las tecnologías que componen una extensión.

XPCOM (Cross Platform Component Object Model): Es un entorno de desarrollo que proporciona una serie de servicios fundamentales para el desarrollo multiplataforma. Por ejemplo, ofrece gestión de la memoria y de los archivos, hilos, estructuras de datos básicas (*strings, arrays*) e interfaces para el paso de mensajes. Como bien aparece indicado en la *Imagen 4, XPCOM* es el cerebro operativo de la extensión; es decir, se encarga del funcionamiento interno del *addon.*

XUL (XML User Interface Language): Es un lenguaje de marcado (dialecto de *XML*) desarrollado por Mozilla que se emplea para el desarrollo de las interfaces de usuario [11]. Su sintaxis es muy similar a la empleada en los lenguajes destinados al desarrollo web, como por ejemplo *HTML*. Una interfaz *XUL* se compone de tres directorios: content, skin y locale.

- Content: contiene los documentos *XUL* donde se define la disposición de los elementos de la interfaz.

- Skin: contiene los archivos *CSS* y las imágenes que definirán la apariencia externa de los elementos que componen la interfaz.

- Locale: contiene los strings en diversos idiomas con el fin de que el texto se muestre de acuerdo al idioma de *Firefox*.

JavaScript: es un lenguaje de programación de alto nivel débilmente tipado que se emplea para dotar de funcionalidad a la extensión. Es uno de las tecnologías más usadas a día de hoy en Internet para una multitud de servicios.

CSS (Cascading Style Sheets): es un lenguaje de estilos que se emplea para establecer el aspecto visual de la extensión. Al igual que *JavaScript*, *CSS* se trata de una tecnología ampliamente utilizada en el mundo de la web junto con *HTML*.

4.2 Funcionamiento de una extensión XUL/XPCOM

4.2.1 XUL

XUL es un lenguaje que a diferencia de *HTML* se concibió desde el principio como un lenguaje de marcado orientado al diseño de interfaces de usuario. Esto implica que es posible insertar elementos de la interfaz simplemente añadiendo etiquetas sin necesidad de incluir ningún tipo de *script.* En este apartado vamos a explicar cómo funciona *XUL*, cuáles son sus mecanismos para crear una interfaz y dotarla de funcionalidad.

4.2.1.1 Añadir elementos a la barra de tareas

4.2.1.2 *Event handlers* y funcionalidades

Para que el botón que hemos implementado en el punto anterior tenga funcionalidad debemos añadirle un *event handler* que entre en funcionamiento cuando el usuario lo pulse. Se define en un archivo JavaScript de la siguiente manera:

```
<!-- Overlay MenEx button -->
<toolbarbutton id="MenEx-MenuExtractorButton"
  label="&MenEx.ExtractMenuButton.label;"
  accesskey="&MenEx.ExtractMenuButton.accesskey;"
  tooltiptext="&MenEx.ExtractMenuButton.tooltiptext;"
  oncommand="MenEx.BrowserOverlay.execute();"
  class="ExtractMenuButton toolbarbutton-1 chromeclass-toolbar-additional" />
</overlay>
```

Imagen 8: Un botón definido en XUL.

A estos *handlers* se les pueden pasar eventos como argumento. Existen por lo menos treinta tipos eventos en *XUL* que pueden ser tratados de diferentes maneras. En este caso el evento es que el usuario ha accionado un botón en la barra de tareas, pero también podría ser que el usuario haya movido el ratón por encima de un elemento de la interfaz o que haya abierto una nueva pestaña en el navegador. Para manejar este sistema *XUL* utiliza un mecanismo que genera un objeto *event* cada vez que detecta que se ha generado uno, por tanto se crea un objeto por cada evento. En cada uno de estos objetos se establecen una serie de propiedades que cambiarán según el tipo de acción que haya ejecutado el usuario. En el caso del movimiento del ratón, por ejemplo, las propiedades serían la posición y qué tecla se ha accionado. Estos eventos son procesados por *XUL* en tres fases.

- Capturing phase: el evento es enviado directamente a la ventana. Una vez ha sido recibido será enviado al al propio documento para luego ser reenviado a

todos los antepasados del elemento XUL en el que se produjo el evento hasta alcanzarlo. Es un recorrrido descendente.

- Target phase: el evento es enviado directamente al elemento XUL.

- Bubbling phase: el evento se le envía a cada elemento de manera ascendente hasta que vuelve a la ventana de nuevo.

Por ejemplo, cuando el ratón se mueve por encima de un botón colocado dentro de un elemento *box,* se genera un evento *mousemove.* Este evento se manda primero a la ventana, después al documento y por último al elemento. Con esto se completa la primera fase. En la segunda fase el evento es enviado directamente al botón y por último se pasa a la última fase en la cual se sigue el mismo procedimiento que en la primera pero en orden inverso; es decir, se enviará el evento al elemento *box,* posteriormente al documento y por último de vuelta a la ventana.

Hay dos maneras de añadirle un *listener* a un elemento. Se le puede pasar un *script* como atributo que contenga el *listener* o llamando al método *addEventListener* del propio elemento. La primera forma es la más común al ser la más simple de escribir.

4.2.1.3 *Skin* y localización

Un archivo *CSS* contiene información sobre el estilo de los elementos que conforman un documento web. En *Firefox* el funcionamiento es similar, pero en lugar de aplicarse a un documento web *HTML* se aplicará a un documento *XUL*. En este archivo se determinarán qué fuentes se utilizarán, qué colores y tamaños tendrán los elementos así como los bordes. *XUL* aplica una hoja *CSS* por defecto, pero en muchos casos esto no le será suficiente al desarrollador por lo que será recomendable sustituir este documento por uno propio. Una *skin* en *Firefox* es una colección de estos documentos de estilo y de imágenes que se aplican a un archivo *XUL*.

La mayoría del *software* se diseña con la intención de que la traducción de las interfaces sea lo más sencilla posible. Por lo general se escriben tablas con todos los *strings* necesarios, una por cada idioma, y cada parte del código en la que se necesita acceder a las cadenas de texto se referencia a la tabla del idioma que corresponda. *Firefox* emplea un mecanismo similar con entidades. Por ejemplo, para que el usuario vea una etiqueta explicando la funcionalidad de un botón cuando coloca el puntero encima, es necesario definir un atributo *label* en el elemento *button* como se puede apreciar en la *imagen 7.*

```
<button label="&findLabel;"/>
```

Imagen 9: Entidad como atributo.

En este caso el texto que aparecerá sobre el botón cuando se sitúe el cursor sobre él será el del valor que tenga la entidad *&findLabel*. Este valor dependerá del idioma del navegador. Estas entidades se almacenan en archivos *DTD* (*Document Type Definition*) que se guardan

en el directorio *locales*. Se crea una carpeta por cada idioma y los más normal es definir un archivo *DTD* por cada *XUL*.

4.2.2 XPCOM

Ya hemos visto cómo se puede definir una interfaz utilizando la tecnología XUL y cómo esta puede interactuar con los usuarios mediante el uso de *listeners* en JavaScript. Sin embargo, hay determinadas acciones que no vamos a poder implementar en estas extensiones tan solo con JavaScript. Para poder utilizar algunos servicios como la sincronización de hilos, el uso de funciones de red (*HTTP* y *FTP*) no son posibles desde JavaScript, necesitamos poder ejecutar código nativo para realizar algunas de estas funciones. Por este motivo disponemos de *XPCOM*, que es un modelo de objeto y componentes desarrollado por Mozilla.

Mozilla Firefox es una colección de componentes, cada uno de los cuales cumple una función bien definida. Hay un componente por cada menú, por cada botón y en general por cada elemento del navegador. Estos componentes se construyen a partir de una serie de interfaces. Una interfaz representa un conjunto de funcionalidades que pueden ser implementadas por los componentes. Por tanto, cada componente tiene una funcionalidad definida por una interfaz.

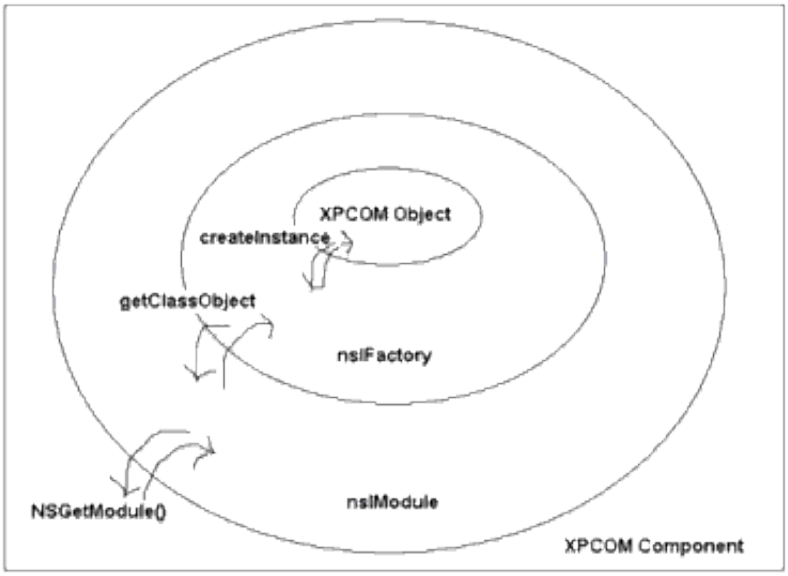

Imagen 10: Esquema de un componente XPCOM.

4.2.2.1 La creación, registro e implementación de componentes *XPCOM*

```
var wmdata = Components.classes["@mozilla.org/rdf/datasource;1?name=window-mediator"].getService();
wmdata.QueryInterface(Components.interfaces.nsIWindowDataSource);
```

Imagen 11: Creación de un componente.

XPCOM se ha desarrollado pensando en la máxima simplificación posible, por tanto el uso de componentes e interfaces no es complicado. Por ejemplo, si queremos hacer un seguimiento de las ventanas abiertas en el navegador podemos utilizar el *window mediator component.* Este componente utiliza la interfaz *nsIWindowDataSource*.

El navegador mantiene un registro en el que almacena todos los componentes que se van a utilizar. Esto significa que cada vez que desarrollemos un nuevo componente hay que darlo de alta en el registro. Una manera explícita de hacerlo es con el programa *regxpcom.* Para utilizar este programa basta con copiar el componente al directorio *components* del cliente y desde la línea de comandos en ese mismo directorio ejecutar *regxpcom.*

4.2.3 *JavaScript*

Para implementar la funcionalidad de nuestra extensión XUL/XPCOM necesitamos utilizar *JavaScript.* Disponemos de una amplia colección de *APIs* que podemos utilizar, como por ejemplo las de XPCOM. Disponemos de librerías para consultar y modificar bases de datos como *mozIStorageRow* o de mecanismos como el objeto *gBrowser* para acceder al contenido interno de una página web.

4.2.4 *CSS*

El componente más externo de una extensión XUL/XPCOM. Tiene la misma funcionalidad que en una aplicación web; es decir, añadirle un estilo visual.

En este caso el estilo se le añade a los elementos de XUL. De igual manera que en un documento HTML, el documento CSS puede estar almacenado en cualquier lugar, incluso se puede acceder de manera remota desde XUL, pero lo más habitual es colocar el archivo XUL y CSS en el mismo directorio. Cualquier elemento de XUL puede obtener un estilo con CSS. Para seleccionar el elemento que quieres estilizar puedes emplear los selectores. A continuación, se muestran algunos ejemplos que se pueden utilizar:

- *button:* Afectará a todos los elementos button.
- #special-button: Afectará a todos los elementos que tengan el identificadort special-button.
- .bigbuttons: Afectará a todos los elementos cuyo atributo class sea bigbuttons.

- toolbar > button: Afectará a todos los botones que se encuentren en la barra de herramientas.

Estos selectores se pueden combinar de la manera que se quiera, pero hay que tener cuidado y ser lo más precioso y claro posible a la hora de declararlos porque fácilmente pueden llevar a confusión.

4.3 Pruebas de los addon *legacy*

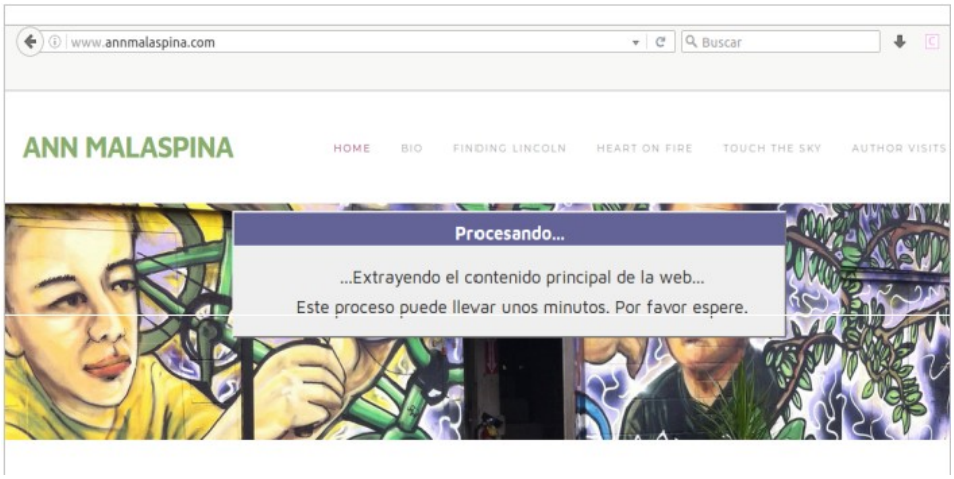

Imagen 12: Versión legacy de ConEx en funcionamiento.

Una vez aclarado cómo funciona una extensión empleando la antigua arquitectura pasamos a analizar su comportamiento para saber cómo conseguir una funcionalidad similar en Webextensions. Para ello necesitamos obtener una versión anterior a las primeras versiones de *Firefox Quantum*. Mozilla tiene un directorio propio en el que se almacenan todas las versiones de *Firefox* para su descarga. El código de los *add-ons* era común comprimirlo en archivos de formato *XPI* [12]. Tan solo es un formato de compresión, si cambiamos la extensión por *ZIP* podemos examinar el contenido con cualquier compresor. Para instalar cualquier extensión tan solo tendremos que arrastrar el archivo *XPI* dentro de la ventana del propio navegador.

Cada una de las tres extensiones presenta un icono en la barra de tareas con un logo que la identifica. Cada vez que el usuario acciona uno de estos botones la extensión muestra un mensaje insertado en la propia página web que le informa de que la extracción de la información se ha iniciado. Este mensaje saldrá en un idioma diferente de acuerdo con el idioma que tenga configurado el usuario en su navegador. Una vez ha concluído la extracción el mensaje desaparecerá y el icono de la barra de tareas se sustituirá por una flecha de dirección. Si volvemos a pulsarlo nos devolverá a la página inicial y lo podremos volver a pulsar para volver de nuevo al contenido extraído. Por tanto, será necesario que en la carpeta *icons* de las extensiones tengamos varios iconos para cada una de las situaciones. Necesitamos un icono para la visualización de la extensión en la barra de tareas, otro para que aparezca cuando la extracción ha terminado y le permita al usuario volver a la página original, así como uno para que el usuario pueda volver a ver el contenido extraído. Las imágenes se encuentran almacenadas en un único *PNG* y mediante un archivo *CSS* le indicamos al navegador qué icono escoger según la acción que haya realizado el usuario.

Imagen 13: Los iconos de TemEx.

Los tiempos de ejecución varían mucho entre las tres extensiones y dependen bastante de el tamaño de estas y especialmente de cómo estén implementadas. Por lo general la extracción del contenido principal de una página web es una operación bastante más costosa que extraer el menú, ya que MenEx trabaja únicamente con una única página mientras que las otras dos extensiones requieren más información para poder desarrollar su cometido.

5. Diseño e implementación de las extensiones en *Webextensions*

El objetivo de este apartado es diseñar una interfaz en función de cómo estaba diseñada en la anterior arquitectura y añadir mejoras visuales. La implementación de los siguientes apartados es idéntica para las tres extensiones.

5.1 Interfaz de usuario

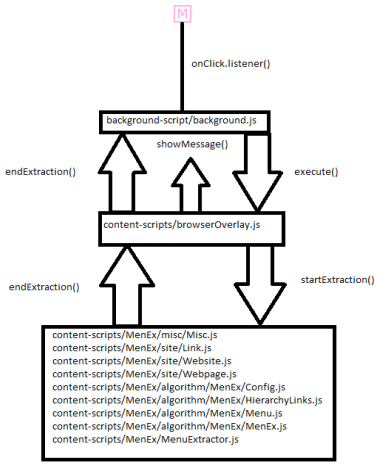

Imagen 14: Esquema de funcionamiento de las tres extensiones.

Como ya hemos comentado anteriormente antes se empleaba el lenguaje *XUL* de Mozilla para el desarrollo de las interfaces de las extensiones. En *Webextensions* se emplean un conjunto de *APIs* más limitadas. Para interactuar con el usuario disponemos de varias herramientas: añadir iconos a la barra de tareas, añadir iconos a la barra del buscador o abrir menús contextuales. En nuestro caso nos interesa que las tres extensiones tengan un icono visible en la barra de tareas.

Disponemos de la carpeta *icons* para almacenar los icons que empleará la extensión. En lugar de utilizar una única imagen *PNG* en este caso hemos separado cada icono en un *PNG* distinto para poder llamar a cada uno individualmente desde el código. Por tanto, necesitaremos cuatro imágenes *PNG* para cada una de las extensiones. Una vez colocadas en el directorio apropiado, podemos definir ya el icono en la propia *toolbar*. Para ello primero tenemos que especificarlo en el *manifest.json* utilizando la clave *browser_action* como se muestra a continuación:

```
//Add MenEx button to Firefox's toolbar
"browser_action": {
  "default_icon": "icons/icons-48.png",
  "default_title": "__MSG_extensionDescription__"
},
```

Imagen 15: Añadimos un icono a la barra de tareas.

Con la etiqueta *default_icon* definimos la ruta en la que se encuentra el icono de la extensión que se va a cargar en la barra de tareas. Con la etiqueta *default_title* cargamos el mensaje que queremos que se le muestre al usuario cuando sitúe el cursor encima del icono. En este caso cargamos una variable cuyo valor dependerá del idioma del navegador del usuario. El navegador buscará la etiqueta *__MSG_extensionDescription__* en el archivo *messages.json* en la carpeta *_locales* del idioma que corresponda.

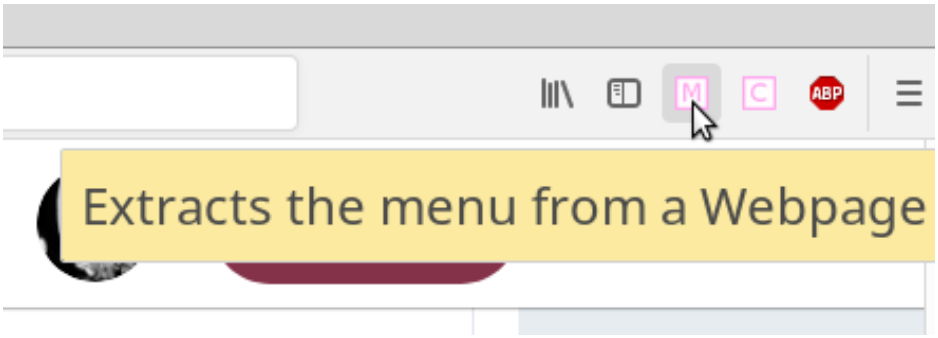

Imagen 16: Icono MenEx.

Podemos definir dos tipos de acciones con la etiqueta *browser_action:* utilizando un *popup* o utilizando un *listener*. En nuestro caso no nos interesa tener ningún menú asociado a nuestra extensión, por tanto utilizaremos directamente un *listener* que se disparará en el momento en el que el usuario accione el botón. Este *listener* necesitamos que se encuentre activo en todo momento, por lo tanto tenemos que declararlo en un *background script* de la siguiente manera:

```
browser.browserAction.onClicked.addListener((tab) => {
  alert("Work");
}
```

Imagen 17: Listener en el background-script.

Como ya hemos visto a la hora de ejecutar la versión *legacy* de la extensión, antes de comenzar la ejecución del algoritmo de extracción el navegador le muestra un mensaje al usuario para informarle de que la extracción ha comenzado y de que tardará un tiempo en completar la operación. Este mensaje se insertaba dentro de la propia página creando un * * en el que se introducía el texto. En la nueva versión se ha decidido cambiar este mensaje por una notificación al usuario mediante la *API Notification.* Estas notificaciones no las crea el propio navegador, sino que dependen del sistema operativo. Por tanto, el aspecto visual de las mismas va a depender de la versión del sistema operativo que tenga cada usuario.

Las notificaciones las vamos a definir también dentro del *background script.* Para poder crear una necesitamos diversos parámetros:

- type: Existen 4 tipos diferentes de notificaciones; *basic, image, list* y *progress.* Según el tipo de notificación que se eliga tendremos una opciones u otras. En nuestro caso no necesitamos insertar ninguna imagen ni tampoco ninguna lista, por tanto escogeremos el tipo *basic.*

- message: El contenido principal que queremos que el usuario lea al desplegar la notificación.

- title: El título de la notificación.

- iconURL: Una URL que apuntará al icono que queremos que se muestre. Como con cualquier otro icono deberá estar almacenado en el directorio *icons.*

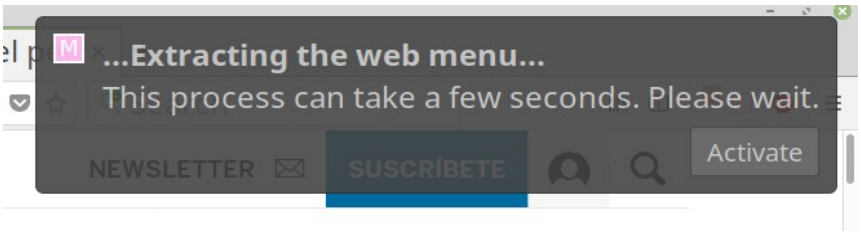

Imagen 18: Notificación que se le muestra al usuario al comenzar la extracción.

Estas notificaciones son visibles durante un breve periodo de tiempo dependiendo del sistema operativo. Si no han desaparecido cuando la extensión ha finalizado su tarea desde el *background-script* la eliminaremos directamente. Lo más común es que la notificación desaparezca antes de que la extensión termine la extracción, por tanto utilizaremos otro mecanismo para indicarle al usuario que la ejecución aun no ha terminado. Para poder hacer esto tenemos diversas alternativas; que se abra una ventana mostrando un mensaje, o colocar un texto en el propio icono de la barra de tareas. Hemos elegido la segunda opción ya que es menos intrusiva con el usuario. Estas etiquetas pueden llevar el texto que le pasemos por parámetro a la función *setBadgeText(),* pero por lo general es recomendable que estos textos sean lo más breves posible, por tanto como se puede apreciar en la *Imagen 17* mostrará el texto "*Ex*" cuando la extensión se esté ejecutando y "*End*" cuando haya terminado.

Imagen 19: Iconos de las extensiones durante y después de la extracción.

En la imagen anterior podemos apreciar la información que verá el usuario mientras la extensión está funcionando y después de que haya terminado. Como se puede ver en la imagen de la izquierda el icono aparece transparente, esto se debe a que bloqueamos el icono cuando el usuario lo acciona, de tal manera que no se pueda volver a utilizar mientras la extensión esté trabajando. Para conseguirlo hacemos uso de la instrucción *browserAction.disable(id)* a la que le pasamos por parámetro el identificador de la pestaña donde se encuentra activa la extensión. Además de esto si el usuario coloca el cursor encima del icono durante el proceso de extracción se le mostrará un mensaje indicándole que en esos momentos la extensión está funcionando.

5.2 Comunicación entre los *background* y los *content scripts*

Para poder realizar la extracción del contenido de una web necesitamos tener acceso al árbol *DOM* interno que la representa. Los *background scripts* se cargan y permanecen así en segundo plano hasta que la extensión es desactivada. En nuestro caso el único *background script* que tenemos en las tres extensiones carga una serie de *listeners* que utilizaremos para interactuar con la interfaz del usuario. Pero este tipo de *scripts* no tienen acceso al contenido interno, por tanto necesitaremos enviar un mensaje a un *content script* que inicie las tareas de extracción [13]. Cuando el icono de la barra de tareas se pulsa se activa un *listener* que activará una función encargada de realizar las siguientes funciones:

- Se carga el mensaje que el usuario verá si coloca el ratón encima del icono.

- Se carga el mensaje “Ex” en la parte superior derecha del icono para indicarle al usuario que la extracción ha comenzado.

- Se llama a una función que le enviará un mensaje a al content script browserOverlay y en ese momento comenzará la extracción.

```
function messagePassing() {
  browser.tabs.query({currentWindow: true, active: true})
  .then(sendMessageToTabs)
  .catch(onError);
}
```

Imagen 20: Función que manda un mensaje del background script al content script.

- Se desactiva el icono para que no se pueda volver a utilizar mientras la extracción está funcionando.

Para poder enviar un mensaje utilizamos una función asíncrona que nos devolverá una promesa. En *JavaScript* las promesas son mecanismos que representan el valor futuro de una operación asíncrona [14][15]. Cuando se devuelve una promesa hay dos posibles respuestas; o se acepta o se rechaza. En el caso de que no se produzca ningún error se enviará el mensaje al *content script,* si se produjera algún error se activaría la función *onError* de la cláusula *catch.*

En la *Imagen 20* podemos observar el código encargado de de recibir el mensaje del *background script.* Cuando lo reciba se llamará automáticamente a la función *ConEx.BrowserOverlay.execute();* que iniciará el proceso de extracción en este caso del contenido principal de la página.

5.3 Extracción del contenido

```
browser.runtime.onMessage.addListener(function(request, sender, sendResponse) {
    ConEx.BrowserOverlay.execute();
});
```

Imagen 21: Código receptor del mensaje en el content script.

Para realizar la extracción del contenido, las tres extensiones hacen uso de un archivo *JavaScript* llamado *browserOverlay* [16], que es el mismo que recibía el mensaje que daba lugar al inicio de la extracción en el apartado anterior. Este archivo pondrá en marcha la extracción y se encargará de comunicarse con el *background script* para indicarle que le muestre un mensaje al usuario o directamente para comunicarle que la extracción ha terminado. Este archivo es idéntico en las tres extensiones, pero hay diferencias bastante sustanciales con respecto a cómo estaba implementado en la anterior arquitectura.

5.3.1 *browserOverlay.js*

```
window.addEventListener("load", function load()
{
    window.removeEventListener("load", load, false);
    TeMex.BrowserOverlay.init();
}, false);
```

Imagen 22: Listener que se activa al cargar el navegador llamando a init() .

En las versiones *legacy* este documento estaba ligado con un documento *XUL* donde estaba definida la interfaz, pero en *Webextensions* no necesitamos el archivo *XUL*, por lo que este archivo se reduce de manera significativa ya que muchas de las funciones que implementaba originalmente hacían referencia a la implementación y funcionalidad de la interfaz de usuario. Todas estas funciones ya las hemos implementado en *Webextensions* mediante el *manifest.json* y el *background script,* por tanto no necesitamos funciones como *installButton, init* y *getTemplateExtractorButton* puesto que cuando este archivo se ponga en funcionamiento todos los elementos del navegador ya estarán cargados y en funcionamiento en la barra de tareas. También tenemos un *listener* que se activa cada vez que se inicia el navegador con la extensión cargada. Este *listener* llamará a la función *init* que comprobará cada 100ms si el icono de la extensión se ha pulsado. Esto ya no es necesario en *Webextensions* ya que como hemos explicado previamente las interacciones del usuario con la interfaz las tratamos con un *listener.*

El código para mostrar el mensaje ya no lo colocamos en este archivo, lo colocamos directamente en el *background.js* y utilizaremos de nuevo el mecanismo de paso de mensajes para que el navegador lo muestre en el momento que queramos. Por tanto, el nuevo código de la función *showProcessingMessage()* será el siguiente:

```
showProcessingMessage : function()
{
    browser.runtime.sendMessage({content: "message"});
},
```

Imagen 23: Paso de mensajes para mostrar el mensaje al usuario.

Como a lo largo de la ejecución vamos a mostrar diversos tipos de mensajes desde el *background script* definiremos una función que mostrará uno u otro en función del parámetro que le pasemos.

```
function treatMessage(message) {

  if(message.content == "message") {
    var header = browser.i18n.getMessage("messageTitle");
    var content1 = browser.i18n.getMessage("bodyMessage");

    browser.notifications.create(notification, {
      "type": "basic",
      "iconUrl": browser.extension.getURL("icons/c2.png"),
      "title": header,
      "message": content1
    });
  }
  else if(message.content == "endExtraction") {
    endExtraction();
  }
  else if(message.content == "error") {
    error = true;
  browser.notifications.clear(notification);
    endExtraction();
  }
}
```

Imagen 24: Función que tratará los mensaje que reciba el background script.

Se puede ver en la imagen anterior que en el caso de que se produzca un error se interrumpirá en ese momento la extracción del contenido. Esto se comentará más adelante, pero los errores más comunes que se pueden llegar a producir durante la extracción tienen que ver por lo general con la configuración de seguridad de los sitios web. Esta función se pondrá en marcha cuando se active el *listener* que se encuentra siempre activo en segundo plano a la espera de cualquier mensaje que provenga de un *content script.* Si el parámetro que recibe la función contiene el *string endExtraction,* se llamará a una función que volverá a dejar a la extensión en el estado inicial.

```
toggleView : function(contentExtractor)
{
    contentExtractor.toggleView(ConEx.BrowserOverlay.toggledView)
},

toggledView : function()
{
    browser.runtime.sendMessage({content: "endExtraction"});
}
```

Imagen 25: Finalizando la extracción desde browserOverlay.

En la función *execute()* debemos realizar también algunos cambios con respecto a la versión anterior. Antes utilizábamos la directiva *content.document* para acceder al documento de la ventana de la que queremos extraer el contenido. En *Webextensions* esta sintaxis ha cambiado ligeramente y utilizamos la directiva *window.document* para acceder al contenido del árbol *DOM.*

5.3.2 *createNamespaces.js*

En las tres extensiones definimos un *namespace.* Un *namespace* es un medio para agrupar y organizar clases de una manera jerárquica. Para conseguirlo definimos una serie de objetos con los que podremos acceder a cualquier directorio de nuestra extensión y así poner en funcionamiento la extracción.

```
ConEx = new Object();
ConEx.loader = new Object();
ConEx.misc = new Object();
ConEx.site = new Object();
ConEx.util = new Object();
ConEx.conex = new Object();
ConEx.menu = new Object();
```

Imagen 26: Namespace en ConEx.

5.3.3 ConEx: extracción del contenido principal

A la hora de poner en funcionamiento el código de extracción *JavaScript* en ConEx surgieron algunos problemas relacionados con la obtención de los enlaces de las páginas y la creación de los *iframes*. Tanto el funcionamiento del procedimiento como las soluciones a estos problemas se comentarán en este apartado.

ConEx utiliza como entrada una página clave (*key page*) y devuelve como resultado el contenido principal. Para conseguir aislarlo utiliza varias páginas pertenecientes al mismo sitio web. Por tanto, tiene que determinar qué paginas son buenas candidatas para la obtención del contenido. El proceso de extracción se puede resumir en los tres siguientes puntos:

- Como entrada recibe una página web (*key page*) y selecciona un conjunto de páginas que pertenecen al mismo sitio web.

- Por cada página en el conjunto el algoritmo compara su árbol *DOM* con el de la *key page*. Si se encuentra con algún nodo que está repetido en en otra página actualiza un contador.

- El conjunto de nodos *DOM* que no se encuentra repetido en ninguna otra página se añade al conjunto de nodos candidatos para representar el contenido.

Para poder obtener el conjunto de páginas necesario para el análisis, debemos obtener los enlaces pertenecientes al dominio de la página principal. Hay dos motivos principales por los cuales es necesario que comprobemos el origen de las direcciones que carga la extensión; eficiencia y seguridad. Se considera que dos páginas tienen el mismo origen si el protocolo, el puerto (en el caso de que se haya definido uno) y el *host* coinciden para ambas páginas. En la siguiente tabla se dan algunos ejemplos para la *URL* http://market.apple.com:

URL	Resultado	Motivo
http://market.apple.com/dir/in.html	Mismo origen	
http://market.apple.com/dir2/ab/in.html	Mismo origen	
https://market.apple.com/dir2/ab/in.html	Distinto origen	Ha cambiado el protocolo
http://market.apple.com:25980/dir2/ab/in.html	Distinto origen	Ha cambiado el puerto
http://surface.apple.com/dir2/ab/in.html	Distinto origen	Ha cambiado el *host*

En un sitio web normalmente hay enlaces que apuntan a otros dominios; por ejemplo para la publicidad o las redes sociales. A nosotros tan solo nos interesan los enlaces dentro del propio dominio ya que de lo contrario el rendimiento de la extensión se resentiría bastante y no nos aportaría ninguna información útil para la tarea. Por otra parte en algunas páginas nos puede dar algún problema de seguridad el mero hecho de cargar enlaces de otros dominios y por ello se interrumpirá la ejecución. Esto se hace con el fin de prevenir ataques *XSS*(*Cross-Site scripting*) y se puede programar simplemente añadiendo la cabecera *X-Frame-Options*

SAMEORIGIN en el archivo de configuración del servidor. Por estos motivos es necesario que definamos una función en *ConEx.js* que compruebe el origen de las *URL* que almacena.

```
this.checkURL = function(URL)
{
    var xhttp = new XMLHttpRequest();
    var isPage = false;

    xhttp.open("GET", URL, false); //Sync mode

    xhttp.onreadystatechange = function () {
        if (this.readyState == this.DONE && this.getResponseHeader("Content-Type")!=null) {
            if(!this.getResponseHeader("Content-Type").includes("html")) {
                return false;
            }
        }
    }

    try {
        xhttp.send();
        return true;
    } catch(e) {
        return false;
    }
}
```

Imagen 27: Comprobamos el origen de los enlaces que cargamos.

```
this.getLinks = function(anchorLinks)
{
    var links = new Array();

    for (var anchorLinkIndex = 0; anchorLinkIndex < anchorLinks.length; anchorLinkIndex++)
    {
        var anchorLink = anchorLinks[anchorLinkIndex];
        var linkStr = anchorLink.href;
        var linkDOMPath = this.getDOMElementPath(anchorLink);
        var link = new ConEx.site.Link(linkStr, linkDOMPath);

        // Not duplicate
        if (ConEx.misc.Misc.arrayContainsSimilar(links, link) || !this.checkURL(linkStr))
            continue;
        links.push(link);
    }

    return links;
}
```

Imagen 28: Añadimos los enlaces que cumplen las condiciones.

Con la función de la *Imagen 26* comprobamos si la *URL* que cargamos es un documento *HTML* y si no pertenece a otro dominio. Necesitamos comprobar que se trata de un documento *HTML* para evitar cargar enlaces con archivos *PDF, JPG* o *PNG.* Si se cumplen estas condiciones lo añadimos al array de enlaces. Podría darse la situación de que una determinada página web no tuviera enlaces. No es la situación más común, pero podría ocurrir puesto que hay páginas web muy sencillas con un único apartado. En el caso de que el usuario intentara ejecutar *ConEx* en este tipo de páginas la extensión no podría extraer el contenido principal. Por ello es necesario que la extensión sea capaz de determinar previamente si se va a poder ejecutar en la página deseada o no. En caso de que no sea posible analizar la página los más apropiado sería informar al usuario de que la acción que pretende realizar no es posible. Para ello comprobaremos el valor de la variable *links.length* almacene un valor superior a 0, en caso contrario mostramos un mensaje al usuario, avisamos al *background script* e interrumpimos la ejecución.

Imagen 29: Mensaje que se mostrará cuando no se pueda extraer el contenido.

Para la realización de las comparaciones entre los distintos árboles *DOM* utilizamos un elemento *HTML* llamado *iframe.* Un *iframe* es un contenedor que te permite incrustar en su interior un documento *HTML.* De esta forma podríamos tener varias páginas web dentro de una misma pestaña del navegador. La creación de estos elementos se realiza en el documento *PageLoader.js* del directorio *loader.* Para cargar una página dentro de un *iframe* primero lo creamos otorgándole una serie de atributos como un identificador, un nombre, un tipo, la altura, la anchura y la *URL.* En *Webextensions* es necesario que la *URL* que cargamos en el *iframe* se cargue en el momento de la creación del mismo. Por tanto, la función *this.setIframeURL(URL)* tendrá también que ser llamada desde la función *this.createNewIframe(URL)* o de lo contrario los *iframe* nos los cargará vacíos.

5.3.4 TemEx: extracción de la plantilla

Empleamos un algoritmo que recibe como entrada una página clave (*key page*) y devuelve la plantilla. Para conseguir identificar e aislar la plantilla, primero analizamos qué páginas del sitio web deben ser analizadas. El proceso de extracción se resume en los siguientes tres puntos:

- Intentamos reducir el número de páginas a analizar. Para conseguirlo, a partir de la página clave, identificamos un subdígrafo completo.

- Se establece un sistema de votación con el fin de solventar el problema de que varias de las páginas que componen el sitio tengan plantillas distintas.

- Se extrae la plantilla comparando el conjunto de páginas que hemos seleccionado previamente. Para hacer esto empleamos la técnica *equal top-down mapping* (*ETDM*) entre el árbol *DOM* de la página clave y el árbol del resto de páginas.

A la hora de extraer la plantilla es importante tener en cuenta que marco de la página queremos analizar. Una pestaña del navegador se compone de una colección de *frames* o marcos. Los *content scripts* se pueden insertar en el marco superior de una pestaña o en todos. Esta característica la podemos activar o desactivar en el archivo *manifest.json.* El marco que nos interesa a nosotros para obtener la información de la página web es el superior. En los otros marcos se colocan los enlaces a paneles publicitarios de otros dominios así como a menús desplegables que permiten acceder a los perfiles de las redes sociales de la propia empresa de la página. Estos enlaces no nos proporcionan ninguna información, por lo tanto es importante que en el *manifest.json* hayamos definido el parámetro <*all_frames*> a *false* en la sección donde se definen las rutas de los *content scripts.*

5.3.5 MenEx: extracción del menú

Esta extensión recibe una plantilla como entrada y devuelve el menú. A diferencia de las anteriores tan solo requiere de una única página, por lo que es la extensión más rápida de las tres. El proceso de extracción se puede resumir en las siguientes cuatro fases:

- Se identifican todos los nodos *DOM* de la página que tienen hijos y se les asigna un peso a cada uno. Después se construye un conjunto de nodos con aquellos que tienen un peso más alto.

- Por cada nodo en el conjunto analizamos sus antecesores y comprobamos sus pesos. Si uno de los antecesores tiene un peso que un nodo que ya se ha procesado, eliminamos el nodo del conjunto que hemos obtenido en la primera fase y lo reemplazamos con su antecesor.

- Los nodos que forman parte del menú se detectan comparando los nodos del conjunto. Por cada nodo procesamos un valor que representa el peso medio de sus descendientes. El menú probablemente sea aquel nodo que tenga mejor media.

- En la última fase comprobamos si hay alguna otra rama de nodos que pudiera pertenecer al menú. Esto se hace comparando el conjunto de nodos *DOM* seleccionado con los ancestros del nodo seleccionado el la fase anterior.

Los *content scripts* en el caso de *MenEx* no requieren de muchas modificaciones en *Webextensions* con respecto a la anterior arquitectura. Esto se debe a que el funcionamiento de *MenEx* no necesita tanto acceso a elementos internos como las otras dos extensiones. Aun así nos será necesario cambiar algunas directivas del archivo *browserOverlay* que como ya se ha explicado es fundamental para el funcionamiento de las comunicaciones entre los *background* y los *content script.*

5.4 Localización de las tres extensiones

En este apartado vamos a tratar la internacionalización de las extensiones en *Webextensions*. Ya hemos comentado previamente que las extensiones funcionan de una manera ligeramente diferente a como funcionaban con las anteriores arquitecturas. No obstante en este apartado vamos a desarrollar más detenidamente cómo funcionan la internacionalización en las nuevas versiones del navegador *Firefox*.

5.4.1 Jerarquía de una extensión internacionalizada

Para almacenar las cadenas de texto es necesario que creemos un directorio llamado *_locales* en el directorio raíz de la extensión. En este directorio crearemos un subdirectorio diferente por cada idioma al que queramos traducir las cadenas de texto. Dentro de cada uno de estos subdirectorios colocaremos un documento *JSON* que se llamará *messages.json.* Es importante que el nombre del subdirectorio se corresponda con la abreviación internacional de ese idioma. Por ejemplo, si queremos almacenar los mensajes en inglés crearemos un subdirectorio llamado "*en"* dentro del directorio *_locales* y dentro de ese subdirectorio colocaremos el archivo *messages.json* con las cadenas de texto en inglés. Si para un mismo idioma queremos traducir el texto a una variante regional, tendremos que nombrar al subdirectorio con el nombre de esa variante utilizando el guión bajo, por ejemplo "*en_US*" en el caso de que queramos utilizar la variante del inglés americano.

5.4.2 Estructura de *messages.json*

Este archivo sigue la estructura tradicional de un archivo *JSON;* es decir, cada uno de sus componentes es un objeto con un nombre que contiene a su vez un mensaje y una descripción.

```
{
  "extensionName": {
    "message": "Content Extractor",
    "description": "Name of the extension."
  },

  "messageTitle": {
    "message": "...Extracting the web content..."
  },

  "bodyMessage": {
    "message": "This process can take a few minutes. Please wait."
  },

  "extensionDescription": {
    "message": "Extracts the content from a Webpage",
    "description": "Description of the extension."
  },
```

Imagen 30: Estructura del archivo messajes.json

5.4.3 Recuperar los mensajes mediante *JavaScript*

Para recuperar los mensajes de los archivos *JSON* hacemos uso de la *API i18n.* Es una *API* bastante simple que tan solo dispone de unas pocas funciones para recuperar los mensajes.

- *i18n.getMessage()*: Con este método recuperamos el mensaje de un idioma determinado.

- *i18n.getAcceptedLanguages()*: Te permite recuperar los idiomas compatibles con el navegador. También se puede utilizar para personalizar la interfaz del usuario de una manera u otra de acuerdo con el idioma del navegador.

- *i18n.detectLanguage()*: te permite determinar el idioma de un string que le pasas como parámetro.

En nuestro caso el único método que nos interesa es el que nos permite recuperar los mensajes para mostrarlos traducidos, como se puede apreciar en la Imagen 32.

```
var header = browser.i18n.getMessage("messageTitle");
var content1 = browser.i18n.getMessage("bodyMessage");
```

Imagen 31: Recuperamos dos mensajes.

Con el código anterior recuperamos dos mensajes; el primero corresponde a aquel mensaje que tiene por nombre *messageTitle* en el *messages.json* y el segundo *bodyMessage.*

6. Pruebas de funcionamiento de las tres extensiones en *Webextensions*

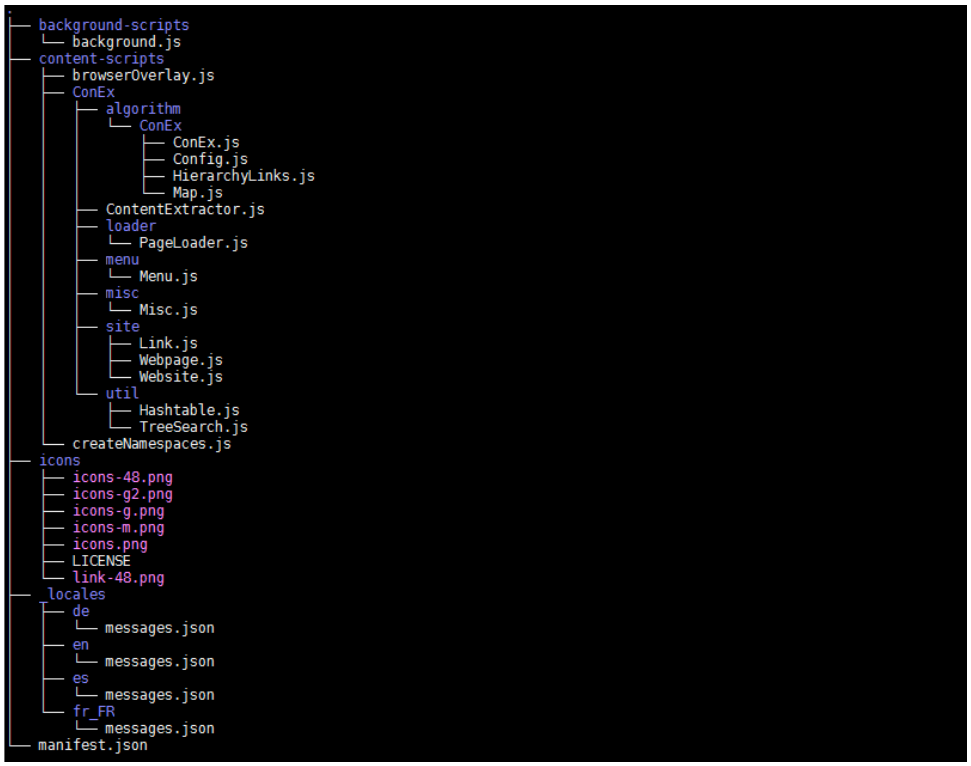

Imagen 32: Jerarquía final de una extensión.

A continuación, ponemos a prueba a las tres extensiones ya desarrolladas en *Webextensions* con las páginas web oficiales de Ann Malaspina y The Web Conference.

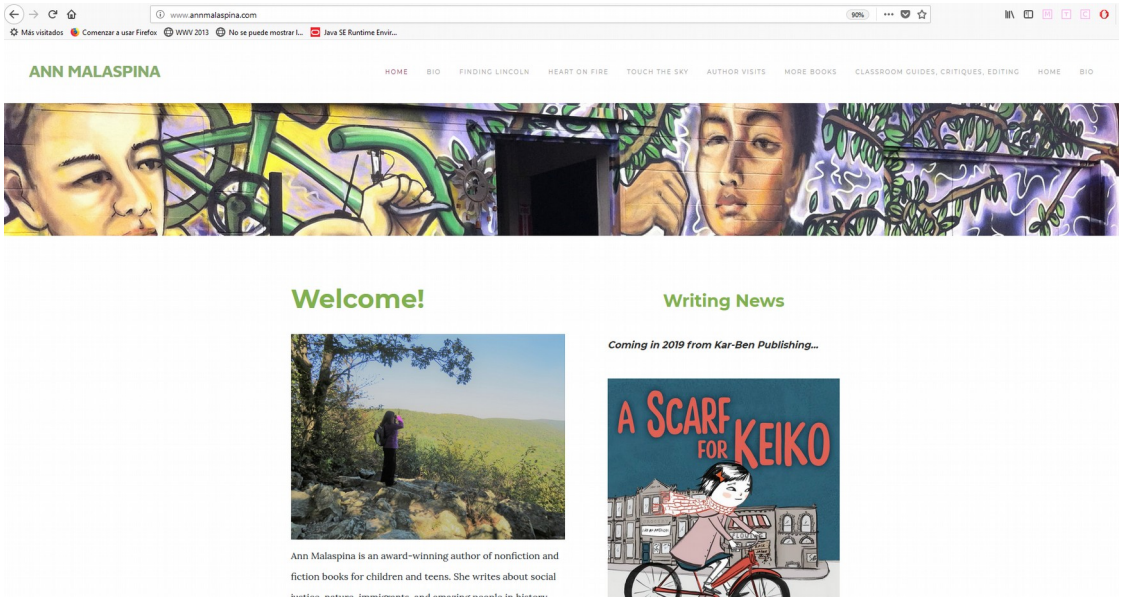

Imagen 33: Página completa.

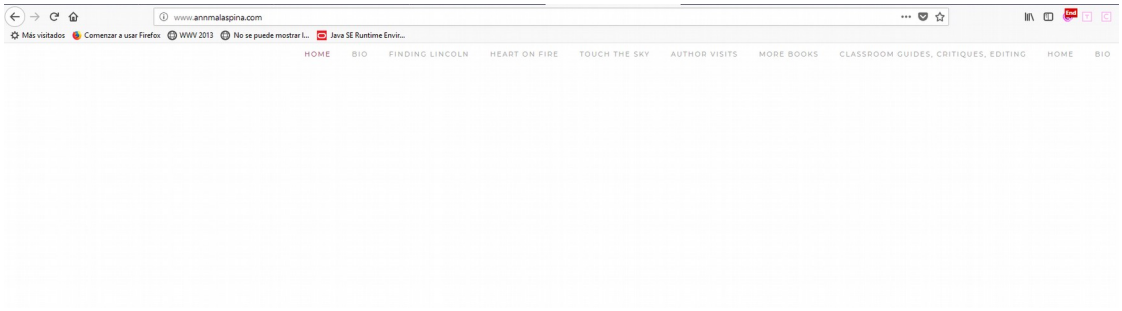

Imagen 34: Menú extraído con MenEx.

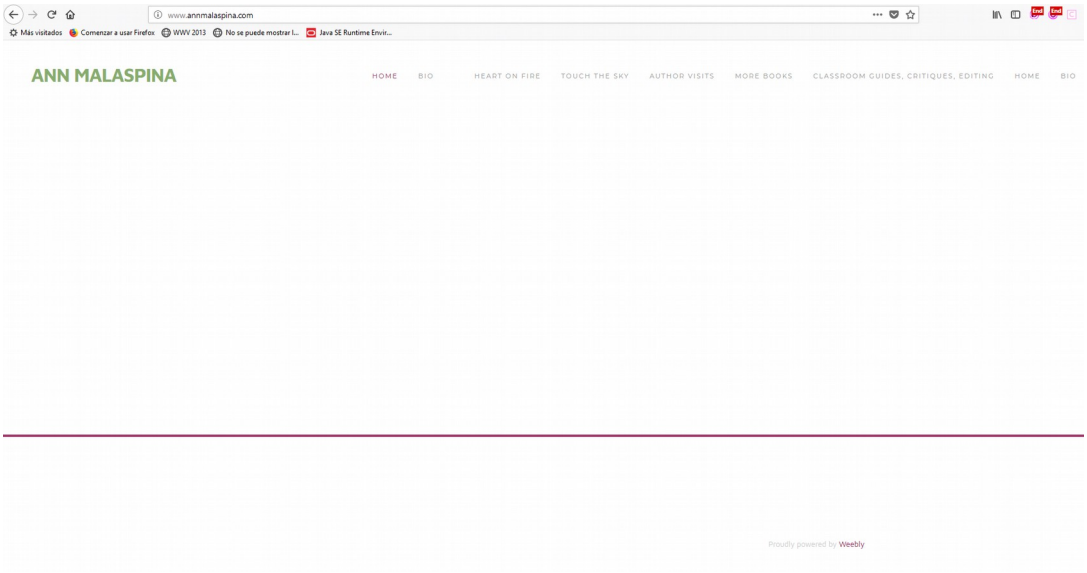

Imagen 35: Plantilla extraída con TemEx.

En el caso de la extracción de la plantilla como se puede observar en la *Imagen 35* puede ocurrir que el menú forme parte del árbol *DOM* de la plantilla y por tanto nos aparecerá con la plantilla extraída. Algo similar ocurre en el siguiente ejemplo.

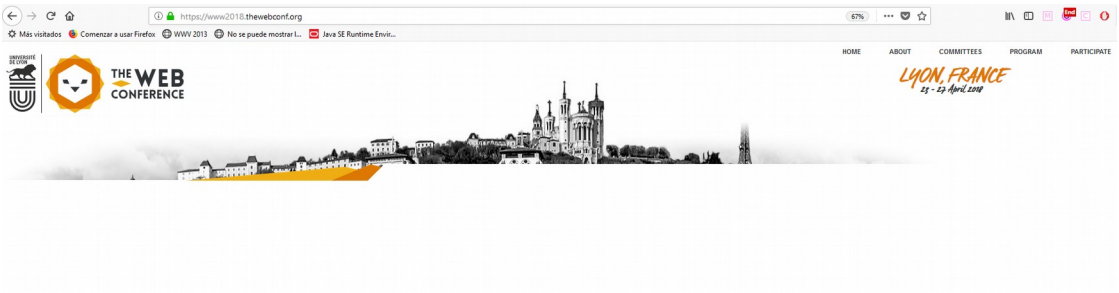

Imagen 36: Ejemplo de plantilla con el menú incorporado.

7. Ampliaciones y mejoras

En este apartado se desarrollan qué mejoras de rendimiento se han añadido a las nuevas versiones desarrolladas con la última arquitectura. Un problema que se presentaba con las versiones anteriores tiene que ver con el tiempo de ejecución. Como se ha desarrollado previamente, para detectar cuál es el contenido que nos interesa debemos analizar los enlaces de las propias páginas. Este proceso, según la página, puede ser muy costoso en términos del tiempo de ejecución. Por este motivo se propondrán algunas mejoras que reducirán la cantidad de páginas necesarias intentado conservar la misma precisión y los mismos resultados.

7.1 Tiempos *MenEx*

MenEx es la extensión que mejores tiempos daba ya en su versión *legacy* si la comparamos con las otras dos. Esto se debe a que tan solo requiere cargar una única página para analizarla y obtener el menú, por tanto no es necesario realizar ningún cambio en cuanto a lo que se refiere a los enlaces que analiza. No obstante, los tiempos de ejecución debidos únicamente al cambio de arquitectura y a la mejora de ejecución de *JavaScript* que traen las últimas versiones de *Firefox* son suficientes como para que los tiempos se reduzcan de manera considerable, como podemos observar en la siguiente tabla.

Página web	***Legacy runtime***	***Webextensions runtime***
https://www2018.thewebconf.org/	6,51 s	1,50 s
http://www.javiercelaya.es/es/	1,08 s	0,71 s
https://www.trendencias.com/	16,03 s	3,00 s
https://www.turfparadise.com	16,65 s	2,80 s
https://www.u-tokyo.ac.jp	14,12 s	2,73 s
https://www.savethechildren.net	7,42 s	1,66 s
https://college.harvard.edu	7,42 s	1,92 s
https://www.raspberry.org	2,41 s	0,92 s
www.annmalaspina.com	2,51 s	0,82 s
dublin.ie	5,82 s	1,42 s
www.amateurgourmet.com	13,74 s	2,56 s
www.museodelprado.es	2,60 s	0,91 s
www.rfet.es	5,15 s	1,43 s

www.centralparknyc.or	7,20 s	1,80 s
manytools.org	6,17 s	1,55 s
clotheshor.se	2,56 s	0,89 s
www.unicef.org	6,85 s	1,56 s
www.news-medical.net	7,62 s	1,57 s
teachreal.wordpress.com	3,64 s	0,96 s
engineering.mit.edu	5,84 s	1,56 s
www.cleanclothes.org	2,50 s	0,54 s
riotimesonline.com	26,85 s	4,52 s
www.ox.ac.uk	31,14 s	3,56 s
www.filmaffinity.com	12,09 s	2,30 s
www.coiicv.org	6,15 s	1,60 s
www.thelawyer.com	10,85 s	2,59 s
www.toureiffel.paris	8,34 s	2,07 s
mobileworldcongress.com	6,80 s	1,61 s
Tiempo medio:	**8,78 s**	**1,82 s**

7.2 Tiempos *TemEx*

TemEx utiliza un *complete subdigraph* (*CS*) que representa una colección de páginas web que están vinculadas entre sí por parejas. Esto es importante porque las páginas que se enlazan con otras a través de los ítems de los menús normalmente forman un *CS.* Esto nos permite identificar las páginas que contienen el menú del sitio web y al mismo tiempo es útil ya que estas páginas son las raíces de las secciones que aparecen en el menú de la página. Si por ejemplo el menú de un sitio web tiene 6 secciones, ya que son 6 enlaces que llevan a diferentes páginas dentro de un sitio web, estas formarán un *6-CS.* Este conjunto de páginas formarán parte del conjunto de páginas candidatas para extraer la plantilla.

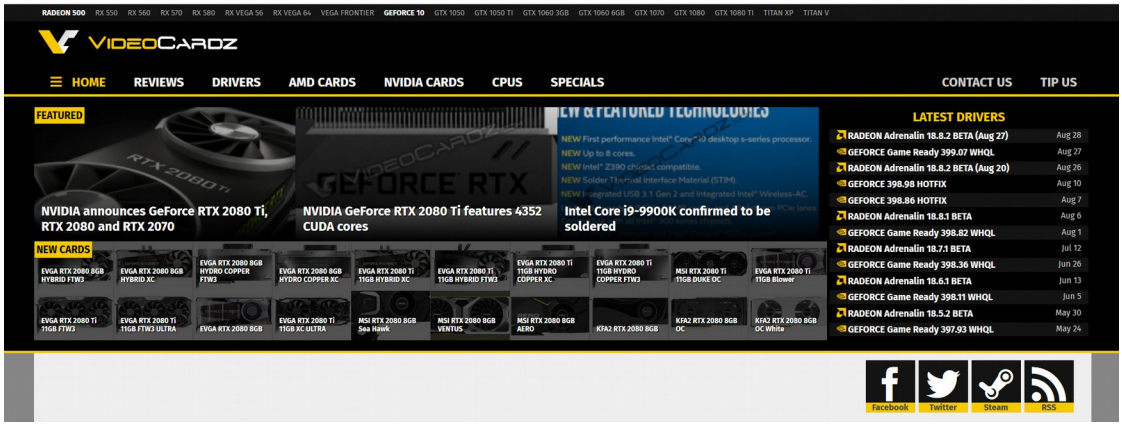

Imagen 37: El menú de este sitio web forma un 6-CS.

Una mejora importante a considerar en esta extensión tiene que ver con el cálculo de los *CS.* No es necesario que se carguen todos los enlaces, la carga de páginas debería terminar una vez encontrado el *CS.* De la misma manera en el proceso de *mapping* la búsqueda del *CS* sería más eficiente si se hiciera en profundidad y no en anchura. Con estas mejoras implementadas podemos observar la mejora de los tiempos de ejecución en la siguiente tabla.

Página web	***Legacy runtime***	***Webextensions runtime***
https://es.fifa.com/	14,72 s	0,71 s
http://www.javiercelaya.es/es/	5,77 s	1,81 s
https://www.trendencias.com/	102,34 s	35,20 s
https://www.turfparadise.com	33,30 s	14,45 s
https://www.u-tokyo.ac.jp	40,42 s	13,11 s
www.mediamarkt.es	26,56 s	52,69 s

https://www.raspberry.org	8,50 s	5,74 s
www.annmalaspina.com	9,26 s	6,71 s
www.vidaextra.com	95,57 s	34,80 s
www.apple.com	15,59 s	6,01 s
www.wikipedia.org	19,10 s	15,50 s
www.swimmingpool.com	10,90 s	18,01 s
http://www.lashorasperdidas.com/	23,85 s	14,00 s
manytools.org	17,01 s	8,71 s
http://www.us-nails.com/	4,76 s	1,19 s
https://linuxmint.com/	9,50 s	2,78 s
http://puppylinux.com/	12,47 s	2,90 s
https://videocardz.com/	25,27 s	11,38 s
https://www.ietf.org/	10,20 s	5,94 s
https://www.pccomponentes.com/	95,46 s	63,82 s
https://www.appinformatica.com/	20,27 s	6,57 s
https://www.dropbox.com	32,24 s	34,71 s
https://www.genbeta.com/	81,62 s	13,12 s
https://www.cpuid.com/softwares/cpu-z.html	65,20 s	16,72 s
http://www.upv.es/	30,43 s	9,50 s
https://www.oracle.com/index.html	43,07 s	73,14 s
https://www.maxon.net/en/products/cinebench/	12,59 s	4,49 s
Tiempo medio	**32,08 s**	**17,55 s**

7.3 Tiempos *ConEx*

ConEx utiliza un conjunto de páginas para extraer la plantilla (*n-SET*). Utilizar un conjunto de páginas más grande aumenta la precisión con la que se puede detectar la plantilla, no obstante aumenta también el tiempo de ejecución. El algoritmo original utilizaba un *4-SET* para realizar el *mapping*; es decir, un conjunto de 4 páginas. El proceso de *mapping* consume un 80% del tiempo total de ejecución, por tanto agilizar este algoritmo resulta crucial si queremos reducir el tiempo global de la extensión. Para lograrlo, en la versión de *Webextensions* se ha decidido cambiar el *4-SET* por un *3-SET*. La reducción de este conjunto sumado a la mejora del rendimiento de *JavaScript* ha implicado una mejora considerable en los tiempos de ejecución como podemos observar en la siguiente tabla.

Página web	***Legacy runtime***	***Webextensions runtime***
https://www.annmalaspina.com	38,70 s	20,47 s
https://linuxmint.com/	61,20 s	6,56 s
http://puppylinux.com/	18,93 s	5,05 s
manytools.org	170,70 s	21,47 s
https://www.raspberry.org	53,62 s	26,34 s
www.amateurgourmet.com	569,33 s	140,70 s
http://clotheshor.se/	7,78 s	9,16 s
dublin.ie	25,31 s	30,81 s
https://teachreal.wordpress.com/	39,34 s	17,80 s
www.rfet.es	99,02 s	22,42 s
www.centralparknyc.org	121,57 s	89,75 s
www.coiicv.org	41,81 s	23,8 s
www.thelawyer.com	298,08 s	81,60 s
www.toureiffel.paris	36,88 s	38,90 s
Tiempo medio	**113,01 s**	**38,20 s**

En la mayoría de los casos la mejora es bastante considerable y la precisión con la que se detectan los nodos se mantiene bastante alta a pesar de haberse reducido el conjunto de páginas. En el peor de los casos los tiempos son similares entre ambas versiones, pero en aquellas páginas con un gran número de nodos la mejora es muy considerable ya que el tiempo de ejecución se reduce más de un 50%.

8. Publicación de las extensiones

Para que las extensiones sean publicadas hay una serie de condiciones que tienen que cumplir. *Mozilla* ofrece una serie de guías que ayudan al desarrollador a saber qué objetivos y condiciones tienen que cumplir sus desarrollos [17]. Lo primero que se exige es que todas las extensiones tengan una descripción que le ofrezca al usuario una explicación sobre la funcionalidad básica de la extensión que se está instalando. Para facilitar este proceso se recomienda que cada extensión tenga una página de ayuda que describa la funcionalidad completa y ofrezca datos de contacto e información sobre los desarrolladores. En nuestro caso al tratarse de tres extensiones hemos desarrollado tres sitios webs diferentes que comparten una plantilla común para que el usuario que quiera pueda consultar toda la información al respecto.

Para desarrollar estas páginas de ayuda hemos hecho uso dos tecnologías fundamentales para el desarrollo web; es decir, *HTML* [18] y *CSS* [19]. De las versiones *legacy* solo *MenEx* tenía una página hecha con una plantilla determinada. A partir de esta plantilla se han desarrollado las dos otras páginas web con ligeras modificaciones. Por ejemplo, se ha añadido una página *FAQ* (Frequently asked questions) en la que se resuelven las dudas más comunes de los usuarios a la hora de trabajar con las extensiones.

Imagen 38: Web de ayuda de MenEx.

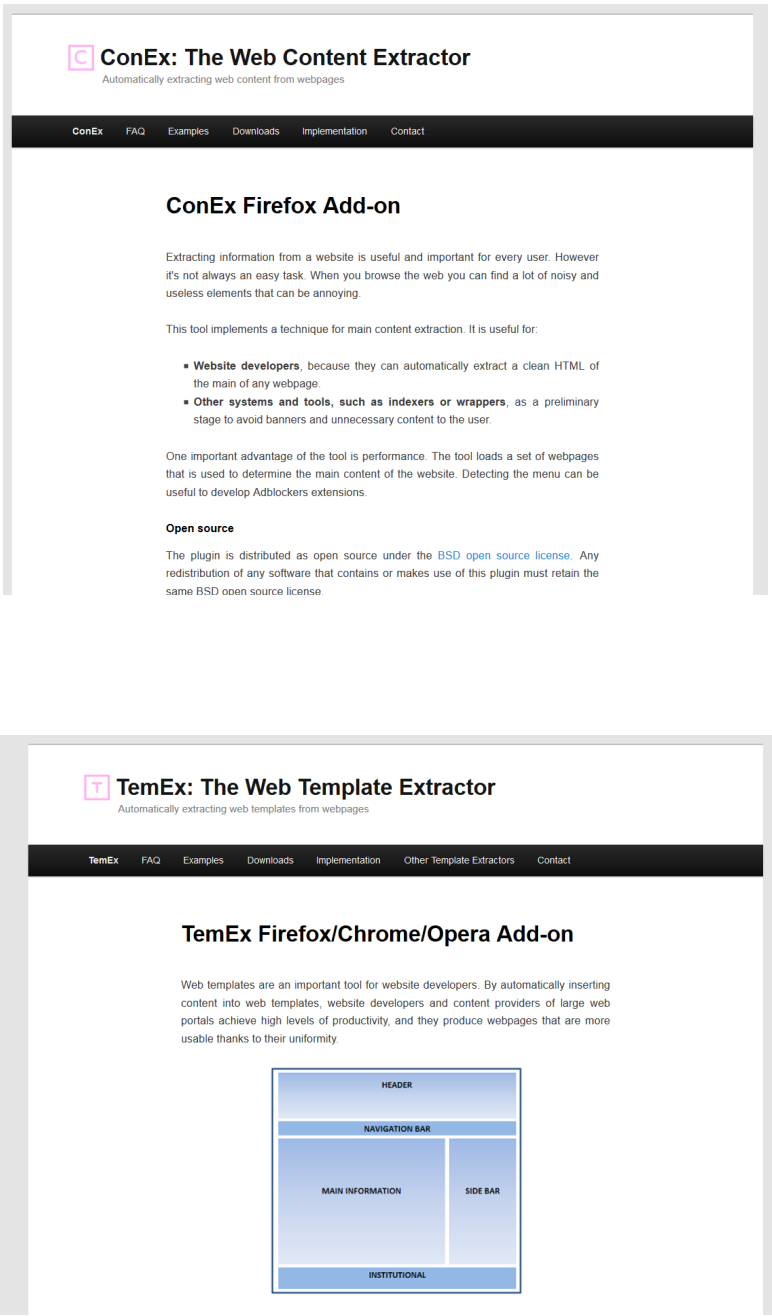

Imagen 40: Página de ayuda de TemEx.

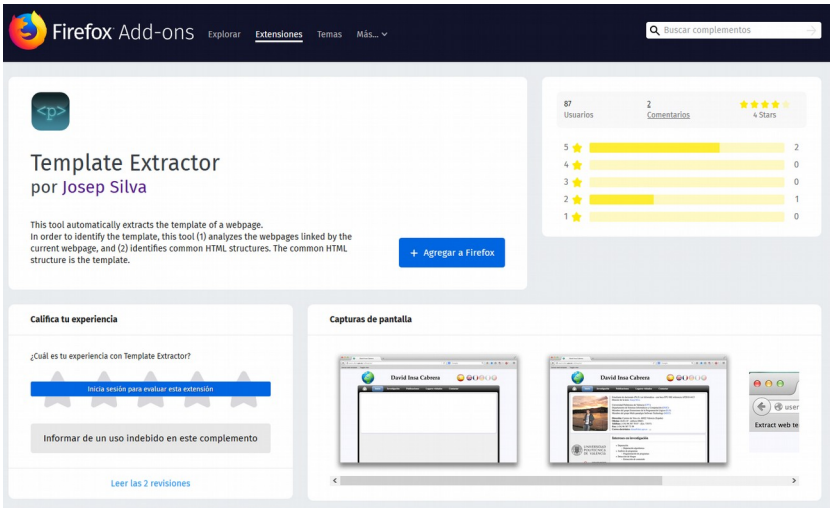

Imagen 41: TemEx en Mozilla Market.

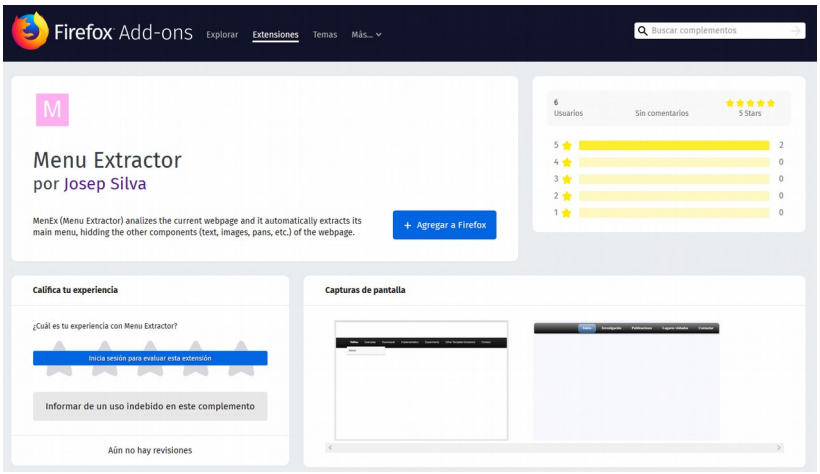

Imagen 42: MenEx en Mozilla Market.

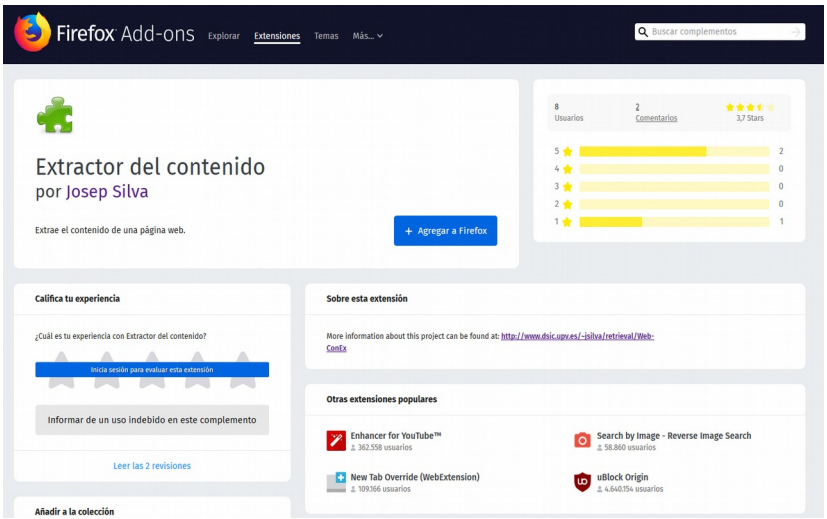

Imagen 43: ConEx en Mozilla Market.

8.1 Proceso de publicación

Para que una extensión finalmente se publique, *Mozilla* tiene que firmarla digitalmente. En caso de que no se haya firmado, la extensión tan solo podrá instalarse en las versiones de desarrollador de *Firefox.* Para esto se pueden seguir tres métodos diferentes.

- Subir la extensión a través de la herramienta *Developer Hub on AMO.*
- Utilizar la *API* que se ha desarrollado oficialmente para firmar las extensiones [20].
- Utilizar la aplicación *web-ext sign.*

Las dos últimas opciones te devolverán la extensión firmada, pero no la agregarán a la lista de distribución de *Mozilla.* En cambio si los desarrolladores optan por subir sus extensiones utilizando la primera opción, podrán elegir si añadirla a la lista o auto distribuir la extensión comprimida. Independientemente de la opción que decida el desarrollador, todos los complementos pasarán por un proceso de validación automático y podrán pasar también el proceso de revisión manual.

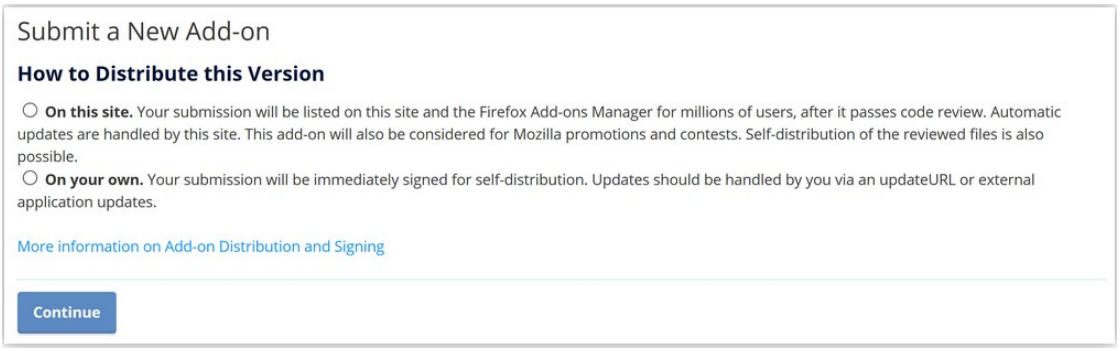

Imagen 44: El proceso de publicación de una extensión en Mozilla Market.

Por lo general lo más conveniente es añadir el complemento a la lista oficial para que pueda estar al alcance de cualquier usuario accediendo al *Mozilla Market,* sin embargo hay desarrolladores que prefieren distribuir el *software* por su propia cuenta.

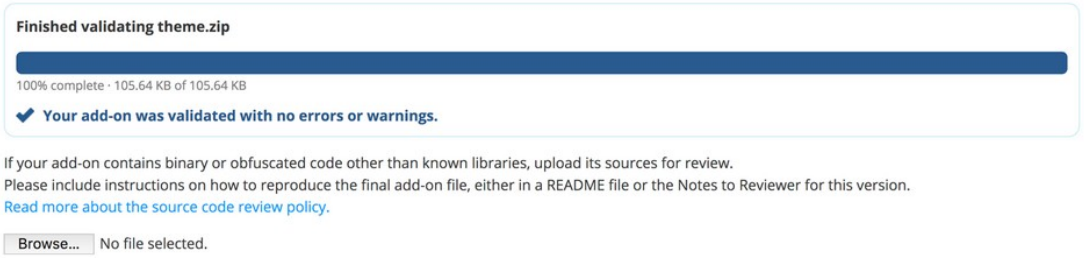

Imagen 45: Validación del complemento completada.

9. Conclusiones

De la realización de estas tres migraciones podemos sacar una serie de conclusiones sobre la nueva arquitectura que ha propuesto la fundación *Mozilla* para su popular navegador *Firefox.* Con este cambio de paradigma, *Firefox* gana en velocidad, tanto de ejecución del propio navegador como de las extensiones que funcionan con él. El navegador gana también estabilidad y especialmente algo trascendental a día de hoy en el mundo de Internet: seguridad. Gana estabilidad ya que las extensiones *XUL/XPCOM* podían verse afectadas cada vez que algún componente interno del navegador cambiaba, ahora esto no sucedería debido a que no se tiene un acceso a estos elementos. Al aislar la ejecución de las extensiones en compartimentos evitando así el acceso a ciertos componentes fundamentales del navegador se garantiza más seguridad, confidencialidad y estabilidad a los usuarios aun cuando las futuras versiones del navegador cambien su funcionamiento interno.

El apartado de la seguridad no solamente ha mejorado por el cambio de arquitectura, sino también por el cambio de políticas, ya que ahora subir una nueva extensión a la tienda requiere que pase un proceso de revisión tanto automático como manual con el fin de garantizar que el *software* está libre de código malicioso. No obstante sí que es cierto que hay ciertos elementos que son mejorables dentro de este apartado. Por ejemplo, muchas de las *APIs* siguen siendo experimentales y no está claro cómo de estricta va a ser la fundación *Mozilla* con su uso. Otro aspecto mejorable que conviene comentar hace referencia al proceso de revisión que hemos comentado anteriormente. El proceso parece ser estricto, no obstante al ser parcialmente manual y parcialmente automático es muy complicado llegar a cubrir todas las amenazas posibles, ya que por ejemplo una página web maliciosa podría intentar modificar funciones dentro de la extensión y ejecutar código con privilegios de administrador. Esto hace que el proceso de revisión manual tenga que ser muy exhaustivo con el fin de evitar estos problemas, aun cuando esto signifique un proceso de aprobación más lento.

Por otra parte, tanto los usuarios como los desarrolladores obtienen ciertas ventajas con el cambio debido a la multiplataforma. Si las extensiones son compatibles para todos los navegadores facilita que usuarios de otras plataformas prueben *Firefox* sin tener que renunciar a las extensiones que utilizaban en dichas plataformas. Esto también facilita el trabajo a los desarrolladores puesto que no requieren reescribir su código para que funcione en cualquier otro navegador, sino simplemente les bastaría con cargar el mismo archivo comprimido con el código en *Opera* o *Google Chrome.* Dado el estado del mercado actual de navegadores, unificar al máximo posible el desarrollo de *software* y características para los mismos era una necesidad cada vez más grande y para *Firefox* inevitable, dado que su cuota de mercado llevaba unos años reduciéndose en favor de *Google Chrome.* La facilidad para el desarrollo de nuevas extensiones debido a la simplificación de la arquitectura y la posibilidad de migrarlas de un navegador a otro sin tener que cambiar el código es algo que probablemente volverá a traer usuarios de nuevo al navegador y ese era precisamente uno de sus objetivos principales.

Como resultado del proyecto obtenemos tres extensiones funcionales en las nuevas versiones de *Firefox* que anteriormente al estar escritas para la arquitectura anterior no se encontraban disponibles en el nuevo *Mozilla Marketplace*. Con este cambio se ha mejorado considerablemente la velocidad de ejecución, ya que en los tres casos reducimos aproximadamente un 50% el tiempo de ejecución. Por último ganamos compatibilidad con otras plataformas ya que las tres extensiones se pueden ejecutar usando el mismo código tanto en *Google Chrome* como en *Opera.*

10. Bibliografía

1. **[GC, 2018]:** *Google Chrome;* What are extensions?; *https://developer.chrome.com/extensions;* Consulta 2018

2. **[MDN, 2018]:** *Mozilla Developer Network;* XPCOM; https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM; Consulta 2018

3. **[MDN, 2018]:** *Mozilla Developer Network;* What are Webextensions?; https://developer.mozilla.org/bn-IN/docs/Mozilla/Add-ons/WebExtensions/What_are_WebExtensions; Consulta 2017

4. **[MDN, 2018]:** *Mozilla Developer Network;* Manifest.json; https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json; Consulta 2017

5. **[MDN, 2018]:** *Mozilla Developer Network;* Background; https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/background; Consulta 2017

6. **[MDN, 2018]:** *Mozilla Developer Network;* Content Scripts; https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_scripts; Consulta 2018

7. **[MDN, 2018]:** *Mozilla Developer Network;* DTD; *https://developer.mozilla.org/en-US/docs/Glossary/DTD*; Consulta 2018

8. **[MDN, 2018]:** *Mozilla Developer Network;* Webextensions/FAQ; *https://wiki.mozilla.org/WebExtensions/FAQ#Is_the_WebExtensions_API_a_Web_standard.3F;* Consulta 2018

9. **[Srilaya Bhavaraju, Tara Smith, Benny Zhang]:** Security Analysis of Firefox Webextensions; https://courses.csail.mit.edu/6.857/2018/project/srilayab-tsmith12-felicity-FFExt.pdf; Consulta 2018;

10. **[Nielsen Jakob]:** 10 Usability Heuristics for User Interface Design**;** https://www.nngroup.com/articles/ten-usability-heuristics/; Consulta 2018

11. **[Thuha Nguyen, 2018]:** *Mozilla Developer Network;* What is *XUL?; https://www.cs.colorado.edu/~kena/classes/7818/f00/presentations/xul.ppt;* Consulta 2018

12. **[MDN, 2018]:** *Mozilla Developer Network;* Package your extension; https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Package_your_extension; Consulta 2018

13. **[MDN, 2018]:** *Mozilla Developer Network;* Communicating with Background Scripts; https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_scripts#Communicating_with_background_scripts; Consulta 2018

14. **[Simpson Kyle, 2016]:** You don't know JS: ES6 & beyond; http://shop.oreilly.com/product/0636920033769.do; Consulta 2017;

15. **[Sebastian Lindström, 2018]:** Getting to know asynchronous JavaScript: Callbacks, Promises and Async/Await; https://medium.com/codebuddies/getting-to-know-asynchronous-javascript-callbacks-promises-and-async-await-17e0673281ee; Consulta 2018

16. **[MDN, 2018]:** *Mozilla Developer Network;* XUL Overlays; https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL/Overlays; Consulta 2018

17. **[MDN, 2018]:** *Mozilla Developer Network;* Submitting an add-on; https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/Distribution/Submitting_an_add-on; Consulta 2018

18. **[W3C, 2018]:** *World Wide Web Consortium;* HTML examples; https://www.w3schools.com/html/html_examples.asp; Consulta 2018

19. **[MDN, 2018]:** *Mozilla Developer Network;* How CSS works; https://developer.mozilla.org/en-US/docs/Learn/CSS/Introduction_to_CSS/How_CSS_works; Consulta 2018

20. **[MDN, 2018]:** *Mozilla Developer Network;* Signing; https://addons-server.readthedocs.io/en/latest/topics/api/signing.html; Consulta 2018