Cómo hacer un complemento para Google Docs.

La suite de Google nos permite desarrollar nuestros propios Add-Ons para utilizarlos en nuestros documentos, hojas de cálculo y presentaciones. En este artículo, veremos el caso concreto de cómo hacer un complemento que nos formatee e indente un bloque de código.

Juan Antonio Jiménez Torres

Fecha: 17/04/2018

Índice

1. Introducción

Últimamente trabajo mucho en Google Docs con bloques de código SQL. En la mayoría de los casos, se tratan de bloques de código en una sola línea, o indentados cada uno de una forma distinta, y según le parece a cada desarrollador: mezclando espacios y tabuladores e introduciendo los saltos de línea al buen tuntún.

He buscado alguna herramienta para normalizar todas estas queries, y si existe, no lo he encontrado. Pero lo bueno de trabajar en Google Docs, es que puedo desarrollarme mi propio "plugin" que haga ésto, sin apenas esfuerzo.

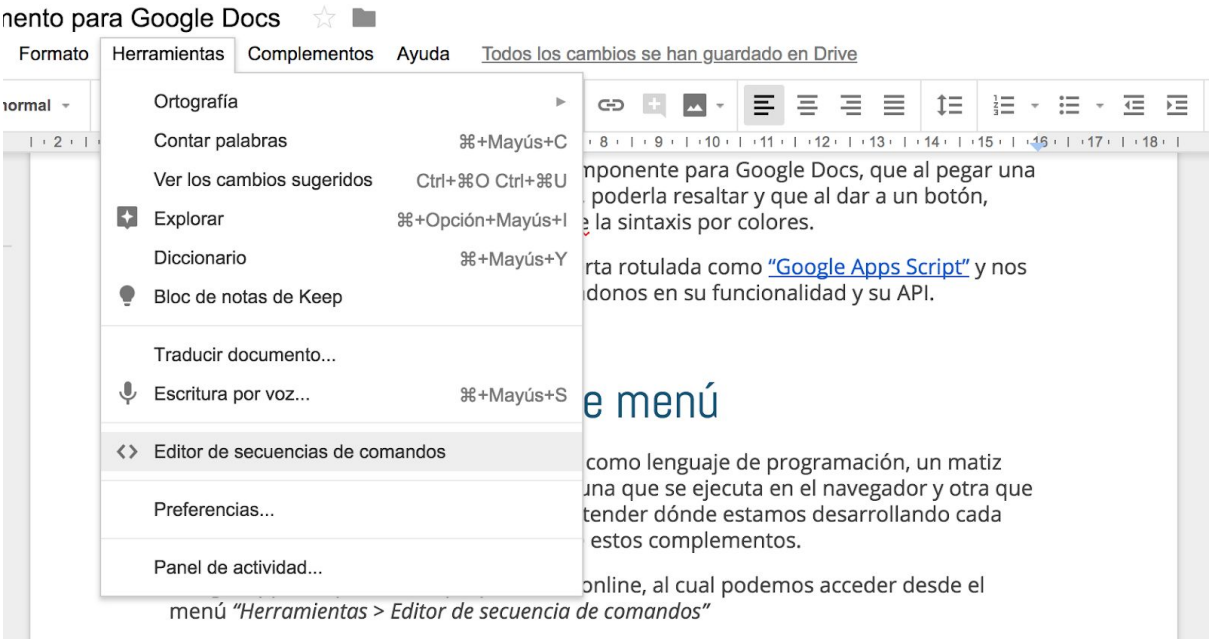
El objetivo de éste artículo es hacer un componente para Google Docs, que al pegar una query mal formateada o en una sola línea, poderla resaltar y que al dar a un botón, mágicamente se formatee y se indente sola.

Para ello echaremos un vistazo tras la puerta rotulada como "Google Apps Script" y nos atreveremos a cruzar su umbral, adentrándonos en su funcionalidad y su API.

2. Creando un punto de menú

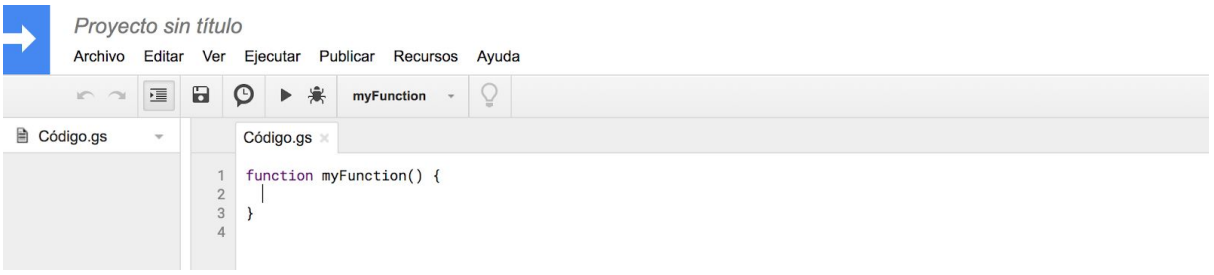
Aunque Google Apps Script usa JavaScript como lenguaje de programación, un matiz importante, es que consta de dos partes: una que se ejecuta en el navegador y otra que se ejecuta en los servidores de Google. Entender dónde estamos desarrollando cada pieza es esencial para captar la esencia de estos complementos.

Google Apps Script tiene su propio editor online, al cual podemos acceder desde el menú *"Herramientas > Editor de secuencia de comandos".*



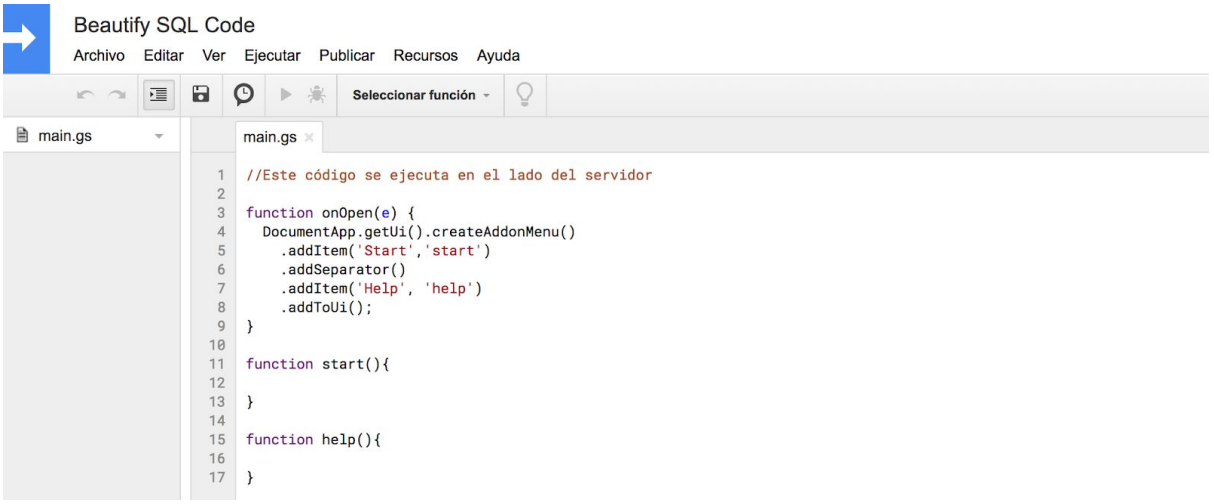
Según entramos al editor nos encontramos con un fichero abierto por defecto llámado

Código.gs con una función vacía. Lo que pongamos en este fichero, se ejecutará en el servidor antes de ser enviado al navegador. Aunque más adelante veremos que se puede establecer una comunicación entre el lado del cliente y del servidor.

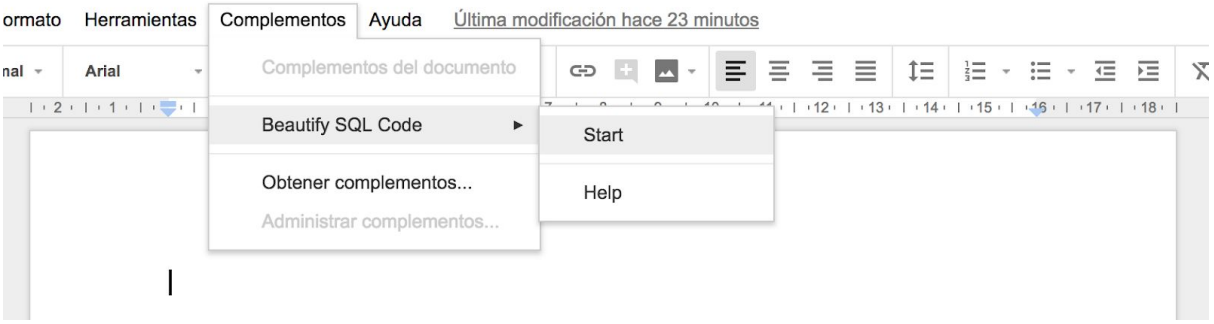


Lo primero que vamos a hacer es dar nombres a las cosas. Empecemos por dar nombre al proyecto, al que llamaremos "Beautify SQL Code". Y cambiaremos el nombre de código.gs por algo más aséptico que nos de información de cuál es el fichero donde se inicia nuestro Add-on. Lo llamaremos main.gs.

En él definiremos un método *onOpen(e)* que se ejecutará al abrir el documento de Google Docs y que recibe por parámetro dicho evento. En ese método, obtenemos la interfaz de usuario vinculada al documento, y añadimos dos entradas al menú de complementos junto con un separador. La primera entrada se llamará *"Start"* y al hacer click sobre ella invocará al método *"start()"*, mientras que la segunda la llamaremos *"Help"* e invocará al método *help()*.



Veamos en qué se traduce ésto en nuestro documento. Nos vamos al documento, lo recargamos y nos dirigimos al menú complementos. Ahora veremos una entrada en dicho menú que se llamará *"Beautify SQL Code"* con dos subelementos separados por una línea: *Start* y *Help*.



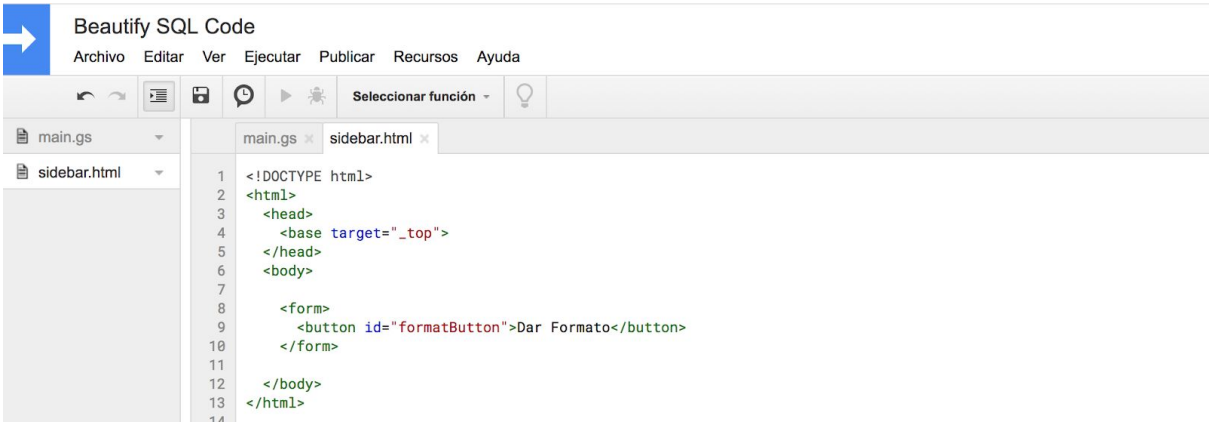
Ya tenemos nuestro punto de menú con sus elementos. Aunque si hacemos click en ellos no pasará nada, ya que aún no tenemos implementadas esas funciones.

3. Implementando una barra lateral

Vamos a implementar la función *start()* y que cuando se invoque abra una ventana lateral llamada *sidebar.html*.

```
function start(){
  var ui = HtmlService.createTemplateFromFile('sidebar').evaluate()
                      .setSandboxMode(HtmlService.SandboxMode.IFRAME)
                      .setTitle('Beautify SQL Code');
  DocumentApp.getUi().showSidebar(ui);
}
```

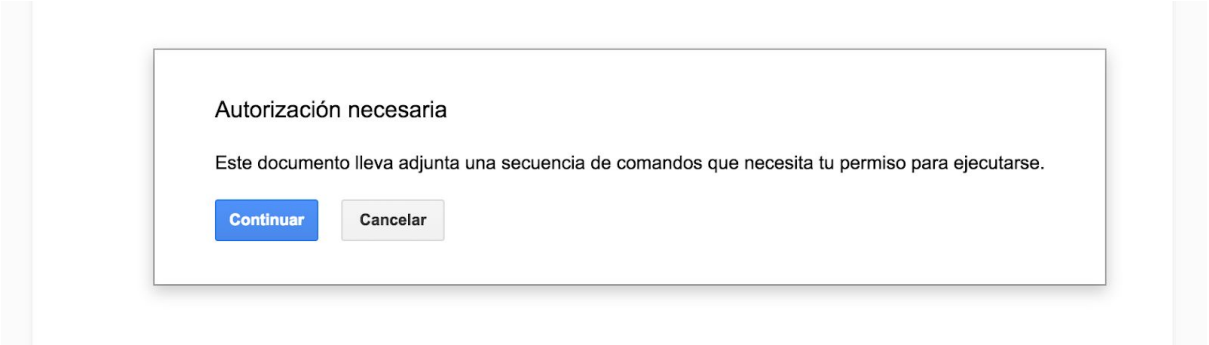
Para ello necesitaremos hacer ese HTML por lo que creamos un fichero nuevo, al que llamaremos de esa manera, y veremos que ya aparece con un mínimo de contenido. Añadimos un formulario y un botón sobre el que podamos hacer click para que formatee nuestro código.



Una nota importante es que el código del HTML se incrusta en forma de IFRAME en una barra lateral. El servidor coge el código y lo inyecta en dicho iframe. Esto hace que la comunicación fuera del IFRAME está contenida por las reglas de seguridad de HTML. Al fin y al cabo ese iframe se ejecuta en el mismo dominio que Google Docs.

Si recargamos el documento para probar el código que acabamos de desarrollar, Google detecta que vamos a incrustar una barra lateral, y nos pide permisos para poder

mostrarla.



Una vez se los concedemos, la barra se muestra, sin mucho estilo, la verdad.

Y si hacemos click en el botón no debería pasar nada pues aún no hemos puesto ningún listener que desencadene una acción. Sin embargo, si hacemos click, se abre una ventana nueva. ¿Y eso por qué?

Bueno, el modo IFRAME se basa en la utilidad de sandbox de HTML5 que añade ciertas restricciones, para que no haya problemas de seguridad. Y deben ejecutarse en el contexto *target="_self"* o *target="_top"*. Eso quiere decir que siempre que hagamos click, si no queremos cambiar de URL, debemos capturar el evento y parar su propagación antes de que abra una ventana nueva.

Para ello añadamos el siguiente código que evita esto mismo.

```
12  <script>
13
14    function preventFormSubmit() {
15      document
16        .querySelector('#f')
17        .addEventListener('submit', function(event) {
18          event.preventDefault();
19        });
20    }
21
22    window.addEventListener('load', preventFormSubmit);
23
24  </script>
```

Ya estamos en condiciones de dar un paso más.

4. Trabajando con el documento

Nuestro objetivo pasa por poder seleccionar un bloque de texto del documento, dar al botón, que lo capture, que lo transforme, y que reemplace el texto seleccionado, por el

nuevo formateado.

4.1. Capturando los párrafos seleccionados

En el lado del servidor, hacemos una función que a través del API de Google Docs obtenga qué texto del documento está seleccionado.

```
function getText(){
  var txt = "";
  var selection = DocumentApp.getActiveDocument().getSelection();
  if (selection){
    var rangeElements = selection.getRangeElements();
    for (i in rangeElements){
      txt += rangeElements[i].getElement().asText().getText() + " ";
    }
  }
  return txt;
}
```

Usamos un bucle *for... in* porque no todas las funcionalidades de ES6 están disponibles en el lado del servidor. Por ejemplo, el bucle *for... of* no funciona.

Con este código obtenemos la selección del documento activo. Puede que se hayan seleccionado varios elementos de distinto tipo. Los recorremos, pero sólo nos quedamos con los elementos que sean de tipo texto, y obtenemos su texto concatenado.

Las selecciones pueden ser parciales, es decir, pueden empezar a mitad de un párrafo y terminar a mitad de otro. Pero ello habría que comprobar si es parcial, y obtener el substring correspondiente. Pero nosotros no queremos eso. Nosotros queremos obtener párrafos enteros que podamos sustituir en bloque. Con este código nos vale para nuestro fin.

Pero ¿cómo lo invocamos? Está en el lado del servidor. Necesitamos una petición AJAX que desde el lado del cliente invoque a ese método, y nos devuelva su resultado. El API de App Script tiene eso resuelto:

```
google.script.run
          .withSuccessHandler(function(text,element){
             alert(text);
          })
          .withFailureHandler(function(err,element){
             alert(err);
          })
          .withUserObject(this)
```

```
            .getText();
```

Este script llama a *getText()* en el lado del servidor. Y si va bien, con lo que devuelva llama a *withSuccesshandler()* y si va mal a *withFailureHander()*. De momento, nos limitamos a mostrar en un *alert()* lo que nos devuelve la función *getText()*.

Veamos qué pasaría si seleccionamos texto a medias, y con una imagen en medio, ¿qué salida obtendríamos?

Perfecto. Tenemos lo que queremos: sólo coge el texto, y si seleccionas bloques parciales, da igual, recuperamos los párrafos completos. Ahora vamos a transformar el texto seleccionado.

4.2 Dar formato SQL: transformando el texto seleccionado.

Para indentar las queries con un formato vamos a utilizar la librería SQL Formatter, que permite dar formato a varios dialectos SQL: IBM DB2, CouchDB N1QL, Oracle PL/SQL y por supuesto SQL estándar. Está liberado con licencia MIT.

Aquí cabría la pregunta de cómo incorporar esta librería a nuestro componente. Tenemos dos opciones:

- Distribuirla con el componente
- Referenciarla desde un CDN.

Cada una tiene sus pros y sus contras.

Por un lado, si la cargamos desde el componente, como el IFRAME se crea al vuelo, y la URL es un hash, cabe preguntarse si el hash siempre será el mismo. Si no es así, como creo que pasa, las librerías javascript no podrán cachearse debidamente en el

navegador. Sin embargo, si usamos un CDN sí.

Pero por otro lado, si queremos usar estos componentes sin conexión, si ponemos la librería fuera del componente, no sé si estará disponible cuando lo usemos *"sin conexión"*. Son cosas a tener en cuenta, y que habría que comprobar antes de tomar una decisión.

Para avanzar, vamos a optar por la versión de distribuirlo con el componente.

Para ello, creamos un nuevo fichero HTML llamado *sql-formatter.min.js.html* y borraremos el contenido por defecto. Añadimos al fichero html vacío las etiquetas *<script>* y cerramos la etiqueta con *</script>*. Entre ambas etiquetas copiamos y pegamos el contenido de *sql-formatter.min.js* y guardamos.

Ahora, vamos a implementar una función en *main.gs* que se llamará *include*.

```
function include(filename) {
  return HtmlService.createHtmlOutputFromFile(filename)
      .getContent();
}
```

Esta función cogerá el contenido de un HTML y lo podemos inyectar en nuestra vista. En nuestro caso concreto queremos tomar el contenido de *sql-formatter.min.js.html* e inyectarlo en *sidebar.html*

```
<?!= include('sql-formatter.min.js.html'); ?>
```

Ahora ya estaríamos en posición de poder usar la librería desde nuestra vista:

```
var formattedText = sqlFormatter.format(text);
```

Y actualizamos el código de nuestra función *format()* a lo siguiente:

```
      //obtiene el texto seleccionado y lo formatea
      function format(e){
        this.disabled = true;
        google.script.run
          .withSuccessHandler(function(text,element){
             var formattedText = sqlFormatter.format(text);
             insertText(formattedText);
             element.disabled = false;
          })
          .withFailureHandler(function(err,element){
             alert(err);
             element.disabled = false;
```

```
        })
        .withUserObject(this)
        .getText();
    }
```

Si estudiamos el código anterior repararemos en que se llama a una función *insertText()* que recibe por parámetro el nuevo texto ya formateado.

4.3 Reemplazando el texto seleccionado

Ahora que ya tenemos el texto formateado guardado en una variable, tenemos que reemplazar el texto seleccionado con este valor. Para ello creamos en la parte cliente una función llamada insertText() que llamará a una función en el servidor llamada homónimamente que reemplazará el texto seleccionado con el texto formateado.

```
    //funcion que reemplaza la query por la query formateada
    function insertText(formattedText) {
      this.disabled = true;
      console.log(formattedText);

      google.script.run
        .withSuccessHandler(function(returnSuccess, element) {
          element.disabled = false;
         })
        .withFailureHandler(function(msg, element) {
          alert(msg);
          element.disabled = false;
        })
        .withUserObject(this)
        .insertText(formattedText);
    }
```

Y su homóloga en el lado del servidor que la hemos copiado tal cual del [ejemplo del Traductor de Google Docs]

```
function insertText(newText) {
  var selection = DocumentApp.getActiveDocument().getSelection();
  if (selection) {
    var replaced = false;
    var elements = selection.getSelectedElements();
    if (elements.length === 1 && elements[0].getElement().getType() ===
        DocumentApp.ElementType.INLINE_IMAGE) {
```

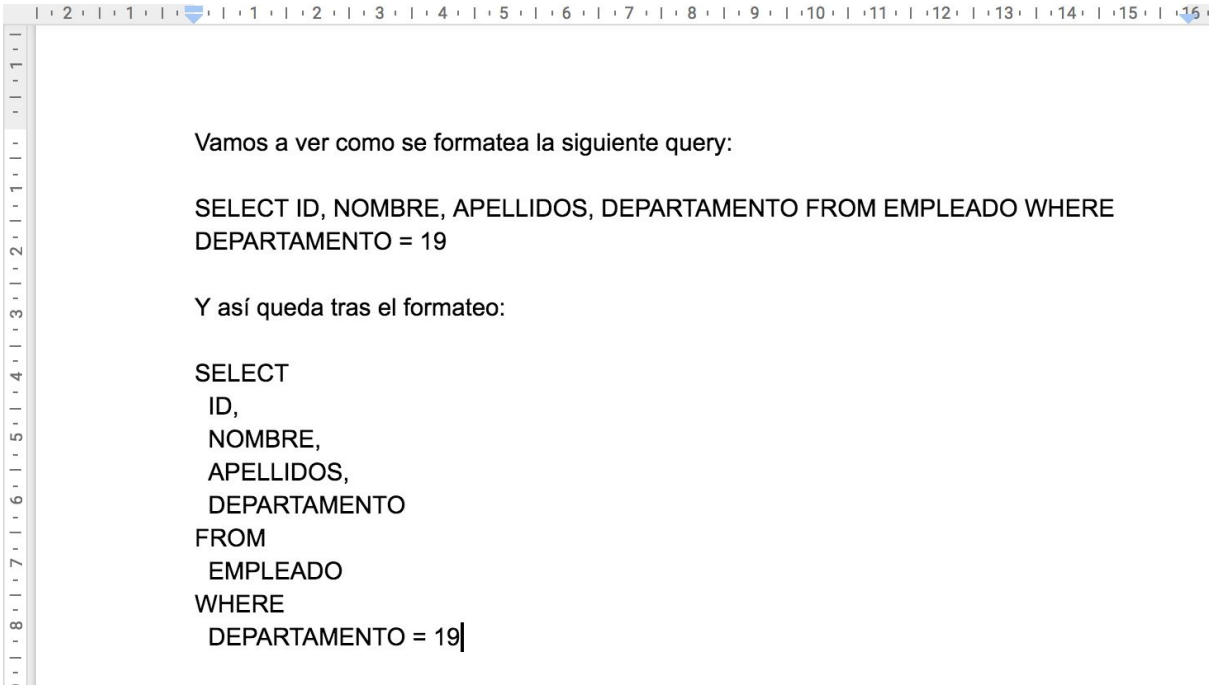
```
      throw "Can't insert text into an image.";
    }
    for (var i = 0; i < elements.length; ++i) {
      if (elements[i].isPartial()) {
        var element = elements[i].getElement().asText();
        var startIndex = elements[i].getStartOffset();
        var endIndex = elements[i].getEndOffsetInclusive();
        element.deleteText(startIndex, endIndex);
        if (!replaced) {
          element.insertText(startIndex, newText);
          replaced = true;
        } else {
          // This block handles a selection that ends with a partial
element. We
          // want to copy this partial text to the previous element so
we don't
          // have a line-break before the last partial.
          var parent = element.getParent();
          var remainingText = element.getText().substring(endIndex + 1);

parent.getPreviousSibling().asText().appendText(remainingText);
          // We cannot remove the last paragraph of a doc. If this is
the case,
          // just remove the text within the last paragraph instead.
          if (parent.getNextSibling()) {
            parent.removeFromParent();
          } else {
            element.removeFromParent();
          }
        }
      } else {
        var element = elements[i].getElement();
        if (!replaced && element.editAsText) {
          // Only translate elements that can be edited as text,
removing other
          // elements.
          element.clear();
          element.asText().setText(newText);
          replaced = true;
        } else {
          // We cannot remove the last paragraph of a doc. If this is
the case,
          // just clear the element.
          if (element.getNextSibling()) {
            element.removeFromParent();
```

```
          } else {
            element.clear();
          }
        }
      }
    }
  } else {
    var cursor = DocumentApp.getActiveDocument().getCursor();
    var surroundingText = cursor.getSurroundingText().getText();
    var surroundingTextOffset = cursor.getSurroundingTextOffset();

    // If the cursor follows or preceds a non-space character, insert a
space
    // between the character and the translation. Otherwise, just insert
the
    // translation.
    if (surroundingTextOffset > 0) {
      if (surroundingText.charAt(surroundingTextOffset - 1) != ' ') {
        newText = ' ' + newText;
      }
    }
    if (surroundingTextOffset < surroundingText.length) {
      if (surroundingText.charAt(surroundingTextOffset) != ' ') {
        newText += ' ';
      }
    }
    cursor.insertText(newText);
  }
}
```

Con esto ya somos capaces de indentar y formatear código SQL en nuestro documento.



Vamos a ver como se formatea la siguiente query:

SELECT ID, NOMBRE, APELLIDOS, DEPARTAMENTO FROM EMPLEADO WHERE DEPARTAMENTO = 19

Y así queda tras el formateo:

```
SELECT
  ID,
  NOMBRE,
  APELLIDOS,
  DEPARTAMENTO
FROM
  EMPLEADO
WHERE
  DEPARTAMENTO = 19
```

5. Conclusión

Estaría muy bien, que éste código se completase, dando algo de estilo “material” al HTML de forma que el look & feel fuese acorde con una aplicación de Google Docs.

Otra variante que se le podría añadir, es que además se resaltase el color del texto por sintaxis, pero eso haría muy largo este post. Sin embargo, ya existe un plugin que hace eso con bastante acierto llamado code-blocks, y que recomiendo estudiar su código a aquellos que quieran profundizar.

Tras alcanzar un grado de acabado, a lo mejor nos interesaría liberar nuestro componente a la comunidad. Google pone a nuestra disposición una extensa documentación de cómo publicarlo.

6. Referencias.

- El repositorio donde puedes encontrar el código de este artículo.
- Código fuente del componente code-blocks
- Información sobre Google App Script
- Este mismo artículo en el portal Adictos al Trabajo