	UNIVERSIDAD DON BOSCO **FACULTAD DE INGENIERÍA** **ESCUELA DE COMPUTACIÓN**	
CICLO: 02	**GUIA DE LABORATORIO #09**	
	Nombre de la Práctica:	**Crud React Firebase**
	MATERIA:	**Diseño y Programación de Software Multiplataforma**

I. OBJETIVOS

Que el estudiante:

- Diseñe aplicaciones web utilizando funciones de React.
- Diseñe aplicaciones con navegación.
- Aprender hacer interfaz para que el usuario pueda interactuar con la aplicación.
- Aprender a crear componente personalizado.
- Aprender hacer interfaz para que el usuario pueda interactuar con la aplicación.
- Aprender a utilizar pipes dentro de la interfaz de usuario.
- Hacer uso de los componentes en el diseño de interfaz React.

Contenido

Render Props 3
Usa Render Props 3
Usando otras Props diferentes de render 8
Ten cuidado al usar Render Props con React.PureComponent 9
Modelo de datos de Cloud Firestore 11
Documentos 11
Colecciones 12
Referencias 13
Obtén actualizaciones en tiempo real con Cloud Firestore 13
QuerySnapshot 13
Async/Await en javaScript 13
2. Entorno 14
Ejemplo práctico: 15

II. INTRODUCCION TEORICA

Render Props

El término "render prop" se refiere a una técnica para compartir código entre componentes en React utilizando una propiedad cuyo valor es una función.

Un componente con un render **prop** toma una función que devuelve un elemento de React y lo llama en lugar de implementar su propia lógica de representación.

```
<DataProvider render={data => (
  <h1>Hello {data.target}</h1>
)}/>
```

Algunas bibliotecas que utilizan render props son React Router, Downshift y Formik. En este documento, discutiremos por qué las render props son útiles y cómo escribir las tuyas.

Usa Render Props

Los componentes son la unidad primaria de reutilización de código en React, pero no siempre es obvio cómo compartir el estado o el comportamiento que un componente encapsula en otros componentes que necesitan ese mismo estado.

Por ejemplo, el siguiente componente rastrea la posición del cursor en una aplicación web:

```
class MouseTracker extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        <h1>Move the mouse around!</h1>
        <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
```

```
    </div>
   );
  }
}
```

A medida que el cursor se mueve alrededor de la pantalla, el componente muestra sus coordenadas (x, y) en un <p>.

Ahora la pregunta es: ¿Cómo podemos reutilizar este comportamiento en otro componente? En otras palabras, si otro componente necesita saber la posición del cursor, ¿podemos encapsular ese comportamiento para poder compartirlo fácilmente con ese componente?
Como los componentes son la unidad básica de reutilización de código en React, intentemos refactorizar el código un poco para usar un componente <Mouse> que encapsule el comportamiento que necesitamos reutilizar en otro lugar.

```
// El componente <Mouse> encapsula el comportamiento que necesitamos...
class Mouse extends React.Component {
 constructor(props) {
  super(props);
  this.handleMouseMove = this.handleMouseMove.bind(this);
  this.state = { x: 0, y: 0 };
 }

 handleMouseMove(event) {
  this.setState({
   x: event.clientX,
   y: event.clientY
  });
 }

 render() {
  return (
   <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>

    {/* ...pero, ¿cómo renderizamos algo más que un <p>? */}
    <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
   </div>
  );
 }
}

class MouseTracker extends React.Component {
 render() {
```

```
    return (
      <>
        <h1>Move the mouse around!</h1>
        <Mouse />
      </>
    );
  }
}
```

Ahora, el componente <Mouse> encapsula todo el comportamiento asociado con la escucha de eventos mousemove y el almacenamiento de la posición (x, y) del cursor, pero aún no es realmente reutilizable.

Por ejemplo, digamos que tenemos un componente <Cat> que representa la imagen de un gato persiguiendo el cursor alrededor de la pantalla. Podríamos usar una propiedad <Cat mouse={{ x, y }}> para indicar al componente las coordenadas del cursor de manera que sepa dónde colocar la imagen en la pantalla.

Como primer paso, puedes intentar renderizar el componente <Cat> dentro del método render del componente <Mouse>, de esta manera:

```
class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      <img src="/cat.jpg" style={{ position: 'absolute', left: mouse.x, top: mouse.y }} />
    );
  }
}

class MouseWithCat extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }
```

```
 render() {
  return (
   <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>

    {/*
     Podríamos simplemente cambiar el <p> por un <Cat> aquí ... pero luego
     necesitaríamos crear un componente <MouseWithSomethingElse> separado
     cada vez que necesitamos usarlo, por lo que <MouseWithCat>
     no es realmente reutilizable todavía.
    */}
    <Cat mouse={this.state} />
   </div>
  );
 }
}

class MouseTracker extends React.Component {
 render() {
  return (
   <div>
    <h1>Move the mouse around!</h1>
    <MouseWithCat />
   </div>
  );
 }
}
```

Esta propuesta funcionará para nuestro caso de uso específico, pero no hemos logrado el objetivo de realmente encapsular el comportamiento de una manera reutilizable. Ahora, cada vez que queramos saber la posición del cursor para un caso de uso diferente, debemos crear un nuevo componente (es decir, esencialmente otro <MouseWithCat>) que renderice algo específicamente para ese caso de uso.

Aquí es donde entran en juego las render props: En lugar de codificar de forma fija un componente <Cat> dentro del componente <Mouse>, y cambiar efectivamente la salida de su método render, podemos proporcionar una función por medio props a <Mouse> que pueda utilizar para determinar dinámicamente lo que debe renderizar -una render prop.

```
class Cat extends React.Component {
 render() {
  const mouse = this.props.mouse;
  return (
```

```
    <img src="/cat.jpg" style={{ position: 'absolute', left: mouse.x, top: mouse.y }} />
   );
  }
}

class Mouse extends React.Component {
  constructor(props) {
   super(props);
   this.handleMouseMove = this.handleMouseMove.bind(this);
   this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
   this.setState({
    x: event.clientX,
    y: event.clientY
   });
  }

  render() {
   return (
    <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>

     {/*
      En lugar de proporcionar una representación estática de lo que <Mouse> renderiza,
      usa la `render prop` para determinar dinámicamente qué renderizar.
     */}
     {this.props.render(this.state)}
    </div>
   );
  }
}

class MouseTracker extends React.Component {
  render() {
   return (
    <div>
     <h1>Move the mouse around!</h1>
     <Mouse render={mouse => (
      <Cat mouse={mouse} />
     )}/>
    </div>
   );
```

```
  }
}
```

Ahora, en lugar de clonar efectivamente el componente <Mouse> y codificar de forma fija otra cosa en su método render para resolver un caso de uso específico, proporcionamos una render prop que <Mouse> pueda usar para dinámicamente determinar que renderizar.
Más concretamente, **una render prop es una prop que recibe una función que un componente utiliza para saber qué renderizar.**

Esta técnica hace que el comportamiento que necesitamos compartir sea extremadamente portátil. Para obtener ese comportamiento, genere un <Mouse> con una render prop que le diga qué renderizar con la posición (x, y) del cursor.

Una cosa interesante a tener en cuenta acerca de las render props es que puedes implementar la mayoría de los componentes de orden superior (HOC) utilizando un componente regular con una render prop. Por ejemplo, si prefiere tener un withMouse HOC en lugar de un componente <Mouse>, puede crear fácilmente uno usando un <Mouse> regular con una render prop:

```
// Si realmente quieres un HOC por alguna razón, puedes fácilmente
// crear uno usando un componente regular con una render prop!
function withMouse(Component) {
  return class extends React.Component {
    render() {
      return (
        <Mouse render={mouse => (
          <Component {...this.props} mouse={mouse} />
        )}/>
      );
    }
  }
}
```

Por lo tanto, usar una render prop hace que sea posible usar cualquier patrón.

Usando otras Props diferentes de render

Es importante recordar que solo porque el patrón se llama render props no tienes que usar una prop llamada render para usar este patrón. De hecho, cualquier prop que es una función que un componente utiliza para saber qué renderizar es técnicamente una “render prop” .
Aunque los ejemplos anteriores usan render, ¡podríamos usar la proposición children con la misma facilidad!

```
<Mouse children={mouse => (
```

```
  <p>The mouse position is {mouse.x}, {mouse.y}</p>
)}/>
```

Y recuerda, la propiedad children en realidad no necesita ser nombrada en la lista de "atributos" en su elemento JSX. En su lugar, puedes ponerlo directamente dentro del elemento!

```
<Mouse>
  {mouse => (
    <p>The mouse position is {mouse.x}, {mouse.y}</p>
  )}
</Mouse>
```

Verás esta técnica utilizada en la API de react-motion.
Ya que esta técnica es un poco inusual, probablemente querrás decir explícitamente que children debería ser una función en tus propTypes cuando diseñes una API como esta.

```
Mouse.propTypes = {
  children: PropTypes.func.isRequired
};
```

TEN CUIDADO AL USAR RENDER PROPS CON REACT.PURECOMPONENT

El uso de una render prop puede no aprovechar la ventaja del uso de React.PureComponent si crea la función dentro del método render. Esto se debe a que la comparación de propiedades poco profundas siempre devolverá false para las nuevas props, y cada render en este caso generará un nuevo valor para la render prop.

Por ejemplo, continuando con nuestro componente <Mouse> de los ejemplos anteriores, si Mouse extendiera React.PureComponent en lugar de React.Component, nuestro ejemplo se vería así:

```
class Mouse extends React.PureComponent {
  // Misma implementación que la anterior...
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>

        {/*
          ¡Esto está mal! El valor de la `render prop`
```

```
        será diferente en cada render.
      */}
      <Mouse render={mouse => (
        <Cat mouse={mouse} />
      )}/>
    </div>
   );
  }
}
```

En este ejemplo, cada vez que se renderiza <MouseTracker>, genera una nueva función como el valor de la propiedad <Mouse render>, negando así el efecto de <Mouse> extendiendo React.PureComponent en primer lugar!

Para solucionar este problema, a veces se puede definir la prop como un método de instancia, así:

```
class MouseTracker extends React.Component {
  // Definido como un método de instancia, `this.renderTheCat` siempre
  // se refiere a la *misma* función cuando la usamos en render
  renderTheCat(mouse) {
    return <Cat mouse={mouse} />;
  }

  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={this.renderTheCat} />
      </div>
    );
  }
}
```

En los casos en los que no puede definir la propiedad de forma estática (por ejemplo, porque necesita encerrar las props y/o el estado del componente), el <Mouse> debería extender React.Component en su lugar.

Modelo de datos de Cloud Firestore

Cloud Firestore es una base de datos NoSQL orientada a los documentos. A diferencia de una base de datos SQL, no hay tablas ni filas. En su lugar, almacenas los datos en *documentos*, que se organizan en *colecciones*.

Cada *documento* contiene un conjunto de pares clave-valor. Cloud Firestore está optimizado para almacenar grandes colecciones de documentos pequeños.

Todos los documentos se deben almacenar en colecciones. Los documentos pueden contener *subcolecciones* y objetos anidados, y ambos pueden incluir campos primitivos como strings o tipos de objetos complejos como listas.

Las colecciones y los documentos se crean de manera implícita en Cloud Firestore. Solo debes asignar datos a un documento dentro de una colección. Si la colección o el documento no existen, Cloud Firestore los crea.

Documentos

En Cloud Firestore, la unidad de almacenamiento es el documento. Un documento es un registro liviano que contiene campos con valores asignados. Cada documento se identifica con un nombre.

Un documento que representa a un usuario `alovelace` puede tener el siguiente aspecto:

- class alovelace

```
first : "Ada"
last : "Lovelace"
born : 1815
```

Nota: Cloud Firestore admite diversos tipos de datos para los valores, como booleanos, números, strings, puntos geográficos, BLOB binarios y marcas de tiempo Además, puedes usar arreglos u objetos anidados, llamados mapas, para estructurar datos dentro de un documento.

Los objetos complejos anidados en un documento se llaman mapas. Por ejemplo, podrías estructurar el nombre del usuario del ejemplo anterior con un mapa como este:

- class alovelace

```
name :
   first : "Ada"
   last : "Lovelace"
born : 1815
```

Tal vez te parezca que los documentos son muy similares a JSON. De hecho, básicamente son JSON. Existen algunas diferencias (por ejemplo, los documentos admiten tipos de datos adicionales y su

tamaño se limita a 1 MB), pero en general, puedes tratar los documentos como registros JSON livianos.

Colecciones

Los documentos viven en colecciones, que simplemente son contenedores de documentos. Por ejemplo, podrías tener una colección llamada **users** con los distintos usuarios de tu app, en la que haya un documento que represente a cada uno:

- **collections_bookmark users**
  - **class alovelace**

    first : "Ada"
    last : "Lovelace"
    born : 1815

  - **class aturing**

    first : "Alan"
    last : "Turing"
    born : 1912

Cloud Firestore no usa esquemas, por lo que tienes libertad total sobre los campos que pones en cada documento y los tipos de datos que almacenas en esos campos. Los documentos dentro de la misma colección pueden contener distintos campos o almacenar distintos tipos de datos en esos campos. Sin embargo, se recomienda usar los mismos campos y tipos de datos en varios documentos, de manera que puedas consultarlos con mayor facilidad.

Una colección contiene solo documentos. No puede contener campos sin procesar con valores de manera directa ni tampoco otras colecciones. (Consulta los datos jerárquicos para ver una explicación sobre cómo estructurar datos más complejos en Cloud Firestore).

Los nombres de documentos dentro de una colección son únicos. Puedes proporcionar tus propias claves, como los ID de usuario, o puedes dejar que Cloud Firestore cree ID aleatorios de forma automática.

No es necesario "crear" ni "borrar" las colecciones. Cuando se crea el primer documento de una colección, esta pasa a existir. Si borras todos los documentos de una colección, esta deja de existir.

Referencias
Cada documento de Cloud Firestore se identifica de forma única por su ubicación dentro de la base de datos. El ejemplo anterior muestra un documento alovelace en la colección users. Para hacer referencia a esta ubicación en tu código, puedes crear una *referencia* a ella.

Node.js

```
let usersCollectionRef = db.collection('users')
```

Obtén actualizaciones en tiempo real con Cloud Firestore

Puedes detectar un documento con el método **onSnapshot**(). Una llamada inicial con la devolución de llamada que proporcionas crea una instantánea del documento de inmediato con los contenidos actuales de ese documento. Después, cada vez que cambian los contenidos, otra llamada actualiza la instantánea del documento.

Node.js

```
let doc = db.collection('cities').doc('SF');

let observer = doc.onSnapshot(docSnapshot => {
  console.log(`Received doc snapshot: ${docSnapshot}`);
  // ...
}, err => {
  console.log(`Encountered error: ${err}`);
});
```

QuerySnapshot

La **QuerySnapshot** contiene los resultados de una consulta. Puede contener cero o más DocumentSnapshot objetos.

Las clases de Cloud Firestore no están diseñadas para ser subclasificadas, excepto para su uso en simulacros de prueba. La subclasificación no es compatible con el código de producción y las nuevas versiones del SDK pueden romper el código que lo hace.

Async/Await en javaScript

Las promesas en javascript son una manera de conseguir que el código asíncrono se comporte como si fuera síncrono, facilitándonos la vida enormemente. Sin embargo, esto tiene algunas limitaciones.

Async/Await es una propuesta para extender la sintaxis de **javaScript** con las palabras reservadas, cuyo uso permitirá -cuando se estandarice- tratar las funciones que devuelven promesas en nuestro código como si fueran funciones síncronas que devuelven directamente valores en vez de promesas. Aunque actualmente ningún navegador soporta **async/await**, podemos utilizar este mecanismo transpilando código que lo utilice a código que sí entiendan los navegadores.

Cómo funciona:

Por ejemplo, sea **getFilm** una función que se conecta a un hipotético servicio web que devuelve una película al azar, y **getMain**()otra función promisificada del mismo servicio que devuelve el nombre del protagonista de una película dada. Con la sintaxis de **Async/Await** podemos hacer lo siguiente:

```
1
2     async function queue(){
3         var film = await getFilm(); //Supongamos que toca 'Matrix'
4         var main = await getMain(film); //Neo
5         console.log(main);
6     }
7     queue();//escribirá 'Neo' en la consola.
8
```

En vez de tener que utilizar la respuesta de la primera petición en un incómodo then(function(){}), usando la palabra especial **await** podemos usar **getFilm()** y **getMain()** como si devolvieran valores síncronos en vez de promesas. Esto es mucho más cómodo de escribir, y sobre todo resulta en código mucho más legible.

2. ENTORNO

Cualquier editor de texto y un navegador **debería** valer para ejecutar **snippets de javaScript**. No obstante,**async/await** una **especificación** de **ecmaScript 7**, y todavía no es soportada de forma nativa por ningún navegador ni por node.js. Para poder utilizar async/await en nuestro código a día de hoy, tendremos que usar algún **transpilador** para convertir nuestro código con la sintaxis de **async/await**

Ejemplo práctico:

1. Crear nuestro primer Proyecto:

```
create-react-app crudreactfirebase
```

2. Instalar Firebase :

```
npm install  - save firebase
```

3. Instalar paquete de notificaciones:

```
npm install --save react-toastify
```

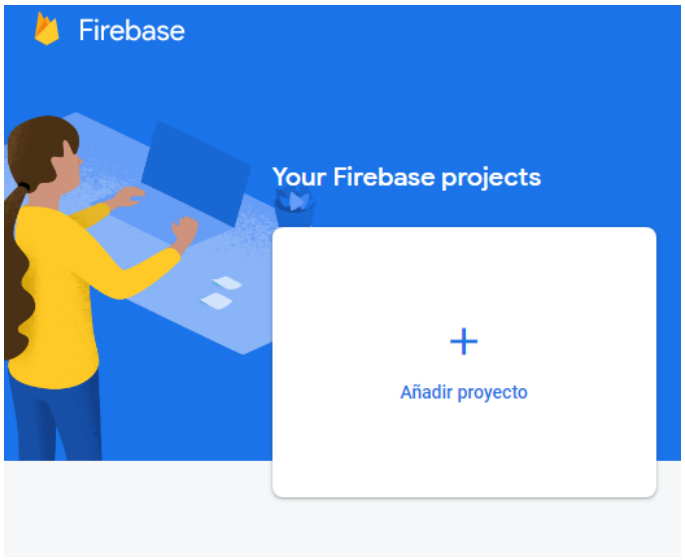
4. Instalar paquete de los estilos:

```
npm install bootswatch
```

5. Una vez hecho esto, ahora podemos configurar Firebase.

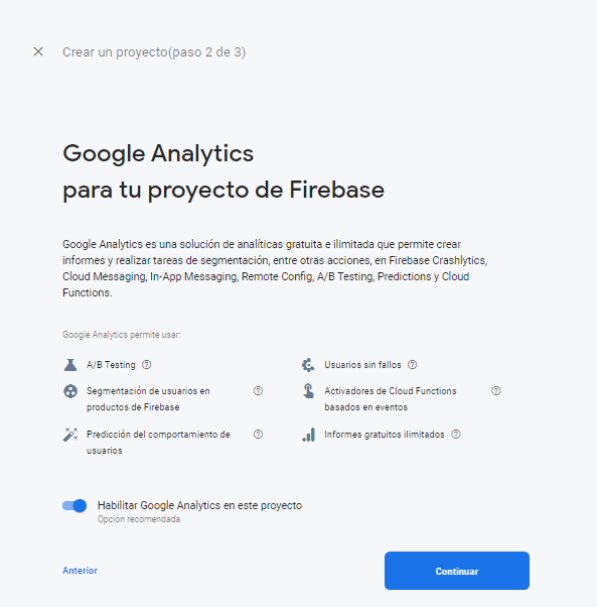
En primer lugar, necesitamos crear un nuevo archivo dedicado a Firebase. Cree un archivo llamado.**Firebase.jsx** el directorio raíz "src"

Ahora, vaya al sitio web de Firebase y haga clic en el botón **Comenzar** . Sera llevado a una página donde podrá crear un nuevo proyecto. "**CrudReactFirebase**"

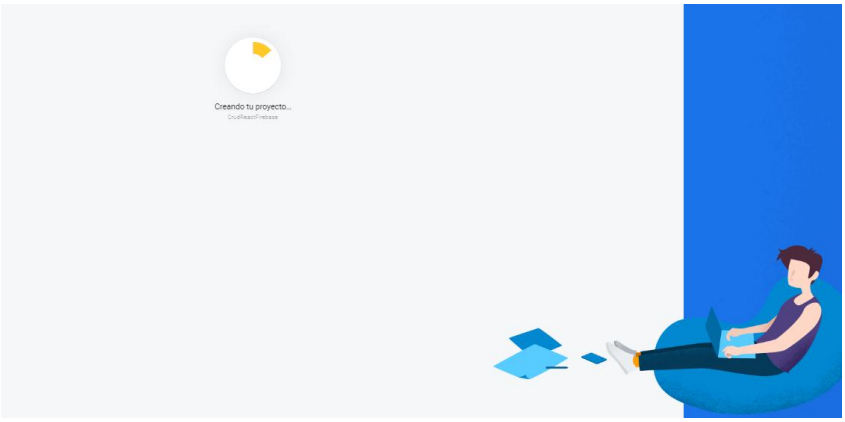


Elegir el nombre del proyecto:

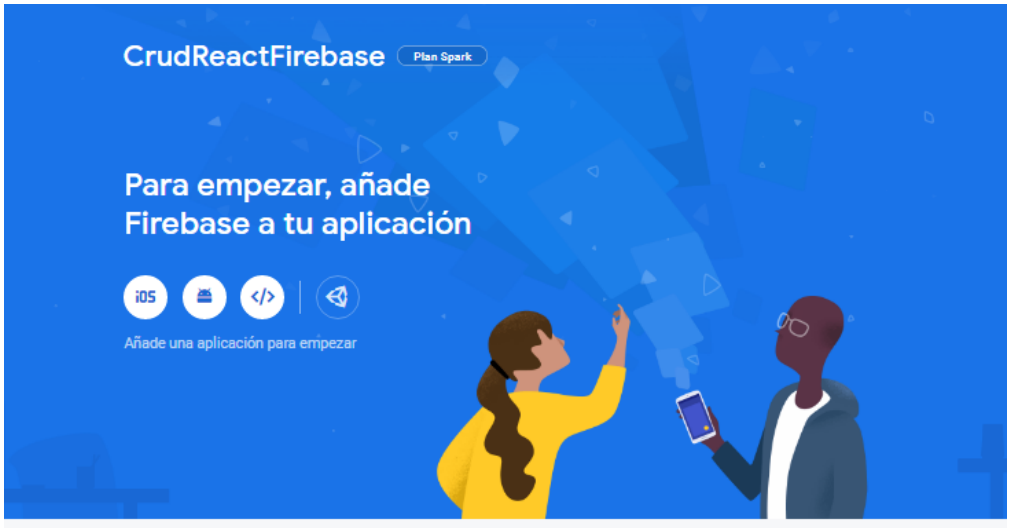
Desactivar Google Analytics



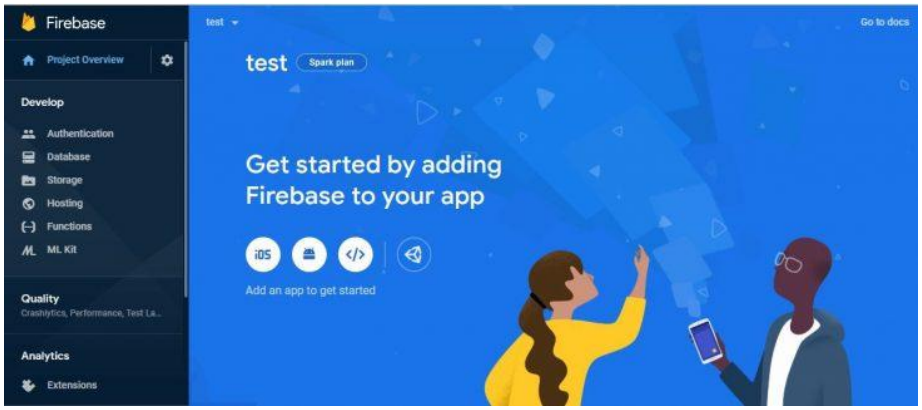
Creando el proyecto



Luego Agregar nuestra aplicación web 

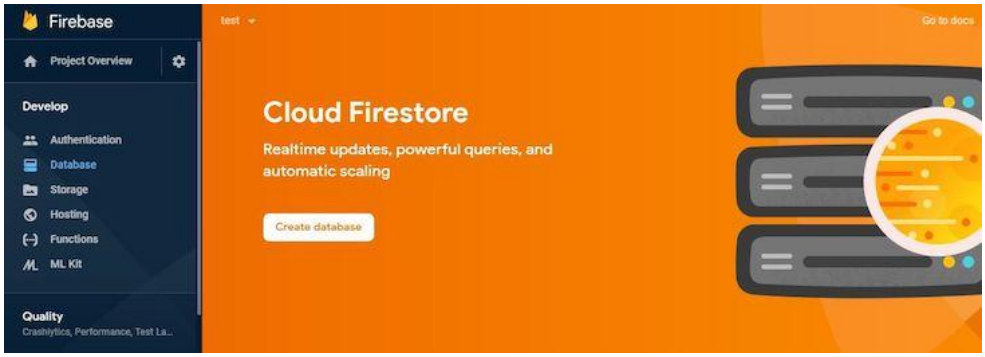


Una vez que haya terminado, debería ser llevado a una página de panel similar a la imagen a continuación.

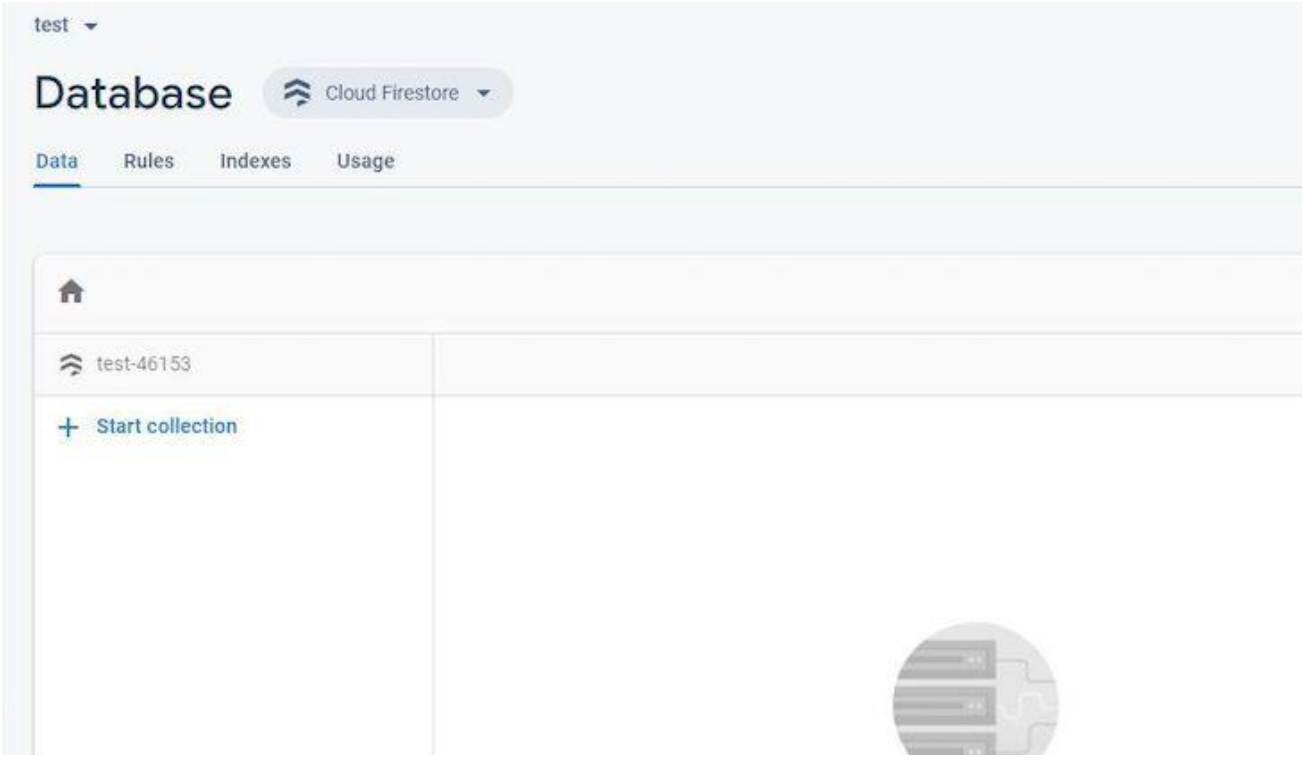


Usaremos dos servicios de Firebase para este proyecto: el servicio de **Cloud Firestore**. Primero configuremos **Cloud Firestore.**

Seleccione la opción **Base de datos** en el menú lateral. Ahora debería poder elegir cualquiera de los dos servicios de base de datos que proporciona Firebase: Cloud Firestore o Realtime Database. usaremos **Cloud Firestore.**



Ahora crea una base de datos de Cloud Firestore. Ahora debería ver las reglas de seguridad de su base de datos. Elija **Iniciar en modo de prueba** . Debería tener una base de datos vacía lista para usar. Debe tener un aspecto como este:



Ahora que se configuraron **Cloud Firestore**, necesitamos obtener los detalles de configuración de nuestro proyecto. Esto es necesario para vincular nuestro código a nuestro proyecto de Firebase.

Para obtener los detalles de nuestra configuración de Firebase, regrese a la página de descripción general del proyecto y agregue una aplicación web al proyecto de Firebase. Después de registrar la aplicación, debería obtener los detalles de configuración en forma de un objeto JavaScript.


```
const firebaseConfig = {
  apiKey : 'AIzaXXXXXXXXXXXXXXXXXXXXXX' ,
  authDomain : 'test-XXXX.firebaseapp.com' ,
  databaseURL : 'https://test-XXXXXX.firebaseio.com' ,
  projectId : 'test-XXXX' ,
  storageBucket : 'prueba -XXXX.appspot.com ' ,
  messagingSenderId : ' XXXXXXX ' ,
  appId : "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
};
```

6. Repositorio : https://github.com/AlexanderSiguenza/crudreactfirebase.git

7. Hosting : https://reactcrudfirebase-73847.web.app/

8. La explicación estará en un video que les compartiré posteriormente.

9. La pantalla debería verse de la siguiente forma

V. DISCUSION DE RESULTADOS

1. Deberán de replicar el siguiente ejercicio, pero para la tabla **empleado (id, nombre, apellido, cargo), deben de colocarlo en un hosting.**

VII. BIBLIOGRAFIA

- React, una biblioteca de JavaScript. (2013-2020). Facebook, Inc. Jordan Walke. Recuperado de https://es.reactjs.org/docs/getting-started.html

- https://firebase.google.com/docs/reference/js