



**GRADO EN INGENIERÍA DE SISTEMAS AUDIOVISUALES Y
MULTIMEDIA**

Curso Académico 2019/2020

Trabajo Fin de Grado

**IMPLEMENTACIÓN DE UN PLUGIN DE VISUALIZACIÓN EN
REALIDAD VIRTUAL EN KIBANA**

Autor: Andrea Villaverde Hernández

Tutor: Dr. Jesús María Gonzalez Barahona

Trabajo Fin de Grado

IMPLEMENTACIÓN DE UN PLUGIN DE VISUALIZACIÓN EN REALIDAD VIRTUAL EN KIBANA

Autor : Andrea Villaverde Hernández

Tutor : Dr. Jesús Mará Gonzalez Barahona

La defensa del presente Proyecto Fin de Grado se realizó el día de
de 2020, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 2020

*Dedicado a
todos aquéllos que
nunca se rindieron.*

Agradecimientos

Gracias a...

«*If you can dream it, you can do it.*» - Walter Elias Disney

Resumen

Este proyecto...

Summary

This...

Índice general

1. Introducción	1
1.1. El problema	1
2. Contexto y tecnologías utilizadas	3
2.1. HTML5	3
2.1.1. Definición	3
2.1.2. En este proyecto	3
2.2. CSS	3
2.2.1. Definición	3
2.2.2. En este proyecto	4
2.3. JavaScript	4
2.3.1. Definición	4
2.3.2. En este proyecto	4
2.4. A-Frame	4
2.4.1. Definición	4
2.4.2. THREE.JS	6
2.4.3. En este proyecto	6
2.5. ELK	6
2.5.1. Definición	6
2.5.2. Elasticsearch	6
2.5.3. Logstash	7
2.6. Kibana	8
2.6.1. Definición	8
2.6.2. En este proyecto	8

2.7.	NodeJS y NPM	9
2.7.1.	Definición	9
2.7.2.	En este proyecto	9
2.8.	BabiaXR	9
2.8.1.	Definición	9
2.8.2.	En este proyecto	9
3.	Desarrollo	11
3.1.	Metodología SCRUM	11
3.2.	Sprint 0: Investigación y Estudio Preliminar	11
3.2.1.	Testeando Cliente de Kibana	12
3.2.2.	Creando el Entorno de Desarrollo	15
3.2.3.	Testeando Biblioteca para las Visualizaciones	17
3.3.	Sprint 1: Primeros Plugins de Visualización	18
3.3.1.	Plugin Simple sin Datos	19
3.3.2.	Implementación de A-Frame sin Datos	24
3.3.3.	Creación de Componente A-Frame	26
3.4.	Sprint 2: Visualización con datos Elasticsearch	30
3.4.1.	Editor y Schemas	30
3.4.2.	Plugin Simple con datos	33
3.4.3.	Integración de A-frame con datos	36
3.5.	Sprint 3: Implementación de BabiaXR	39
3.5.1.	Visualización Pie Chart sin Datos	40
3.5.2.	Visualización Pie Chart con datos	45
3.6.	Sprint 4: Nuevas Visualizaciones	48
3.6.1.	Visualización Simple Bar	48
3.6.2.	Visualización 3D Bars	54
3.6.3.	Visualización Bubbles	60

Índice de figuras

Capítulo 1

Introducción

1.1. El problema

Capítulo 2

Contexto y tecnologías utilizadas

2.1. HTML5

2.1.1. Definición

Es la quinta versión principal de HTML, un lenguaje de etiquetas (también llamado lenguaje de marcado) para la elaboración de páginas web. Las siglas vienen de HyperText Markup Language y la versión definitiva se publicó en octubre de 2014. Cada etiqueta es un elemento que puede tener atributos y contenido. Estas etiquetas pueden ser de apertura ej: `<body>`, o cierre `</body>`. El estándar está a cargo del consorcio W3C (World Wide Web Consortium) o WWW. Todos los navegadores actuales han adoptado este lenguaje para la visualización de páginas web. En HTML5 existen la variante “clásica” de sintaxis para HTML (`text/html`) y la sintaxis XHTML (`application/xhtml+xml`), que se han desarrollado en paralelo.

2.1.2. En este proyecto

2.2. CSS

2.2.1. Definición

Sus siglas vienen de “Cascading Style Sheets” (hojas de estilo en cascada). Es un lenguaje utilizado para definir la presentación visual de los elementos en una página web. CSS nos permite por tanto modificar la apariencia por defecto de las etiquetas HTML. También permite crear

identificadores, clases y selectores que recojan un conjunto de reglas, luego podremos aplicar dicha clase al elemento (o elementos) que queramos. El estándar CSS también es mantenido por el consorcio W3C y es compatible con todos los navegadores web modernos.

2.2.2. En este proyecto

2.3. JavaScript

2.3.1. Definición

Es un lenguaje de programación orientado a objetos, ligero e interpretado que proviene del estándar ECMAScript. El tipo de las variables está ligado a su valor. La sintaxis es similar a C aunque adopta nombres y convenciones de Java (aunque con semánticas y propósitos diferentes). Se utiliza principalmente en desarrollo web y se ejecuta en el navegador del lado del cliente. Permite la realización de páginas web dinámicas y mejorar la interactividad de la experiencia del usuario. Hoy en día es muy habitual su uso para enviar y recibir información entre el servidor y el cliente de forma asíncrona utilizando la tecnología AJAX. Todos los navegadores modernos tienen un intérprete de JS.

2.3.2. En este proyecto

2.4. A-Frame

2.4.1. Definición

A-frame es un framework web que permite la creación de experiencias VR (Realidad Virtual). Desarrollado por Mozilla pensado para implementar contenido VR a nuestra web de manera sencilla, sin la necesidad de instalar nada. Se trata de un proyecto de Código Abierto, por lo que ha sido muy bien recibido en las comunidades VR.

A primera vista, A-frame parece de fácil manejo; pues permite la creación de escenarios 3D utilizando simples etiquetas HTML. Pero no todo esto se queda aquí, pues es un poderoso framework de entity-component que viene dado por su fichero three.js.

A-Frame soporta dispositivos de VR como Vive, Rift, Daydream, Gear VR o Cardboard. Además, gracias a dispositivos de tracking y controladores de posición, permite sumergirse en experiencias VR en escenarios a 360º.

Características

- Crea VR de forma sencilla: simplemente utilizando las etiquetas «script» y «a-scene» podrás crear un escenario 3D con toda la configuración para VR, de forma predeterminada, sin la necesidad de instalar nada.
- Declaraciones en HTML: al estar basado en HTML es muy fácil de usar ya seas desarrollador web, amante del VR, artista, diseñador, educador o ni no.
- VR Multiplataforma: permite usar los diferentes dispositivos con sus respectivos controladores. En caso de no disponer de ninguno, también se puede usar en portátiles, tablets o teléfonos móviles.
- Arquitectura Entity-Component: A-frame es un poderoso framework donde se provee de una poderosa estructura entity-component. Al tratarse de HTML, los desarrolladores tienen acceso ilimitado a javascript, DOM API, three.js, WebVR y WebGL.
- Rendimiento: A-Frame está optimizado desde cero para WebVR. Como A-Frame usa el DOM, sus elementos no tocan el motor del navegador. Los objetos 3D se actualizan en la memoria con una sola llamada “requestAnimationFrame”.
- Tool Agnostic: como la web se crea en HTML, A-frame es compatible con la mayoría de las bibliotecas y herramientas web tales como React, Preact, Vue.js, d3.js, Ember.js y jQuery.
- Inspector Visual: A-frame proporciona un visor 3D incorporado. En la que permite abrir la escena 3D y modificar algunos de sus elementos.
- Registro: al igual que Unity Assets Store, a-Frame recopila componentes para que los desarrolladores puedan publicar y buscarlos de forma sencilla.
- Componentes: con a-Frame se puede correr geometrías, luces, materiales, animaciones, modelos, sombras, audios, texto, etc (además de los controladores para los dispositivos).

Además de, gracias a su comunidad, sistemas de partículas, físicas, multijugador, aguas, montañas, reconocimiento de voz y un gran etcétera.

2.4.2. THREE.JS

Es una biblioteca escrita en Javascript que permite la creación y visualización de objetos 3D en entornos web. Es muy convenientes pues permite utilizarse en conjunto con Canvas (HTML5) SVG y WebGL. Por lo que podemos decir que es compatible con cualquier navegador que soporte WebGL.

Además, permite importar modelos 3D, en formato JSON, creados en Maya, Blender o Max3D.

2.4.3. En este proyecto

A-Frame supone una parte importante de este proyecto, pues el objetivo de este proyecto es crear un plugin que permita dar una experiencia VR a la hora de representar la visualización de los datos mostrados en Kibana.

2.5. ELK

2.5.1. Definición

Es un conjunto de herramientas de gran potencial que ayuda con la administración de registros, permitiendo monitorizar, consolidar y analizar logs (no siempre son logs) generados en distintos servidores.

Estas herramientas son: Elasticsearch, Logstash y Kibana. Las tres se complementan entre sí pero, se pueden utilizar de forma independiente.

2.5.2. Elasticsearch

Se trata de un motor de búsqueda y análisis fácilmente escalable. Permite almacenar, buscar y analizar grandes volúmenes de datos casi en tiempo real. Se puede acceder de forma sencilla gracias a su elaborada API.

Está escrito en Java, de código abierto y generalmente utilizado con fines empresariales o de investigación.

Características

- Documentos: está orientado a documentos que se insertan en formato JSON, son esquemas sin indexar. Lo que permite una búsqueda mucho más rápida.
- API: cuenta con una potente API muy fácil de usar. Ésta permite hacer peticiones de tipo HTTP.
- Rapidez: Gracias a su distribución de escalado dinámico; elasticsearch encuentra rápidamente cualquier consulta que se le haga, incluso cuando tenemos grandes cantidades de datos. Ya sean búsquedas simples, como complejas.
- Gran componente: Elasticsearch junto con Kibana y Logstash forman un conjunto de herramientas perfecta para la recopilación, análisis y visualización de datos.
- En tiempo real: Las actualizaciones de los índices de elasticsearch se realizan de manera tan rápida que prácticamente se puede consultar en tiempo real.

En este proyecto

Para este proyecto, no es una parte importante; pues solo la utilizaremos para indexar los datos de prueba que vamos a visualizar posteriormente en Kibana.

2.5.3. Logstash

Es la herramienta encargada de recolectar los logs de una aplicación, parsearlos; traducirlos y pasarlo a formato JSON para luego poder almacenarla en elasticsearch.

Esta parte no la utilizaremos en este proyecto en ningún momento.

2.6. Kibana

2.6.1. Definición

Es una plataforma que permite visualizar los datos almacenados en elasticsearch, para su posterior monitorización y análisis de estos desde el propio navegador web.

Al tratarse de código abierto, la propia empresa invita a que desarrolladores puedan contribuir con su mejoría o a la creación, como en nuestro caso, de plugins para personalizarlos al gusto del usuario.

Características

- Visualizaciones: podemos encontrar representaciones con histogramas, gráficas de tiempo, roscos o tablas que nos permite visualizar e interactuar con los datos almacenados en elasticsearch.
- Datos en tiempo real: la buena conectividad entre ellos, permite visualizar y buscar la información unos pocos segundos después de ser introducida en elasticsearch.
- Dashboards: que recogen las visualizaciones en paneles para poder tener una vista global y así poder entender mejor grandes cantidades de datos.
- Geolocalización: en caso de tener datos de ubicaciones; ésta te muestra las distintas coordenadas en mapas.
- Extras: también incluyen extras como timeseries, graphs o machine learning.

2.6.2. En este proyecto

Esta es la base de dicho proyecto, pues lo que queremos es crear un plugin que permita modificar los distintos tipos de visualizaciones en formato 3D, aportando esa experiencia en VR que tanto queremos conseguir.

2.7. NodeJS y NPM

2.7.1. Definición

También llamado “node package manager”. Es un gestor de paquetes para node.js. Nos permite instalar y desinstalar dependencias y aplicaciones que se encuentran en el repositorio desde la línea de comandos. Está escrito en javascript y viene incluído por defecto en todas las versiones de node.js desde la 0.6.3.

2.7.2. En este proyecto

2.8. BabiaXR

2.8.1. Definición

2.8.2. En este proyecto

Capítulo 3

Desarrollo

3.1. Metodología SCRUM

3.2. Sprint 0: Investigación y Estudio Priliminar

Antes de comenzar con el desarrollo, debemos estudiar un poco el funcionamiento de las herramientas con las que vamos a trabajar. Por eso, empezaremos con este sprint, que llamaremos sprint 0 porque uno será un sprint en sí.

Primero será establecer las herramientas que necesitaremos conocer para llevar a cabo el desarrollo. Como nuestro objetivo es crear un plugin para visualizar gráfica es VR para Kibana, lo ideal sería que conociéramos un poco el funcionamiento de ésta.

Luego crearemos un entorno de desarrollo para Kibana y documentarse cómo se crean plugins para Kibana.

Por último, jugaremos un poco con la biblioteca Babiaxr para conocer su funcionamiento.

En resumen:

1. Testearemos el cliente de Kibana.
2. Crearemos un entorno de desarrollo.
3. Testearemos la biblioteca Babiaxr.

3.2.1. Testeando Cliente de Kibana

Como se ha mencionado antes, lo primero que haremos será testear el cliente de Kibana. Lo que haremos será instalar un elasticsearch, que usaremos para que nos proporcione los datos; y un cliente de Kibana para que exploremos su funcionamiento.

Estas dos herramientas se pueden conseguir de forma gratuita en la web oficial de Elastic. Descargamos las últimas versiones oficiales (versión 7.6) y las instalamos en el propio ordenador. Un dato a tener en cuenta es que ambos deben ser de la misma versión, sino no funcionará (o no funcionará correctamente). Para una primera vez, lo mejor es no tocar la configuración de ninguno de los dos programas.

Anteriormente se mencionó que kibana era una manera de monitorizar los datos recopilados en elasticsearch, pero aunque no tengamos ninguna base de datos en elasticsearch, Kibana nos permitirá descargar unas bases de datos de prueba para que podamos utilizarlo en nuestro testeo. Para esta ocasión utilizaremos la base de datos de eCommerce.

Lo primero que nos encontramos es la pestaña “Discover” que nos muestra una lista de todos los items que hay en la base de datos (en este caso, las ventas que se han realizado). A la izquierda, tenemos un editor que nos permite filtrar los campos que queremos que aparezca en el listado. Además, en la parte de arriba, justo encima de la tabla de datos, hay una gráfica que nos muestra una visualización gráfica de los datos que hemos filtrado, para esta ocasión el número de ventas que se realizan a lo largo del día.

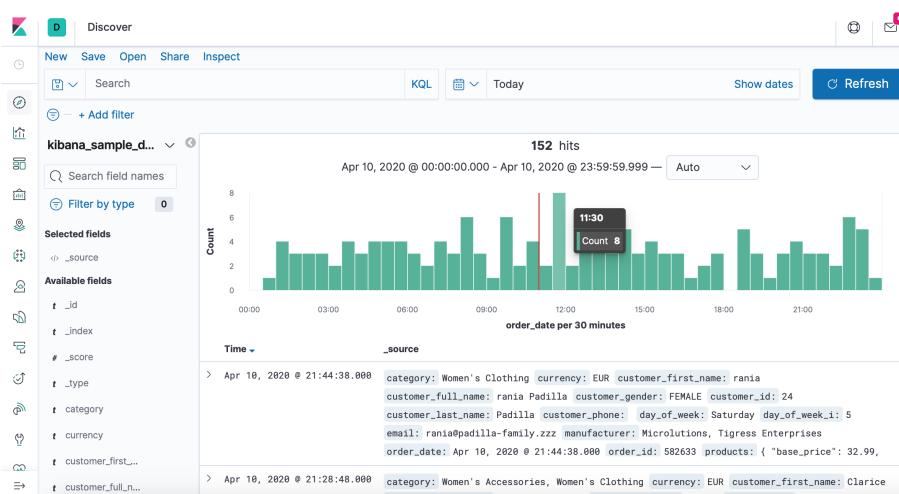


Figura 3.1: Pestaña Discover del cliente de Kibana

Una vez que ya hemos comprendido, viéndolo en la pestaña de “Discover”, con que clase de datos trabajamos; pasaremos a ver cómo se crean las visualizaciones. Para ello, iremos a la pestaña “Visualization” y lo primero con lo que nos encontramos es con que, al habernos descargado datos de prueba, ya vienen unas predeterminadas. Antes de ponernos a probar a crear una, lo mejor será coger una de estas predeterminada para entender su funcionamiento.

Esta visualización nos muestra una gráfica tipo Pie que viene configurada para que nos muestre el número de ventas en base al género del cliente, ya sea hombre (MALE) o mujer (FEMALE). A mano izquierda vemos que existe también un editor que nos permite configurar la gráfica. Existen dos pestañas dentro de la misma:

- Data: donde configuraremos qué datos queremos que nos muestre, separados entre metrics y buckets (que explicamos más adelante).
- Options: como su propio nombre indica, configuraciones opcionales. Por ejemplo, en este caso te permite indicarle donde se desea colocar la leyenda, o si queremos que la gráfica sea rellena o no.

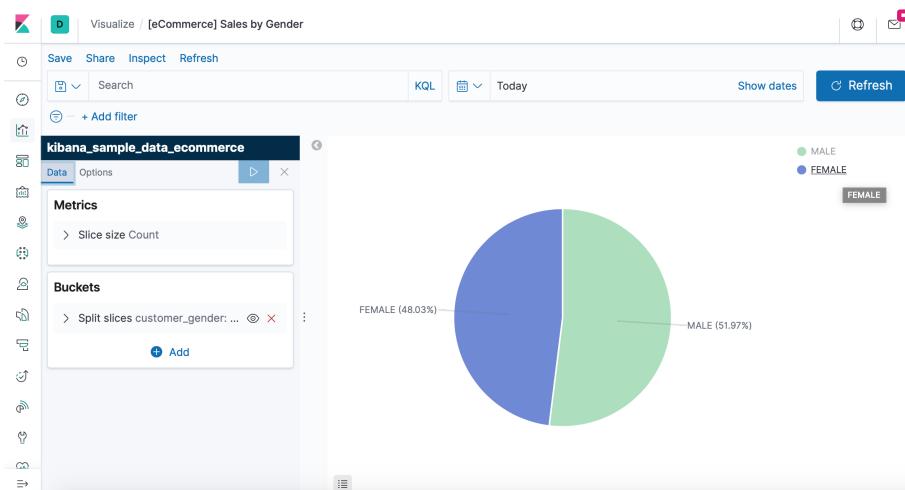


Figura 3.2: Visualizacion Pie en el cliente de Kibana

Ahora que ya entendemos un poco el funcionamiento de las visualizaciones, pasaremos a crear una propia; pero que sea diferente de la anterior. Por ejemplo, nos interesa ver desde qué países se han realizado las ventas. Elegimos “Create Visualization” y seleccionamos, para este caso, la visualización tipo Table.

Como no nos queremos complicar mucho, lo único que haremos será pedirle que nos muestre la cantidad de ventas que se han realizado en base a los países. Para ello, vamos en la pestaña “Data” y en metrics le indicamos “Count” que nos mostrará la cantidad de ventas. Y luego “Buckets” que nos los clasifique por país. Para ello, se ha de seleccionar en “Aggregations” la opción de “Terms” y luego en “Field” le indicamos que queremos que sea por nombre de país “geoip.region_name”. Además, nos permite que lo podamos ordenar de mayor a menor o viceversa; o que solo nos muestre, por ejemplo, los 5 primeros. O incluso, darle una etiqueta a los datos que estamos mostrando.

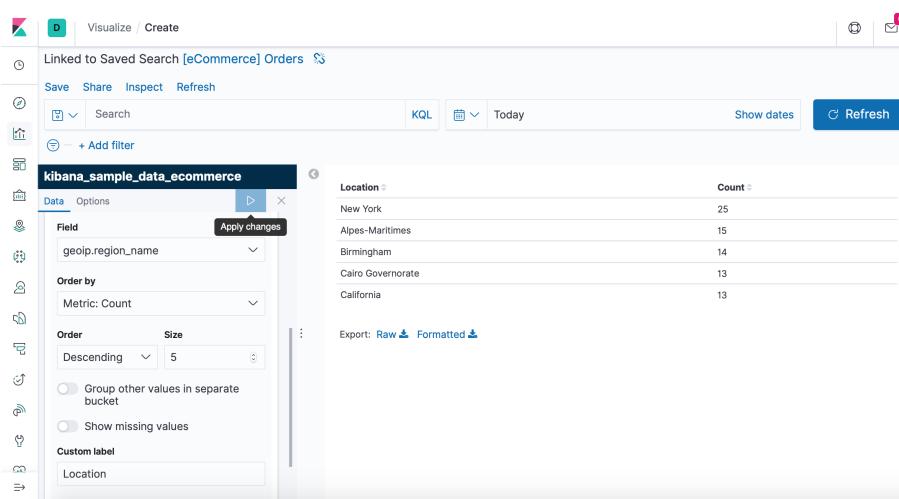


Figura 3.3: Creación de Vizualización Table en el cliente de Kibana

Una vez seleccionado cómo queremos nuestra visualización, solo tenemos que darle a “Apply changes” y se actualizará al instante. Y una vez que tengamos la visualización que queremos, simplemente la guardamos dándole a “Save”.

Después de haber probado alguna visualización y haber creado una propia, nos interesa conocer otra sección interesante y es la que encontramos en la pestaña “Dashboard”. En esta, nos encontramos con que ya existe una predeterminada así que, aunque podemos crear una nueva, la editaremos.

Para entenderlo bien, la dashboard nos muestra en una misma página todas las visualizaciones que tenemos creadas y que queremos que sean mostradas. Lo que nos permite en edición es añadir o eliminar visualizaciones, colocarlas en un sitio u otro; o incluso darles el tamaño que nos interese.

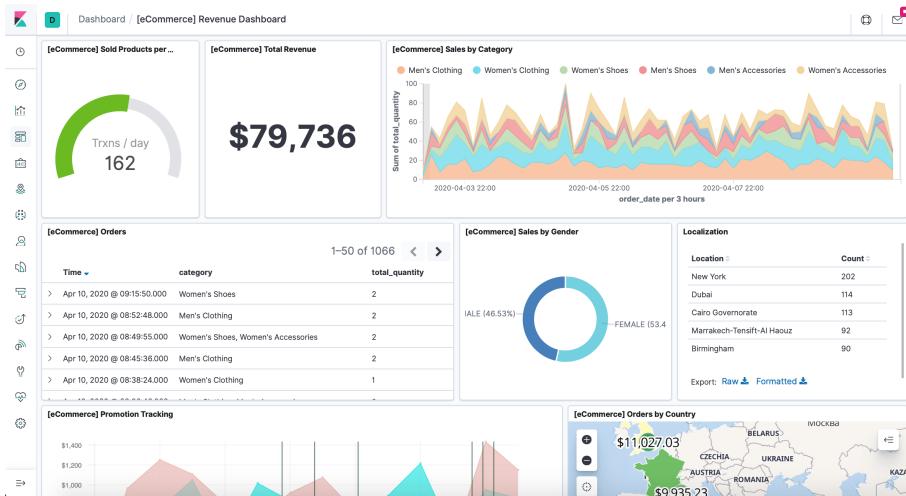


Figura 3.4: Dashboard en el cliente de Kibana

3.2.2. Creando el Entorno de Desarrollo

Ahora que ya entendemos un poco el funcionamiento del cliente de Kibana, pasaremos a crearnos el entorno de desarrollo que utilizaremos para crear nuestro plugin. Este nos permitirá ver en tiempo real los cambios que realicemos.

Siguiendo las indicaciones que podemos encontrar en la sección correspondiente para contribuir en el proyecto (“CONTRIBUTING.md”), dentro del repositorio de kibana.

Una vez tengamos creado nuestro entorno de desarrollo, pasaremos a documentarnos cómo se crean plugin para Kibana. Para ello, elastic no proporciona artículos en su blog de como crear unos plugins básicos. Aunque está un poco obsoleta la documentación, únicamente nos quedaremos con la parte de cómo es la estructura de ficheros.

Por el momento no vamos a detallar el contenido de los ficheros (lo explicaremos más adelante), simplemente nos enfocaremos en qué función tiene cada uno.

Lo primero que haremos será crear un directorio con en nombre de nuestro plugin en “KIBANA_HOME/plugins”. Dentro de esta habrá dos ficheros:

- package.json: un simple fichero para npm donde le indicamos el nombre de nuestro plugin, version, autor, etc. Y sobretodo, las dependencias que necesitamos que nos instale para poder utilizarlas en nuestro proyecto.
- index.js: este es el módulo principal. En él registramos nuestro plugin indicando el tipo

de plugin (dentro de kibana) que es, visualización en este caso; y hacemos referencia al fichero principal.

A esto le sigue crear un nuevo directorio con el nombre “public/” que contendrá todos los ficheros de nuestro plugin (controllers, configurations, templates, etc). Antes hemos dicho que desde “index.js” hacíamos referencia a el fichero principal de nuestro plugin, pues es aquí donde lo crearemos.

- public/plugin-name.js: para el caso de las visualizaciones es donde registramos las diferentes visualizaciones, puede ser una o varias, que vamos a añadir en kibana a través de nuestro plugin.

Todos los ficheros anteriormente mencionados son los principales para el funcionamiento de esto; pero a mayores se le pueden añadir más ficheros. Algunos ejemplos opcionales:

- public/plugin-name-controller.js: como su propio nombre indica, sería el controlador de nuestro plugin. En el contiene toda la parte lógica de nuestra visualización.
- public/plugin-name.html: template del plugin.
- public/plugin-name.less: hoja de estilo del plugin.
- public/options-template.html: template para la pestaña “options” del editor.

De forma más visual, podemos decir que la estructura de ficheros para crear un plugin para kibana es la siguiente:

```
KIBANA_HOME/plugins/
|---plugin-name/
|   |---package.json
|   |---index.js
|   |---public/
|       |---plugin-name.js
|       |---plugin-name-controller.js
|       |---plugin-name.html
|       |---plugin-name.less
|       |---options-template.html
```

3.2.3. Testeando Biblioteca para las Visualizaciones

Al igual que necesitamos conocer el funcionamiento de kibana, también necesitamos tener unos conceptos básicos sobre los componentes visuales que vamos a incorporar para nuestras visualizaciones para kibana.

La opción de usar Babiaxr ha sido muy acertada, ya que permite crear gráficas, a partir de datos proporcionados, dentro de un escenario webVR. Así que antes de nada, analizaremos lo que nos ofrece y cómo se crean.

Dentro del repositorio de babiaxr podemos ver los distintos tipos de componentes que tienen creados. Dichas componentes han sido creadas a partir del framework a-frame (ver ...).

Lo primero que haremos será probar los diferentes componentes que tienen: “geopiechart”, “geosimplebarchart”, “geo3dbarchart” y “geobubblechart”. Todas ellas se pueden probar desde su web.

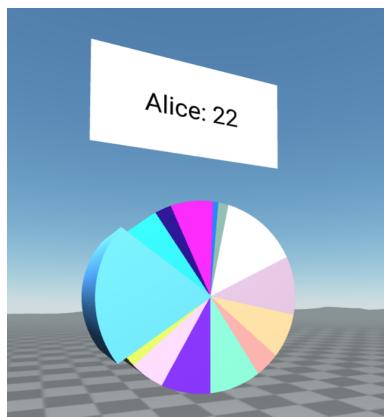


Figura 3.5: Geopiechart

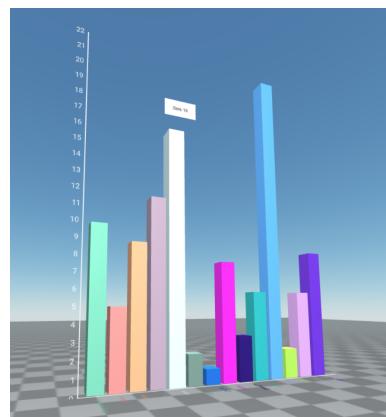


Figura 3.6: Geosimplebarchart

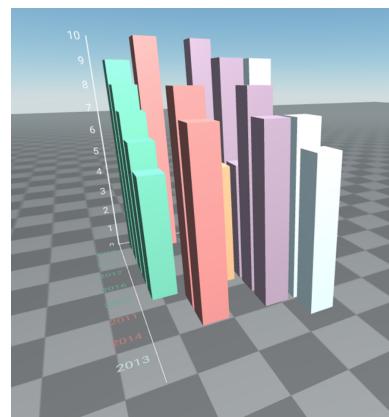


Figura 3.7: Geo3dbarchart

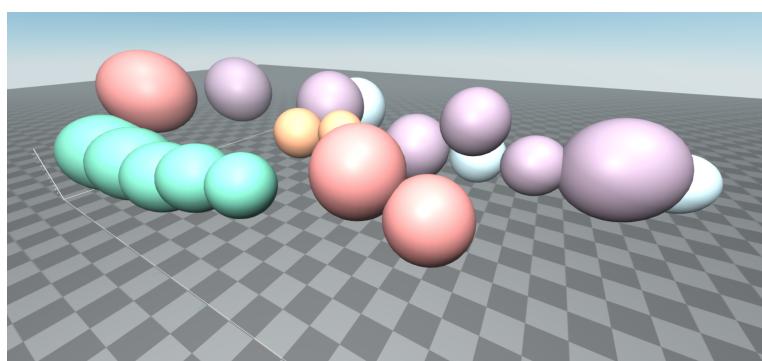


Figura 3.8: Geobubblechart

Una vez que hemos visto lo que hace, probaremos a crear nuestro propio escenario con la componente “geopiechart” con datos que le daremos de forma manual. Para ello, nos apoyaremos en la documentación que nos facilita su autor.

```
<a-entity geopiechart='legend: true;
  data: [{"key": "Andrea", "size": 9}, {"key": "URJC", "size": 8},
  {"key": "ETSIT", "size": 7}, {"key": "kibana", "size": 6},
  {"key": "elasticsearch", "size": 5}, {"key": "Jesus", "size": 8},
  {"key": "David", "size": 7}]
  position="-3 5 15" rotation="90 0 0"></a-entity>
```

Obteniendo como resultado:

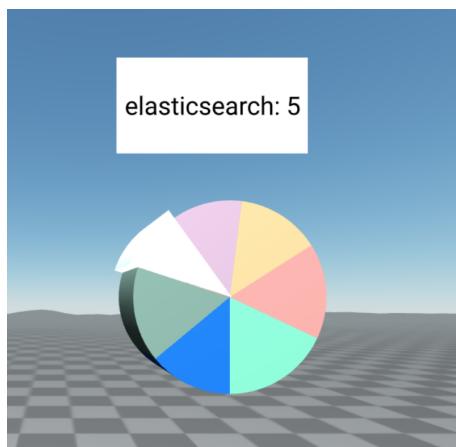


Figura 3.9: Pie Chart de Prueba

Con todo esto, ya podemos proceder a crear nuestro primer plugin.

3.3. Sprint 1: Primeros Plugins de Visualización

Ahora que conocemos las herramientas con las que trabajamos, es hora de ponernos a crear nuestro primer plugin. Esta parte es vital para el aprendizaje, así que iremos paso a paso.

Lo primero que haremos será crear un plugin simple, sin usar datos, que nos muestre un “Hola Mundo” como visualización. Esto nos ayudará a conocer más en profundidad el funcionamiento de la estructura anteriormente explicada.

Una vez que hayamos conseguido crear este primer plugin, intentaremos implementar una figura creada con a-frame. Esto nos ayudará a entender cómo podemos añadir componentes a-frame a nuestras visualizaciones.

También intentaremos crear un componente sencillo de a-frame y lo implementaremos. Esto nos ayudará a entender cómo se han creado los componentes de babiaxr.

En resumen:

1. Crearemos un plugin de visualización sin datos.
2. Implementaremos A-frame sin usar datos.
3. Crearemos un componente A-Frame y lo implementaremos.

3.3.1. Plugin Simple sin Datos

Anteriormente (ver Capítulo) explicamos cómo era la estructura base para crear plugins para Kibana. Iremos parte por parte hasta conseguir tener una visualización que simplemente nos muestre un “Hola Mundo”.

A este primer plugin lo llamaremos “hello-world-vis” y comenzaremos a programar los primeros archivos que nos permitirá registrar el plugin e instalar las dependencias npm que necesitamos.

```
{  
  "name": "hello-world-vis",  
  "version": "0.0.1",  
  "kibana": {  
    "version": "kibana"  
  }  
}
```

Parámetros utilizados:

- name: nombre del plugin.
- version: versión de nuestro plugin.
- kibana: versión de kibana.

Como solo es una versión de prueba, únicamente indicaremos el nombre del plugin, la versión y para que version de kibana es. Esta última se debe indicar bien, sino no funcionará. Por eso, como estamos desarrollando para la version 7.6, esto tiene que venir reflejado en nuestro “package.json”. En esta ocasión, al ser un simple hola-mundo no necesitaremos instalar ninguna dependencia.

Ahora procederemos a registrar nuestro plugin en “index.js”:

```
export default function (kibana) {
  return new kibana.Plugin({
    uiExports: {
      visTypes: [
        'plugins/hello-world-vis/hello-world-vis'
      ]
    }
  });
}
```

Parámetros utilizados:

- uiExports: indica que el tipo de modulo vamos a exportar. Dentro de esta existen varios tipos: (hacks, visTypes, fieldformats, etc).
- vis_Types: indicamos que el módulo que vamos a exportar se trata de un plugin de visualización.

Una vez dentro de “vis_Types” indicaremos el directorio donde se encuentra el archivo principal de nuestro plugin “public/hello-world-vis.js”. Ahora pasaremos a crearlo. Al igual que para los plugins, las visualizaciones también deben registrarse.

Lo primero que haremos será importar el paquete dentro de kibana que nos permitirá registrar nuestra visualización. Debido a la migración que se está realizando en kibana, la documentación que facilitaban y cualquier otro tutorial estaban obsoletos. Por lo que se tuvo que realizar una extensa búsqueda entre el repositorio de kibana en github y sus issues para ver cómo era la nueva forma de registrar las visualización.

Al final, la mejor forma de averiguarlo fue inspeccionar dentro del código de kibana y ver

cómo registraban las visualizaciones que vienen de forma predeterminada en el cliente de kibana. Esta se encuentra en el fichero “kibana/src/legacy/core_plugins/kbn_vislib_vis_types/public/kbn_vislib_vis_types.js”, y la usaremos de base para la forma de registrar nuestras futuras visualizaciones.

Con esto, ya podemos importar el paquete que nos permitirá registrar la visualización:

```
import { setup as visualizations } from ' ../../src/legacy/
core_plugins/visualizations/public/np_ready/public/legacy ';
```

Ahora pasaremos a definir nuestra visualización:

```
// define the new visualization
const helloWorldDefinition = {
  name: 'hello_world_vis',
  title: 'Hello World',
  icon: 'visBarVertical',
  description: 'Hola mundo',
  visualization: hello_world_Controller,
  editor: 'default',
}
```

Parámetros utilizados:

- name: nombre de nuestra visualización.
- title: título que aparecerá en el menú para elegir nueva visualización.
- icon: ícono que aparecerá en dicho menú.
- description: descripción que aparecerá en dicho menú.
- visualization: indicaremos el controlador que usaremos en la visualización, éste lo creamos a continuación para exportarlo.
- editor: añade el editor, pero como por ahora no nos interesa, lo dejaremos en ‘default’.

Y por último, registramos la visualización:

```
// Register the provider with the visType Register
visualizations.types.createBaseVisualization
(helloWorldDefinition);
```

Ya tenemos registrada nuestra primera visualización. Lo que le sigue es crear el controlador donde estará toda la parte lógica de nuestro plugin, pero antes de comenzar con ella vamos a crear un pequeño template “public/index.html”, con el contenido de la visualización, que no es más que un “Hola Mundo”.

```
<div class='Hello world'>
    <p>!‘Hola Mundo!</p>
</div>
```

Listo esto, comenzamos a crear nuestro controlador “public/hello-world-controller.js”. Lo primero que haremos será importar nuestro template.

```
import template from './index.html'
```

Este controlador es una clase con su correspondiente constructor; pero también necesitará las funciones de “render()” y “destroy()” para funcionar. Comenzamos creando el constructor:

```
// define class
class VisController {
    constructor(el, vis) {
        console.log('Creating ...');
        this.el = el;
        this.vis = vis;

        this.container = document.createElement('div');
        this.container.className = 'myvis-container-div';
        this.el.appendChild(this.container);
    }
}
```

Lo único que hacemos es coger las variables el y vis y crear un container:

- el: contenido de la visualización.

- vis: contiene los parámetros que se crean al registrar la visualización.
- container: en este contenedor añadiremos todo lo que queremos que muestre la visualización.

A continuación crearemos la función “destroy()” que destruirá la visualización cada vez que salimos de la visualización o cada vez que actualizamos los datos que se van a mostrar en la visualización.

```
destroy () {
  this.el.innerHTML = '';
  console.log('Destroying ...');
}
```

Y por último, creamos la función “render()”.

```
render(visData, status) {
  if (visData) {
    this.container.innerHTML = template;
  }
};
```

Como Kibana es un cliente que permite monitorizar los datos almacenados en elasticsearch, al crear un visualización se espera que kibana realice una petición a elasticsearch con los datos que quiere mostrar; y éste los devuelva. Estos datos vienen en dentro de la variable “visData”; por tanto actualizaremos nuestra visualización cada vez que recibamos datos, aunque en esta ocasión los datos estarán en blanco. Lo único que haremos es meter el template que anteriormente hemos creado en el container de la visualización.

El siguiente paso es exportar el controlador:

```
export { VisController };
```

E importarlo en “public/hello-world-vis.js”:

```
import { VisController } from './hello-world-controller.js';
```

Como resultado nos encontramos con que dentro de crear nueva visualización aparece nuestra visualización con su nombre, icono y descripción.

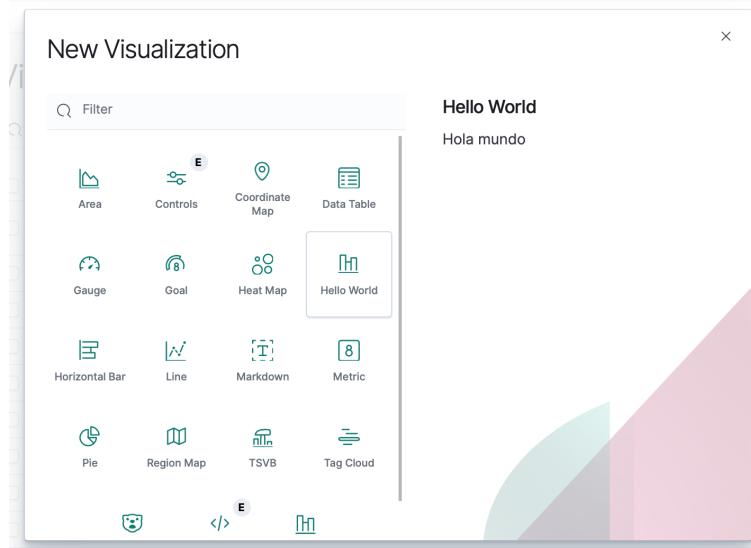


Figura 3.10: Menú con la nueva visualización

Y obteniendo como resultado nuestra visualización “Hola Mundo!”.

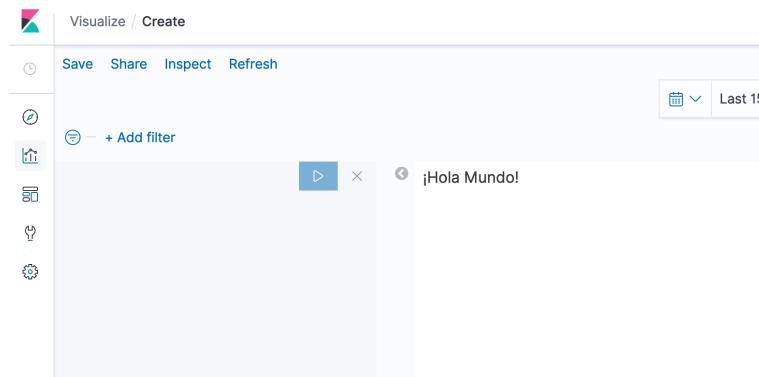


Figura 3.11: Resultado Visualización "Hola Mundo"

3.3.2. Implementación de A-Frame sin Datos

Una vez aprendidas las bases de cómo se crea un plugin de visualización para Kibana, el siguiente paso será conseguir integrar algún componente de la biblioteca de a-frame para comprobar si estas dos tecnologías son compatibles entre sí.

En esta situación crearemos una visualización, tomando como base el plugin “hello-world-vis” que creamos anteriormente; y le intentaremos añadir un componente primitivo de la biblioteca de a-frame (por ejemplo, un cubo).

Para esto, necesitaremos tener instalado previamente el módulo npm de a-frame. Esto se consigue añadiendo las dependencias en “package.json”.

```
"dependencies": {
  "aframe": "^0.9.2"
}
```

Y para instalarlo usaremos el comando “npm install” dentro del directorio que contiene nuestro “package.json”.

Una vez que hemos descargado e instalado el módulo de a-frame, lo declaramos en el controlador de la visualización “hello-world-controller.js” para que podamos usarlo en el template.

```
// import aframe
const aframe = require('aframe');
```

Y modificamos el HTML del template “index.html” para que dibuje un cubo con a-frame. Para ello, debemos conocer al menos estas dos etiquetas:

- a-scene: permite manejar elementos creados con THREE.js y WebVR. Todos estos elementos deben estar contenido en esta etiqueta, sino no funcionará. Salvo que se le declare, creará por defecto la cámara y las luces.
- a-box: crea un componente cubo en 3D según los atributos que le indiquemos.

```
<a-scene>
  <a-box position="-1 0.5 -3" rotation="0 45 0"
    color="#4CC3D9"></a-box>
</a-scene>
```

Obteniendo como resultado la siguiente visualización:

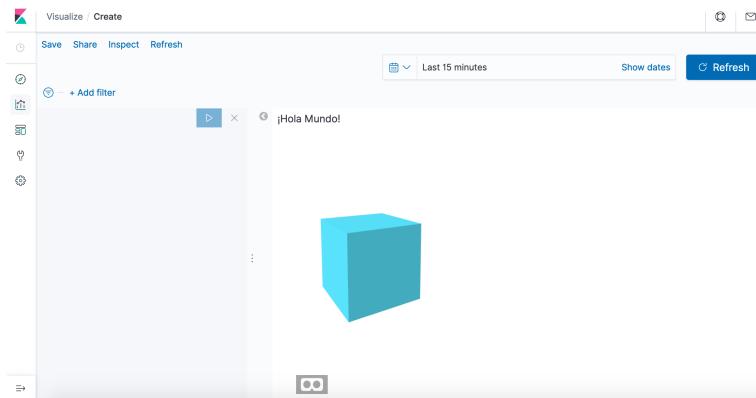


Figura 3.12: Resultado Visualización de cubo con A-frame son Datos

Con esto podemos asegurar que Kibana soporta visualizaciones creadas con la biblioteca a-frame.

3.3.3. Creación de Componente A-Frame

Ahora ya sabemos que kibana nos permite usar a-frame en sus visualizaciones, pero vamos a estudiarlo más en profundidad. Como se explicó anteriormente (ver cap), BabiaXR es una biblioteca de componente creadas con a-frame; por lo que en esta ocasión se creará un componente a-frame para entender mejor el funcionamiento interno de babiaXr.

Ya que hemos creado un cubo como componente primitivo, esta vez lo sustituiremos por otro cubo; pero creado por este método tomando como referencia el ejemplo que nos facilita la documentación de a-frame.

El primer paso es crear un fichero que se llamará “box.js” y registrar el componente de la siguiente esquema:

```
// Registro de la componente
AFRAME.registerComponent('box', {
  schema: {
    width: { type: 'number', default: 1 },
    height: { type: 'number', default: 1 },
    depth: { type: 'number', default: 1 },
    color: { type: 'color', default: '#AAA' }
  },
})
```

Los parámetros que le indicamos en el constructor son las propiedades que, más adelante, indicaremos dentro de las etiquetas del HTML.

Estas componentes funcionan como clases y deben contener ciertas funciones para que se puedan renderizar: init(), update(), destroy()

Comenzaremos creando la función “init()”

```
// Funcion inicial
init: function () {
    var data = this.data;
    var el = this.el;

    // Crea la figura
    this.geometry = new THREE.BoxBufferGeometry(data.width,
        data.height, data.depth);

    // Se le da un material
    this.material = new THREE.MeshStandardMaterial({
        color: data.color});

    // Creamos la mesh
    this.mesh = new THREE.Mesh(this.geometry, this.material);

    // Creamos el objeto 3D
    el.setObject3D('mesh', this.mesh);
},
...
});
```

Esta función inicializa la geometría, material y malla del componente a partir de los parámetros que recibe cuando se declaran en el HTML. Para luego renderizar la figura inicializada.

La función “update()” se ejecutará cuando algún dato cambia. En este caso no lo vamos a necesitar pero como más adelante vamos a extraer datos de elasticsearch para modificar el

componente, lo creamos.

```
// En caso de que los datos cambien, actualizamos la figura
update: function (oldData) {
    var data = this.data;
    var el = this.el;

    // If 'oldData' is empty, then this means we're in the
    // initialization process.
    // No need to update.
    if (Object.keys(oldData).length === 0) { return; }

    // Geometry-related properties changed. Update the geometry
    if (data.width !== oldData.width ||
        data.height !== oldData.height ||
        data.depth !== oldData.depth) {
        el.getObject3D('mesh').geometry = new
        THREE.BoxBufferGeometry(data.width, data.height,
        data.depth);
    }

    // Material-related properties changed. Update the material
    if (data.color !== oldData.color) {
        el.getObject3D('mesh').material.color = new THREE.Color(
        data.color);
    }
},
```

Esta función lo que hará será modificar nuestro cubo con los nuevos valores recibidos.

Y por último, creamos la función “destroy()” que borrará la figura al cerrar la visualización.

```
// Borramos la figura
remove: function () {
```

```
this.el removeObject3D('mesh');
}
```

Ahora que ya tenemos creada una componente propia, lo que sigue es declararla en nuestro controlador.

```
import './box.js'
```

Y modificar el template para que nos dibuje un cubo creado con nuestra componente; y le añadiremos también una esfera con la etiqueta `<a-sphere>` para ver que se pueden mezclar componentes primitivas de a-frame con componente creadas por otros usuarios.

Para insertar estas componentes, que no vienen de forma pre-determinadas por a-frame, usaremos la etiqueta `<a-entity>` seguido del atributo con el nombre de la componente tal y como se ha registrado. En nuestro caso `<a-entity box></a-entity>`.

```
<a-scene embedded>
  <!-- Figura cubo por componente aframe -->
  <a-entity box="width: 0.5; height: 0.25; depth: 1;
    color: #4CC3D9" position="-1 0.5 -3"
    rotation="0 45 0"></a-entity>
  <!-- Figura esfera basica de aframe -->
  <a-sphere position="0 1.25 -5" radius="1.25"
    color="#EF2D5E" shadow></a-sphere>
</a-scene>
```

Una de las complicaciones que no se mencionaron anteriormente fue que la escena se salía de la visualización de kibana. Añadiendo el atributo `embedded` a la escena corregimos este error; pero para asegurarnos haremos una pequeña hoja de estilos para que el `<div>` de nuestra visualización no se salga, la cual llamaremos “hello-world.less”.

```
.myvis-container-div {
  width: 100%;
}
```

Y la importamos en el controlador de la visualización.

```
import style from './aframe.less';
```

Obteniendo como resultado la siguiente visualización:

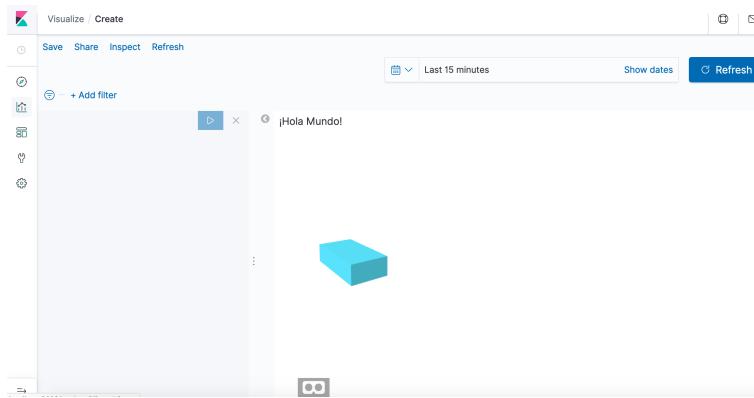


Figura 3.13: Resultado Visualización de componente cubo con Esfera de A-frame sin Datos

3.4. Sprint 2: Visualización con datos Elasticsearch

Hasta este punto hemos aprendido a crear un plugin simple para kibana e integrarlo con aframe. El siguiente paso será aprender a sacar datos de elasticsearch y poder dibujar un figura de aframe, en este caso un cubo, con las dimensiones que le obtenemos de dichos datos. Para ello, seguiremos los siguientes pasos:

1. Añadir Editor y Schemas.
2. Construir Visualización que muestre datos.
3. Integración de Aframe con datos.

3.4.1. Editor y Schemas

Antes de ponernos a crear una nueva visualización donde nos muestre datos de elasticsearch, necesitamos un editor que nos permita seleccionar los datos que queremos mostrar.

Comenzaremos creando el diseño de nuestro editor. Kibana nos permite seleccionar dos tipos de datos: metrics y buckets.

- metrics: hace referencia a datos que se pueden calcular, por ejemplo, media, máximo, sumatorio, etc.
- buckets: refiere a un conjunto de datos que no se pueden calcular, por ejemplo, fechas, sucursales, tipos, etc.

Como el objetivo final de este sprint es crear una visualización que nos muestre un cubo, determinamos que necesitaremos 3 datos metrics que serán los datos que usaremos para crear las dimensiones del cubo que dibujaremos más adelante.

Estos datos vienen estructurados dentro de Schemas que el propio Kibana permite declarar a la hora de crear la visualización para luego registrarla. Para ello, añadiremos las siguientes líneas en “kbn_aframe.js”:

Antes de nada, importamos los correspondientes paquetes que nos permite declarar los schemas e indicar el tipo de dato que vamos a usar, en este caso metrics.

```
import { Schemas } from 'ui/vis/editors/default/schemas';
import { AggGroupNames } from 'ui/vis/editors/default';
```

Una vez importado, añadiremos nuevos parámetros dentro de la función `createBaseVisualization()`:

```
schemas: new Schemas([
  {
    group: AggGroupNames.Metrics,
    name: 'x-axis',
    title: 'X-axis',
    min: 1,
    max: 1,
    aggFilter: [ 'count', 'avg', 'sum', 'min', 'max',
      'cardinality', 'std_dev' ]
  },
  {
    group: AggGroupNames.Metrics,
    name: 'y-axis',
    title: 'Y-axis',
  }
])
```

```

        min: 1,
        max: 1,
        aggFilter: [ 'count', 'avg', 'sum', 'min', 'max',
          'cardinality', 'std_dev' ]
      },
      {
        group: AggGroupNames.Metrics,
        name: 'z-axis',
        title: 'Z-axis',
        min: 1,
        max: 1,
        aggFilter: [ 'count', 'avg', 'sum', 'min', 'max',
          'cardinality', 'std_dev' ]
      }
    ],
  )
)
]
),

```

Parámetros utilizados:

- group: indica el tipo de dato: metric o bucket.
- name: nombre que recibirá el dato.
- title: el título que mostrará en el editor.
- min: indica el mínimo de datos que tenemos usar.
- max: indica el máximo de datos que podemos usar.
- aggFilter: aquí podemos indicarle qué tipos de dato, en este caso tipos de métricas, queremos usar.

Al instante de añadir esto, obtenemos como resultado un editor como el que vemos a continuación:

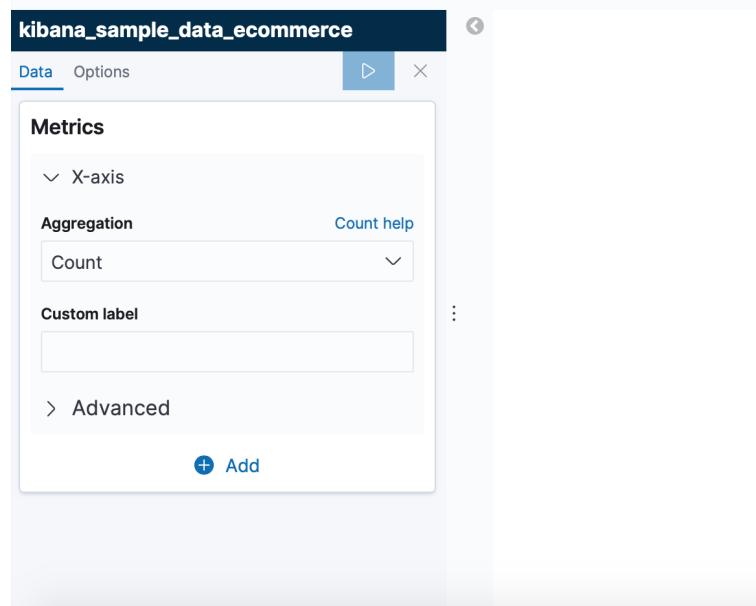


Figura 3.14: Resultado del Editor

3.4.2. Plugin Simple con datos

Ahora que ya podemos elegir los datos en el editor de Kibana, el siguiente paso es conseguir que la visualización muestre dichos datos de forma sencilla.

Para esto debemos haber aprendido, como se vió en el sprint anterior, el funcionamiento de creación y renderizado para entender dónde Kibana recibe los datos de Elasticsearch. Así que lo primero que tenemos que hacer es encontrar dónde recibimos los datos que pedimos con el editor.

Si revisamos el esquema del proceso de creación de una visualización (ver capítulo), podemos suponer que los datos que recibimos vendrán en la variable visData. Lo comprobaremos de la siguiente forma:

```
render(visData, status) {
  if (visData) {
    console.log(visData);
  }
}
```

Con esto podremos ver que datos recibimos en la consola del navegador.

```
[{"type": "kibana_datatable", "rows": [{"col-0-1: 1, col-1-2: 20.984375, col-2-3: 1}], "columns": [{"id: "col-0-1", name: "Count"}, {"id: "col-1-2", name: "Max products.price"}, {"id: "col-2-3", name: "Average day_of_week_i"}]}]
```

Figura 3.15: Captura de visData en la consola

Esto nos sirve para analizar en cómo es el formato de los datos recibidos de elasticsearch, para luego extraer los datos que nos interesa mostrar.

En primer lugar, vemos que se divide en 3 registros:

- type: tipo del archivo. En este caso, kibana_datatable.
- rows: contiene únicamente los datos obtenidos en formato registro en la que la clave con la forma col-X-X que indica la posición; y el valor con el dato obtenido.
- columns: contiene las etiquetas que hacen referencia a los datos obtenidos. Vienen en forma de array de registros, donde el id indica la posición del dato; y el name indica el nombre del valor.

Analizando todo esto sacaremos los datos, con sus nombres, para guardarlos en la variable metrics.

```
const table = visData;
const metrics = [];

// get metrics
table.columns.forEach((column, i) => {
  var value;
  const name = column.name;
  const id = column.id;
  table.rows.forEach(row => {
    if (row[id]){
      value = Math.round(row[id]*100)/100;
    }
  });
  metrics.push({
    id,
    name,
    value
  });
});
```

```

        name: name ,
        value: value
    });
});

```

Tras esto, lo añadimos a la visualización, de forma simple, de la siguiente forma:

```

// render metric in vis
metrics.forEach((metric, i) => {
    const metricDiv = document.createElement('div');
    metricDiv.className = 'myvis-metric-div';
    metricDiv.innerHTML = '<b>${
        metric.name}</b>' +
        ${
            metric.value};
    metricDiv.setAttribute('style', `font-size: ${this.vis.params.fontSize}pt`);
    this.container.appendChild(metricDiv);
});

```

Como resultado obtenemos las figuras, que agregamos anteriormente en el capítulo anterior, seguido de los 3 valores que pedimos desde el editor.

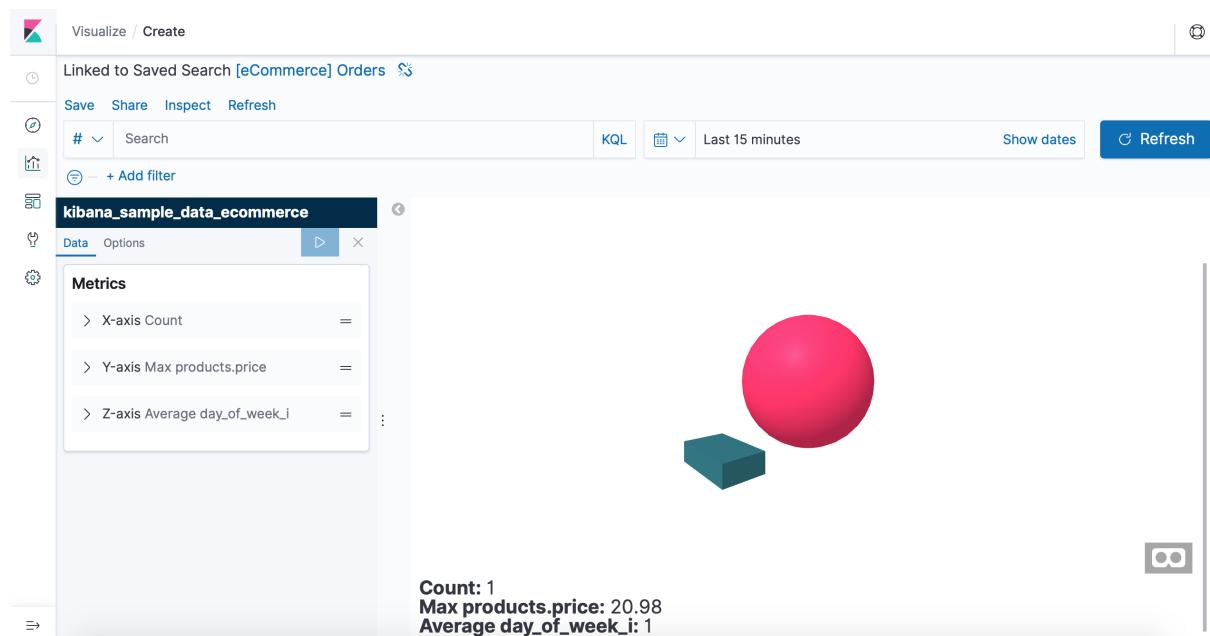


Figura 3.16: Resultado Visualización con datos

3.4.3. Integración de A-frame con datos

Ahora que ya hemos aprendido a obtener los datos que pedimos a elasticsearch para incluirlos en nuestra visualización, el siguiente paso es saber añadir estos datos para poder crear las figuras en base a estos datos.

Únicamente nos centraremos en crear el cubo tomando como valores las 3 métricas que pedimos desde el editor. Para ello, tenemos dos opciones:

1. Insertar los datos al html usando angularJS.
2. Crear la figura directamente desde el controller usando a-frame con javascript.

Analizando un poco como kibana crea sus visualizaciones predeterminadas, creo que la mejor opción es la segunda y así nos ahorramos tener que usar más bibliotecas.

Por eso, eliminaremos el archivo “index.html”, eliminamos la línea donde importamos dicho archivo y cambiamos la siguiente línea de render():

```
this.container.innerHTML = '';
```

Ahora crearemos el escenario justo de donde añadimos las métricas obtenidas de elasticsearch.

```
// Creando escenario
var escena = document.createElement('a-scene');
escena.setAttribute('embedded', true);
```

El siguiente paso será añadir las figuras. Empezaremos por dibujar una esfera usando las primitivas que tiene a-frame. Usaremos los mismos valores que teníamos en “index.html” pero variando algún dato al gusto. Por el momento solo le añadimos los datos de forma manual porque lo único que queremos es probar cómo podemos crear una figura de esta forma.

Lo creamos de la siguiente forma:

```
// entity primitive
var entidad = document.createElement('a-entity');
entidad.setAttribute('geometry', {
  primitive: 'sphere',
  radius: 1.25
});
```

```

});  

entidad.setAttribute('position', {  

    x: -2.5,  

    y: 1,  

    z: -5  

});  

entidad.setAttribute('material', {  

    color: '#EF2D5E'  

});  

entidad.setAttribute('shadow', true);  
  

escena.appendChild(entidad);

```

Esta última línea, añade la figura en el escenario anteriormente creado.

Ahora que está creado todo la escena con la figura, la añadimos el contenedor de la visualización para que nos la muestre en Kibana.

```
this.container.appendChild(escena);
```

Obteniendo como resultado la siguiente visualización.

Una vez hemos conseguido que nos dibuje la esfera, haremos el mismo procedimiento para crear el cubo pero con un par de variaciones:

1. Usaremos el cubo que creamos por componente.
2. Añadiremos los datos obtenidos de elasticsearch.

Como usaremos el componente, revisaremos que “box.js” este importado en el controlador.

Ahora, declaramos un array con las dimensiones que vamos a usar para crear el cubo.

```
const axis = [];
```

Y añadimos los valores en la variable axis tras añadirlo también en metrics.

```
// add axis  
axis.push(value);
```

Por último, solo nos queda crear la figura de la misma forma y añadirlo en la escena:

```
// entity con box.js
var caja = document.createElement('a-entity');
caja.setAttribute('box', {
  height: axis[0]/10, //0.5
  width: axis[1]/10, //0.25
  depth: axis[2]/10, //1
});
caja.setAttribute('position', {
  x: 0,
  y: 1,
  z: -5
});
caja.setAttribute('rotation', {
  x: 0,
  y: -45,
  z: 0
});
caja.setAttribute('material', 'color', '#4CC3D9');

escena.appendChild(caja);
```

Como se ve, la diferencia con la esfera es que creamos un a-entity con el atributo 'box' y no 'geometry' y dándole los datos que declaramos en el constructor de box.js.

Con esto, obtenemos como resultado un cubo que varía en base a los datos que le pedimos a elasticsearch.

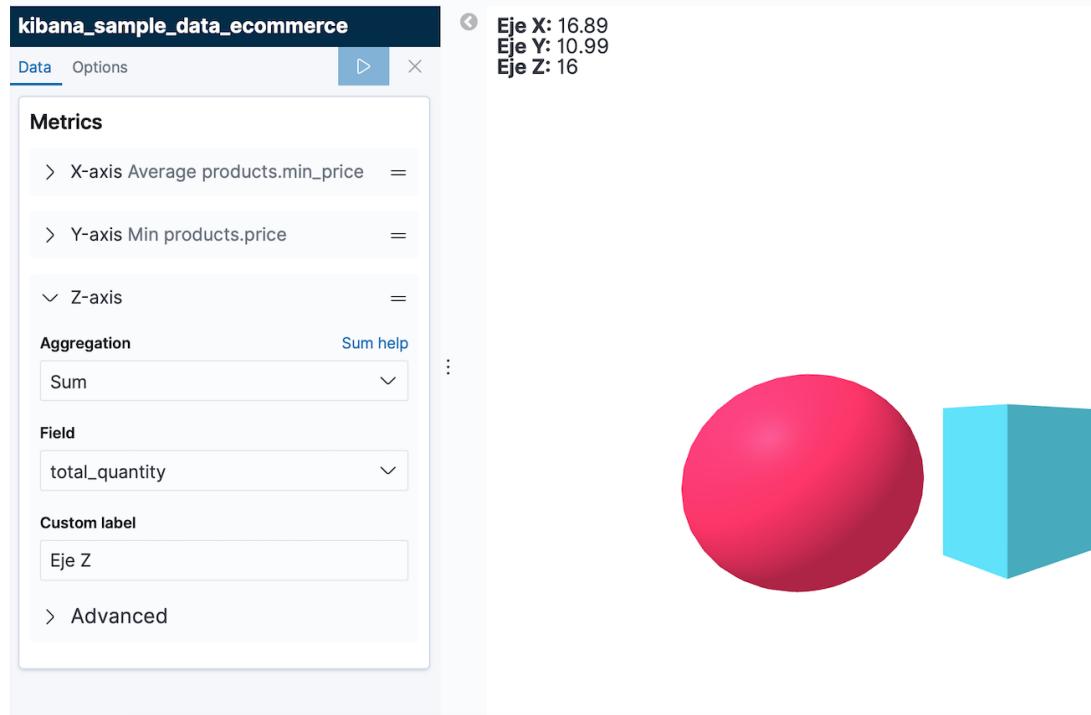


Figura 3.17: Resultado de integración de A-Frame con datos

3.5. Sprint 3: Implementación de BabiaXR

Tras haber completado toda la parte básica en cuanto a la creación de un plugin para kibana y la integración de gráficos VR usando A-frame, procederemos a integrar la biblioteca de BabiaXR.

Como explicamos anteriormente (ver capítulo) BabiaXR es una biblioteca de componentes donde cada componente es un tipo de gráfica y por tanto será un tipo de visualización de nuestro plugin.

Para este sprint, cogeremos un solo componente de BabiaXR y lo integraremos en nuestro plugin. El componente elegido será “geopiechart” que nos creará un visualización que nos mostrará una gráfica de tipo pie.

Al igual que hicimos en el anterior sprint, dividiremos el sprint en dos partes:

1. Integraremos pie chart sin usar datos.

2. Integraremos pie chart usando datos.

3.5.1. Visualización Pie Chart sin Datos

Anteriormente estábamos trabajando con un plugin de prueba; pero como esta parte ya será parte del resultado final, crearemos un nuevo plugin, cogiendo como referencia nuestro plugin de prueba, y lo llamaremos “kbn_aframe”.

Para empezar, introducimos el nombre de nuestro nuevo plugin, además de agregarle el nombre del autor y las dependencias que necesitaremos. Como lo hemos hecho anteriormente, este va dentro de “package.json”.

```
{
  "name": "kbn_aframe",
  "version": "0.0.1",
  "kibana": {
    "version": "7.6.0"
  },
  "authors": [
    "Andrea Villaverde <camaratomoyo@gmail.com>"
  ],
  "dependencies": {
    "aframe": ">=1.0.0",
    "aframe-babia-components": ">=1.0.5",
    "aframe-extras": ">=6.1.0",
    "aframe-environment-component": ">=2.0.0",
  }
}
```

Dependencias que necesitamos:

- aframe: para que podamos crear componentes hechas con a-frame.
- aframe-babia-components: biblioteca de componentes de a-frame con las gráficas que vamos a implementar.
- aframe-extras: componente de aframe que añade otros tipos de controles diferentes a los que vienen de base con a-frame.

- aframe-environment-component: componente de a-frame que permite crear entornos visualmente más atractivos que el que viene de base en a-frame.

También añadiremos el directorio de nuestro nuevo fichero principal, al que llamaremos kbn_aframe.js, dentro de “index.js”.

```
export default function (kibana) {
  return new kibana.Plugin({
    uiExports: {},
    visTypes: [
      'plugins/kbn_aframe/kbn_aframe'
    ]
  });
}
```

Y declararemos nuestra nueva visualización para registrarla posteriormente.

```
name: 'pie_chart',
title: 'VR Pie',
icon: 'visPie',
description: 'Compare parts of a whole (VR Chart).',
visualization: VisController,
editor: 'default'
```

Ahora nos vamos al controlador, que hemos llamado “kbn_aframe_controller.js”, y lo primero que haremos será importar las dependencias que hemos instalado anteriormente.

```
import 'aframe';
import 'aframe-babia-components';
import 'aframe-environment-component';
import 'aframe-extras';
```

El siguiente paso es crear nuestra visualización dentro del render tomando como referencia el ejemplo que encontramos en la documentación de BabiaXR. Para ello, lo dividiremos en varias partes:

1. Crearemos el entorno y le añadiremos las luces de la escena.
2. Añadiremos los datos de forma manual, ya que en esta ocasión estamos trabajando sin datos.
3. Añadiremos el componente “geopiechart” a la visualización.
4. Añadiremos los controles y la cámara a la escena.

Comenzamos creando el escenario a-frame, añadiendo un tipo de entornos con aframe-environment-component; y le añadiremos las luces. Para no complicarnos mucho, le pondremos luces de tipo ambiente que iluminará la escena por completo.

```
// Create a_scene
var escena = document.createElement('a-scene');
escena.setAttribute('id', 'AframeScene');
escena.setAttribute('embedded', true);

// Environment
var environment = document.createElement('a-entity');
environment.setAttribute('environment', {
  preset: 'contact',
  skyType: 'gradient'
});
escena.appendChild(environment);

// LIGHT
var light = document.createElement('a-light');
light.setAttribute('type', 'ambient');
light.setAttribute('intensity', 1);
light.setAttribute('position', { x:10, y:20, z:30 });
escena.appendChild(light);
```

Atributos utilizados dentro del componente environment:

- preset: indicas el estilo del entorno que quieras usar. Aframe-environment-component muestra un catálogo de entornos para elegir al gusto.
- skyType: indica el tipo de skybox que deseas poner. Aunque puedes elegir el entorno predeterminado, también se le puede personalizar sus elementos. Este es un ejemplo de ello, el cual te permite indicar si quieres que sea un solo color o con degradado, entre otros.

Ahora procedemos a añadir los datos que vamos a necesitar para construir el componente “geopiechart”. Estos datos irán en formato json y deben seguir la estructura que viene especificada en la documentación de babiaxr.

```
[{"key ":" kbn_network ", "size ":10} ,  
 {"key ":" Maria ", "size ":5} ,  
 ...  
 ]
```

Según esto construiremos los datos de la siguiente manera:

```
let data_json = [{"key ":"URJC ", "size ":5} , {"key ":"Andrea ",  
 "size ":18} , {"key ":"David ", "size ":6} , {"key ":"Jesus ", "size ":2} ,  
 {"key ":"Aframe ", "size ":22}]
```

Y procedemos a crear tal y como nos lo indica la documentación de babiaxr.

```
var chart = document.createElement('a-entity');  
chart.setAttribute('geopiechart', {  
    legend: true, // legend  
    axis: true, // axis  
    data: data_json // data  
});  
chart.setAttribute('position', { x: 0, y: 2, z: -4 });  
chart.setAttribute('rotation', { x: 90, y: -20, z: 0 });  
escena.appendChild(chart);
```

Los atributos utilizados:

- legend: si deseas que al pasar por encima del elemento nos muestre su valor.
- axis: si deseamos que se muestre los ejes de la gráfica, aunque en este componente no se mostrarás porque no usa ejes.
- data: introduce los datos que quieras mostrar en la gráfica.

Por último, añadiremos los controles (usando aframe-extras), para que también puedan usarse en un futuro con las gafas tipo oculus, entre otras; y la cámara indicando la posición en la que queremos empezar dentro de la escena.

```
var controls = document.createElement('a-entity');
controls.setAttribute('movement-controls', {fly: true});

var camera = document.createElement('a-camera');
camera.setAttribute('position', {x: 0, y: 2.5, z: 1});
controls.appendChild(camera);

var cursor = document.createElement('a-entity');
cursor.setAttribute('cursor', {rayOrigin: "mouse"});
controls.appendChild(cursor);

var laser = document.createElement('a-entity');
laser.setAttribute('laser-controls', {hand: "right"});
controls.appendChild(laser);

escena.appendChild(controls);
```

Una vez hemos añadido todo esto, pasamos a ver el resultado en kibana.

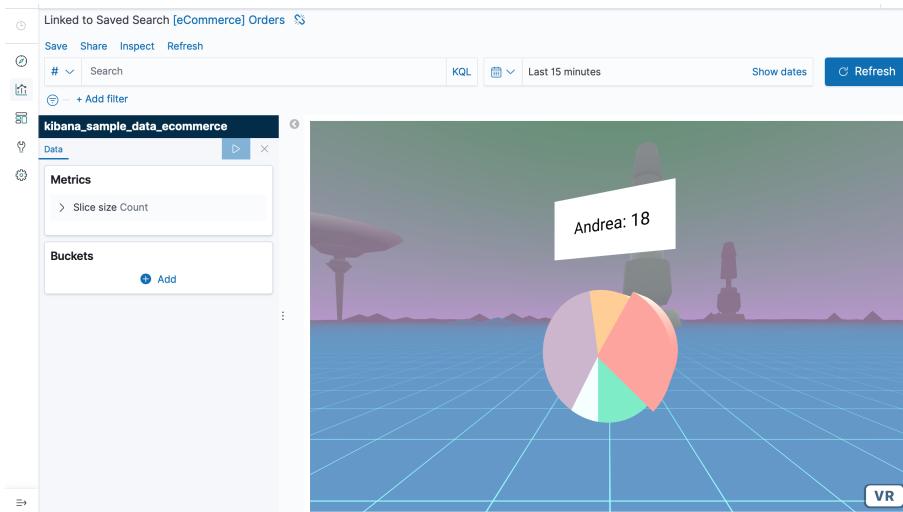


Figura 3.18: Resultado implementacion de geopiechart sin datos

3.5.2. Visualización Pie Chart con datos

Ahora que ya sabemos que se puede implementar el componente de babiaxr; lo último que nos queda para terminar este sprint es hacer que se cree el componente “geopiechart” usando los datos que nos devuelve kibana de elasticsearch.

Lo primero será ver qué tipo de datos necesitaremos y lo indicaremos en el editor para que kibana pueda hacer su correspondiente.

Como trabajamos con un formato de key-size, utilizaremos un bucket que nos dará las keys y una metric que nos devolverá un valor por cada key. Una vez establecido esto, procedemos a configurar el editor, de la misma forma que lo hicimos en el sprint 2.

```
editorConfig : {
  schemas: new Schemas([
    {
      group: AggGroupNames.Metrics ,
      name: 'metric' ,
      title: 'Slice size' ,
      min: 1 ,
      max: 1 ,
      aggFilter: ['sum', 'count', 'cardinality', 'top_hits'] ,
      defaults: [{ schema: 'metric' , type: 'count' }],
```

```

} ,
{
  group: AggGroupNames.Buckets ,
  name: 'segment' ,
  title: 'Split slices' ,
  min: 0 ,
  max: 1 ,
  aggFilter: [ '!geohash_grid' , '!geotile_grid' , '!filter' ] ,
},
]) ,
}

```

Tal y como vimos también anteriormente cuando creamos el cubo a partir de datos; tomaremos esa referencia para crear el data.json según el formato que nos indica babiaxr.

Esta vez es la primera vez que trabajamos con un bucket, por lo que la forma de sacar las métricas será diferente, ya que ahora debemos trabajar con las keys del bucket y no solo con metrics. Es decir, en esta ocasión, meteremos en metrics los valores obtenidos en forma de diccionario.

```

if (( ! old_data ) || ( old_data != visData )){

  this . container . innerHTML = '' ;
  old_data = visData ;
  const table = visData ;
  const metrics = [] ;

  var id = [] ;
  // get metrics
  table . columns . forEach (( column , i ) => {
    var name = column . name ;
    id . push ( column . id );
  });
}

```

```
table.rows.forEach(row => {
  let values = [];
  id.forEach(id => {
    if (row[id]){
      values.push(row[id]);
    };
  });
  // push metrics
  metrics.push({
    "key": values[0],
    "size": values[1]
  });
});
})
```

Ahora nos encontramos con la situación de que el componente “geopiechart” necesita que le pasemos los datos en formato json, por lo que le damos ese formato usando la función “JSON.stringify()”.

```
let data_json = JSON.stringify(metrics);
```

Con esto, hemos conseguido que a partir de ahora los datos que introducimos desde el editor sean mostrados en una gráfica tipo pie.

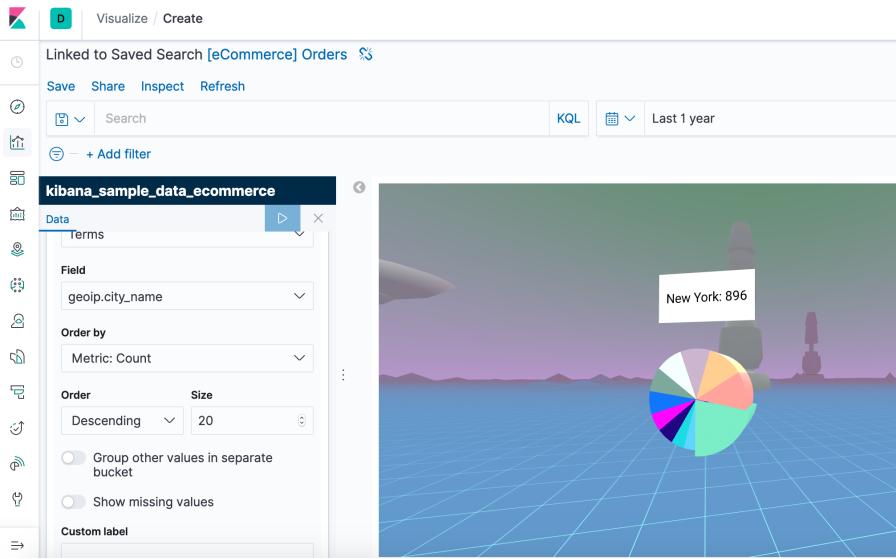


Figura 3.19: Resultado de geopichart usando datos de Kibana

Para obtener esta gráfica, le hemos pedido a kibana que nos muestre el número de ventas segmentadas por ciudad; y que solo nos muestre las 20 con más ventas.

3.6. Sprint 4: Nuevas Visualizaciones

En este último sprint, incluiremos el resto de componentes que existen actualmente dentro del módulo de babiaxr. También aprovecharemos para optimizar el código y prepararlo en caso de querer añadir más características o más visualizaciones, como ocurrirá en este sprint.

Como cada componente utiliza diferentes tipos de formato a la hora de introducir los datos, iremos desde el más sencillo hasta el más complejo. Dicho esto, establecemos que las tareas de este sprint serán:

1. Crear visualización con geosimplebarchart.
2. Crear visualización con geo3dbarchart.
3. Crear visualización con geobubblechart.

3.6.1. Visualización Simple Bar

A primera vista parece que es muy sencillo de hacer, pero hay que tener un par de cosas en cuenta que antes no teníamos; pues esta es la primera vez que tenemos que registrar dos

visualizaciones en un mismo plugin. Hasta ahora siempre habíamos trabajado con una sola visualización por plugin.

En realidad esta parte es muy sencilla pues simplemente hay que crear otra constante con los parámetros de la visualización y registrarla.

Analizando un poco cómo lo hacen los desarrolladores de kibana, vemos que crean las definiciones de las visualizaciones en archivos individuales que luego importan en el fichero principal y únicamente hay que registrar las visualizaciones en él.

Por eso, comenzaremos trasladando el código con la definición de la visualización de pie-chart a otro fichero que llamaremos “pie.js”.

Otro de los detalles a tener en cuenta es cómo el controlador va a saber que se trata de una gráficas u otras, ya que usaremos el mismo controlador para todas las visualizaciones que vamos a introducir. Para eso, existe un parámetros dentro de la definición de la visualización que permite pasar datos que luego recogeremos en “this.vis.params” dentro del controlador.

Este parámetros es “visConfig” y nos permite introducir datos por defecto desde la definición de la visualización o a través del editor en caso de crear una pestaña con opciones.

Para nuestro caso, usaremos la opción por defecto y en ella le indicaremos que añada el tipo de gráfica de cada una de las visualizaciones. Esto hará que tengamos una variable que nos ayuda a identificar qué gráfica es y cómo proceder a la hora de coger las métricas y cómo construir la gráfica.

Así que, añadiremos esta líneas dentro de la definición de pie chart que ya se encuentra en “pie.js”.

```
visConfig : {  
    defaults : {  
        type : 'pie',  
    },  
},
```

Al mismo tiempo, vamos a necesitar exportar la definición de la visualización de pie chart para poder importarla más adelante.

```
export const pieDefinition = {
```

```
    ...
```

```
// definición de pie chart
};
```

Una vez que ya tenemos trasladada la definición de pie chart, crearemos otra definición para la nueva visualización. Anteriormente se mencionó que iríamos desde la más sencilla a la más compleja. Para esto analizamos el formato de cada uno de los componente que hay en kibana y los comparamos con el que ya tenemos implementado. A primera vista el componente “geo-simplebarchart” es el más parecido al “geopiechart” pues su formato JSON son exactamente iguales.

Por tanto, procedemos a definir la nueva visualización en “simple_bar.js”:

```
export const simpleBarDefinition = {

  name: 'bar_chart',
  title: 'VR Vertical Bar',
  icon: 'visBarVertical',
  description: 'Assign a continuous variable to each axis  
(VR Chart).',
  visualization: VisController,

  visConfig: {
    defaults: {
      type: 'simple_bar',
    },
  },

  editorConfig: {
    schemas: new Schemas([
      {
        group: AggGroupNames.Metrics,
        name: 'metric',
        title: 'Y-axis',
      },
    ]),
  },
};
```

```

        min: 1,
        max: 1,
        aggFilter: [ 'sum', 'count', 'cardinality',
                      'top_hits'],
        defaults: [{ schema: 'metric', type: 'count' }],
      },
      {
        group: AggGroupNames.Buckets,
        name: 'segment',
        title: 'Split Chart',
        min: 0,
        max: 1,
        aggFilter: [ '!geohash_grid', '!geotile_grid',
                      '!filter'],
      },
    ],
  }
};


```

Como se ve, hemos añadido también el “type” dentro de “visConfig” para indicarle que esta visualización es en la que queremos crear una gráfica usando el componente “geosimplebar-chart”.

Lo que sigue es importar las definiciones, la que ya teníamos y la nueva, dentro de “kbn_aframe.js”.

```

import { pieDefinition } from './pie';
import { simpleBarDefinition } from './simple_bar';

```

Y registramos la nueva visualización:

```

visualizations.types.createBaseVisualization
  (simpleBarDefinition);

```

Desde kibana ya se podrá ver que ya se puede crear un nuevo tipo de gráfica; pero antes haremos un par de modificaciones en la lógica de nuestro controlador; pues si lo dejamos tal

cual dibujará también una gráfica tipo pie.

Para arreglar esto hemos creado el parámetro type en la definición de cada visualización. Esta se guarda dentro de la variable “this.vis.params.type” y con ella trabajamos para que cree la gráfica de forma correcta para cada caso.

```
let type = this.vis.params.type
if ( type == 'pie' ) {
    chart.setAttribute('geopiechart', {
        legend: true, // legend
        data: data // data
    });
    chart.setAttribute('position', { x: 0, y: 2, z: -4 });
    chart.setAttribute('rotation', { x: 90, y: -20, z: 0 });
} else if (type == 'simple_bar') {
    chart.setAttribute('geosimplebarchart', {
        legend: true, // legend
        axis: true, // axis
        data: data // data
    });
    chart.setAttribute('position', { x: -2, y: 0, z: -5 });
    chart.setAttribute('rotation', { x: 0, y: 0, z: 0 });
}
```

Hasta esta parte, la nueva visualización funciona sin problemas; pero decidimos anteriormente que íbamos a optimizar un poco el código para que fuera un poco más fácil de comprender en caso de que alguien quiere contribuir más adelante.

Como primera parte de esta tarea pasaremos las partes que siempre será igual en todos los escenarios a otras funciones que irán en módulos a parte y que los incluiremos en el render con sus llamadas correspondientes. Estas partes serán la de creación del entorno y la de insertar la cámara y controles. Lo haremos de esta manera porque una parte tiene más que ver con el aspecto estético de la escena y el otro más con la parte del manejo y controles sobre la escena.

Empezamos trasladando el código en el que creamos la escena, el entorno y las luces a un

función “CreateScene()” que exportamos. Este fichero los nombramos “a_scene.js”.

```
function createScene (){
    // Create a scene
    var escena = document.createElement('a-scene');
    ...
    // addimos environment y lights
    return escena;
}

export { createScene };
```

Hacemos lo mismo con la líneas correspondientes a la creación de los controles y la cámara; y lo metemos en la función “createControls()” dentro del nuevo fichero “a_controls.js”.

```
function createControls (){
    ...
    // addimos controls y camera
    return controls;
}

export { createControls };
```

Hecho esto, importamos estas funciones en “kbn_controller.js” y hacemos las llamadas correspondiente donde antes teníamos esa parte de código.

```
import { createScene } from './a_scene';
import { createControls } from './a_controls';
...

let escena = createScene();
let controls = createControls();

escena.appendChild(controls);
escena.appendChild(chart);
this.container.appendChild(escena);
```

Finalmente podemos irnos a Kibana para ver los resultados.

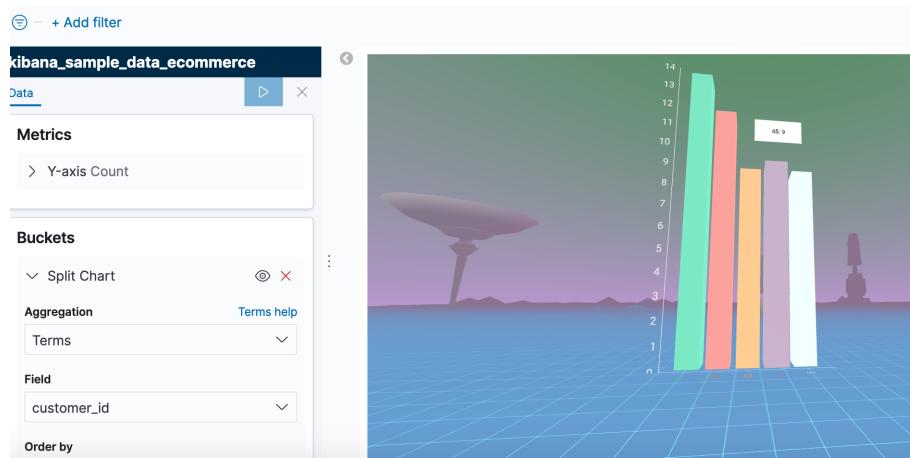


Figura 3.20: Resultado visualización Simple Bar Chart

Para este ejemplo le hemos pedido que nos muestre lo usuarios que han hecho más compras durante un periodo de tiempo, se les etiqueta por id de usuario; y que nos muestre con la altura de la barra la cantidad de dichas compras.

3.6.2. Visualización 3D Bars

Seguimos incluyendo nuevas visualizaciones. En esta ocasión agregaremos una visualización que creará un gráfica de barras en 3D usando el componente “geo3dbarchart”. A partir de aquí, los componentes que añadamos no tendrán el mismo formato JSON como tenían “geopiechart” y “geosimplebarchart”.

```
[{"key": "David", "key2": "2019", "size": 9},
 {"key": "David", "key2": "2018", "size": 8},
 ...
 ]
```

Analizando el formato JSON de “geo3dbarchart”, vemos que en esta ocasión existe una nueva variable que es “key2”. Esta indica que, al igual que “key”, corresponde a una forma de etiquetar cada valor mostrado, en este caso serán dos etiquetas por valor. Esto, en términos visuales, lo que hará es etiquetar las barras en los ejes X y Z.

Una vez hemos comprendido esto, ya podemos determinar como será nuestro editor. Trabajaremos con 2 claves y un valor, lo que en término del editor de kibana se traduce en 2 buckets y un metric.

Determinado esto, procedemos a crear la definición de la nueva visualización en “3dbar.js”.

```
export const geo3dBarDefinition = {  
  
    name: '3d_bar_chart',  
    title: 'VR Vertical Bar 3D',  
    icon: 'visBarVertical',  
    description: 'Assign a continuous variable to each axis  
        (VR Chart).',  
    visualization: VisController,  
  
    visConfig: {  
        defaults: {  
            type: '3d_bar',  
        },  
    },  
  
    editorConfig: {  
        schemas: new Schemas([  
            {  
                group: AggGroupNames.Metrics,  
                name: 'metric',  
                title: 'Y-axis',  
                min: 1,  
                max: 1,  
                aggFilter: ['sum', 'count', 'cardinality',  
                    'top_hits'],  
                defaults: [{ schema: 'metric', type: 'count' }],  
            },  
        ]),  
    },  
};
```

```
{
    group: AggGroupNames.Buckets ,
    name: 'segment' ,
    title: 'Split Chart' ,
    min: 2 ,
    max: 2 ,
    aggFilter: [ '!geohash_grid' , '!geotile_grid' ,
        '!filter' ] ,
},
],
}
};
```

Como se ve en la configuración del editor, le estamos indicando que debe tener 2 buckets porque sino la gráfica no se dibujará.

Y procedemos a registrarla dentro en “kbn_aframe”.

```
import { geo3dBarDefinition } from './3d_bar';
...
visualizations.types.createBaseVisualization
(geo3dBarDefinition);
```

Ahora que hemos registrado la visualización, pasamos a la parte del controlador. Empezando por la parte de la obtención de los datos que luego pasamos a JSON para luego crear su correspondiente gráfica. A cada caso que metemos, el código se vuelve más grande, por lo que se tomó la decisión de sacar de “render()” y crear una función “getMetrics()” para solo centrarnos en esa parte.

Esta función se encontrará en un fichero “metrics.js”, donde le introduciremos los datos y el tipo de visualización para que nos devuelva sus correspondientes datos, perfectamente estructurados según el formato de cada tipo, para posteriormente convertirlos en JSON.

```
function getMetrics (table , type){
    var metrics = [];
    var id = [];
```

```
if (table.columns.length == 1){  
    // only one  
    var name = table.columns[0].name;  
    var size = table.rows[0][table.columns[0].id];  
    metrics.push({  
        "key": name,  
        "size": size  
    });  
} else {  
    table.columns.forEach((column, i) => {  
        var name = column.name;  
        id.push(column.id);  
    });  
  
    table.rows.forEach(row => {  
        let values = [];  
        id.forEach(id => {  
            if (row[id]){  
                values.push(row[id]);  
            };  
        });  
        // push metrics  
        if ( type == '3d_bar' ){  
            metrics.push({  
                "key": values[0].toString(),  
                "key2": values[1].toString(),  
                "size": values[2],  
            });  
        } else if ((type == 'simple_bar')||(type == 'pie')) {  
            metrics.push({  
                "key": name,  
                "size": size  
            });  
        }  
    });  
}
```

```

        "key": values[0].toString(),
        "size": values[1],
    });
}
};

return metrics;
}

export { getMetrics };

```

Una vez hemos trasladado el código con sus correspondientes modificaciones para el caso de la nueva visualización, procedemos a importar la función y llamarla donde antes teníamos las líneas para obtener las métricas.

```

import { getMetrics } from './metrics';
...

let metrics = getMetrics(table, this.vis.params.type);

```

Por último nos faltaría añadir el caso del nuevo componente en la parte donde creamos las gráficas; pero ,al igual que con la obtención de métricas, nos encontramos que a cada caso nuevo que añadimos, el código se vuelve más complejo. Por lo que los sacaremos también y lo metemos en una nueva función “createChart()” dentro de un nuevo fichero “a_charts.js”.

A esta función se le pasar los datos, ya en JSON, y el tipo de visualización en la que estamos; y nos devolverá la entidad que contiene la gráfica correspondiente para luego añadirla a la escena.

A esto, le añadimos también las nuevas líneas con el caso para crear una gráfica con el componente “geo3dbarchart”.

```

function createChart (type, data) {
    var chart = document.createElement('a-entity');

    if ( type == 'pie' ) {
        chart.setAttribute('geopiechart', {

```

```

        legend: true ,    // legend
        data: data        // data
    });

chart.setAttribute('position', { x: 0, y: 2, z: -4 });
chart.setAttribute('rotation', { x: 90, y: -20, z: 0 });
} else if (type == 'simple_bar') {
chart.setAttribute('geosimplebarchart', {
    legend: true ,    // legend
    axis: true ,      // axis
    data: data        // data
});

chart.setAttribute('position', { x: -2, y: 0, z: -5 });
chart.setAttribute('rotation', { x: 0, y: 0, z: 0 });
} else if (type == '3d_bar') {
chart.setAttribute('geo3dbarchart', {
    legend: true ,    // legend
    data: data        // data
});

chart.setAttribute('position', { x: 0, y: 0, z: -15 });
chart.setAttribute('rotation', { x: 0, y: 0, z: 0 });
};

return chart;
}

export { createChart };

```

Y por último, importamos al función al controlador y hacemos la llamada donde corresponde.

```

import { createChart } from './a_charts';
...
let chart = createChart(this.vis.params.type, data_json);

```

Con esto obtenemos la siguiente visualización en Kibana.

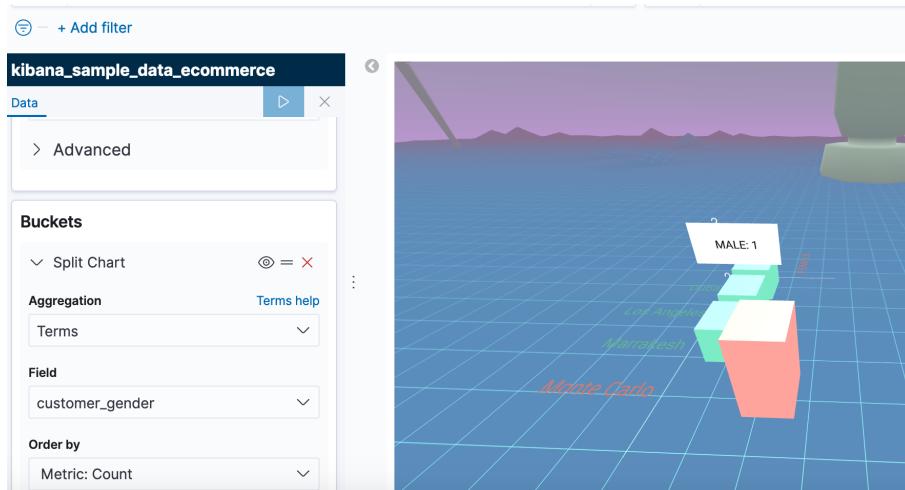


Figura 3.21: Resultado Visualización 3D Bars Chart.

Para este ejemplo, le hemos pedido que nos muestre las ventas que se han realizado en función del género de comprador y la ciudad donde se ha realizado la compra.

3.6.3. Visualización Bubbles

Por último, sólo nos queda crear la visualización de tipo bubbles. A partir de aquí será más sencillo pues al haber reestructurado el código, según van apareciendo nuevos casos, únicamente habrá que añadir las líneas correspondientes a dicho caso.

Al igual que hemos hecho anteriormente con el resto de visualizaciones, empezaremos determinando cómo vamos a tratar los datos.

Según la documentación de BabiaXR para el componente “geobubblechart”, el formato de nuestro JSON deberá ser el siguiente:

```
[{"key ":"David","key2 ":"2019","height":1,"radius":9},  
 {"key ":"David","key2 ":"2018","height":2,"radius":8},  
 ...  
 ]
```

Como vemos, en esta ocasión vamos a trabajar con dos claves: “key” y “key2”; con dos valores: “height” y “radius. Esto, en términos del editor de kibana, lo traducimos en dos bu-

kets y dos metrics. Una vez determinado esto, procedemos a crear la definición de esta nueva visualización en “bubble.js”.

```
export const bubbleDefinition = {

    name: 'bubble_chart',
    title: 'VR Bubble 3D',
    icon: 'heatmap',
    description: 'Bubbles with data (VR Chart).',
    visualization: VisController,

    visConfig: {
        defaults: {
            type: 'bubble',
        },
    },

    editorConfig: {
        schemas: new Schemas([
            {
                group: AggGroupNames.Metrics,
                name: 'metric',
                title: 'Y-axis',
                min: 2,
                max: 2,
                aggFilter: ['sum', 'count', 'cardinality',
                           'top_hits'],
                defaults: [{ schema: 'metric', type: 'count' }],
            },
            {
                group: AggGroupNames.Buckets,
                name: 'segment',
            }
        ])
    }
}
```

```

        title: 'Split Chart',
        min: 2,
        max: 2,
        aggFilter: [ '!geohash_grid', '!geotile_grid',
                      '!filter'],
    },
}

],
}
};


```

Y, al igual que hicimos con las anteriores visualizaciones, lo importamos a “kbn_aframe.js” y registramos la visualización.

```

import { bubbleDefinition } from './bubble';
...
visualizations.types.createBaseVisualization(bubbleDefinition);

```

Nos vamos a la parte del controlador; pero en esta ocasión directamente editaremos sobre las funciones de “getMetrics()” y “createChart()” que se encuentran en “metrics.js” y “a_charts.js” correspondientemente.

Siguiendo los pasos que seguimos con las otras visualización, añadimos las nuevas líneas correspondientes para estructurar los datos obtenidos de elasticsearch al formato que necesitamos para poder crear el componente en la función “getMetrics()”.

```

} else if (type == 'bubble'){
    metrics.push({
        "key": values[0].toString(),
        "key2": values[1].toString(),
        "height": values[2],
        "radius": values[3],
    });
}

```

Y por último, añadir el caso de esta nueva visualización y que nos cree el componente “geobubblechart”.

Obteniendo en kibana una visualización resultado con una gráfica de tipo bubble.

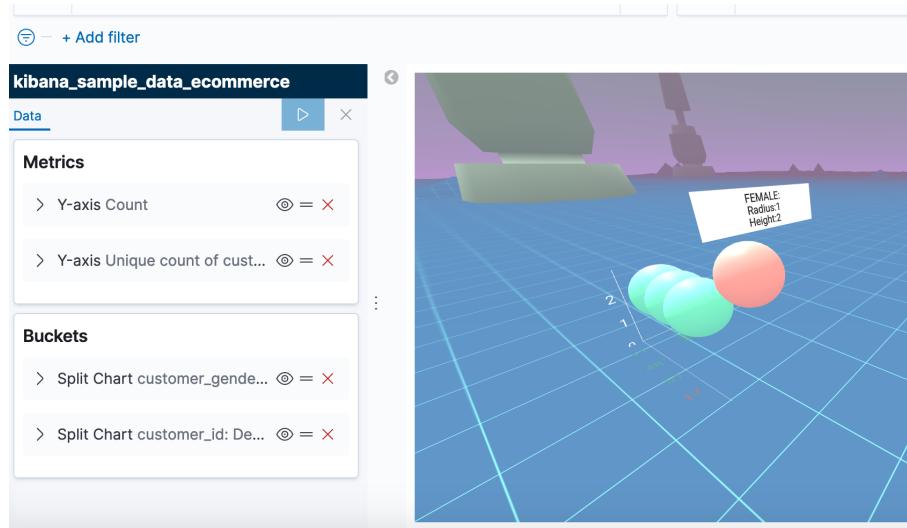


Figura 3.22: Resultado Visualización Bubble Chart

Para este ejemplo hemos tomado el número de ventas para determinar la altura, el contador uno por cada comprador y que nos los clasifique por id de usuario y según sexo.

Y tras haber añadido el último de los componentes que hay actualmente en BabiaXR, damos por finalizado este sprint y con ello, finalizamos la parte de desarrollo de este proyecto.