EXERCISES QUE TRABAJAREMOS EN LA CUE:

- EXERCISE 1: BEFORECREATE.
- EXERCISE 2: LIFECYCLE - CREATED.
- EXERCISE 3: LIFE CYCLE (BEFOREMOUNT - MOUNTED).
- EXERCISE 4: LIFECYCLE (BEFOREUPDATE - UPDATED).
- EXERCISE 5: LIFECYCLE (BEFOREDESTROY - DESTROYED).

EXERCISE 1: BEFORECREATE

beforeCreate():

Cuando usas este ciclo de vida, el método se disparará antes que cualquier elemento de nuestro componente, y tendremos acceso a su instancia misma, pero NO acceso a datos, eventos, métodos.

```
<script>
export default {
  name:"LifeCycleComponent",
  data: function(){
    return {
      number:0,
    }
  },
  //methods,
  //lifecycles
  beforeCreate(){
    //retorna la instancia del componente
    console.log(this)
    //retorna indefinido ya que la data a un no se inicia.
    console.log(this.number)
  },
}
</script>
```

EXERCISE 2: LIFECYCLE - CREATED

Created():

Es el segundo hook en ser llamado, y es ideal para generar llamada a un endpoint. En este caso, nos da un tiempo de cargado antes de que se renderice algo. Aún no podemos leer DOM, pero si acceder a la data, propiedades computadas, métodos, entre otros.

Vamos a utilizar la API de Rick and Morty, y ocuparemos la librería Axios.

```
<template>
  <div>
    <ul>
      <li
      v-for="character in characters"
      :key="character.id"
      >
      {{character.name}}
      </li>
    </ul>
  </div>
</template>

<script>
import axios from "axios"
export default {
  name:"LifeCycleComponent",
  data: function(){
    return {
      number:0,
      characters:[],
    }
  },
  //methods,
  //lifecycles
  created(){
    // console.log(this)
    // console.log(this.number)
    axios.get("https://rickandmortyapi.com/api/character")
      .then(resp=>{
        console.log(resp)
        this.characters = resp.data.results
      })
      .catch(error=>{
        console.log(error)
      })
  }
```

```
}
</script>

<style scoped>
  div{
    border: 1px solid blue;
    width:60%;
    margin:0 auto;
  }
</style>
```

Axios

Es una librería que, a modo de cliente, nos permite hacer peticiones HTTP a un servidor. Las tareas son realizadas por medio del objeto Promise() de JavaScript, y junto a Node JS, hace que Axios tenga compatibilidad con las herramientas y Frameworks que también son compatibles con Node JS, como React JS, Vue JS, entre otros.

Axios permite hacer solicitudes a un servidor, con métodos como: GET, DELETE, POST, PUT, PATCH, HEAD Y OPTIONS.

Al hacerlas, podemos obtener determinados datos, y manipularlos para lograr el objetivo que tengamos en mente.

Los datos que recibimos del servidor vienen en forma de objetos JavaScript, y Axios los serializa en formato JSON, que es uno de los más populares para el intercambio de datos en las aplicaciones, y es muy dinámico, ya que podemos llamar a modo de strings, booleanos, enteros, entre otros, datos que pueden ser de tipo imágenes, números, o videos.

Para trabajar con Axios, usamos la API de promesas. Esto quiere decir que, al recibir la respuesta del servidor, se ejecutará una función callback configurada en "then", y cuando se produzca un error (por ejemplo, una respuesta con status 404 de recurso no encontrado), se ejecutará la función callback definida por el "catch".

API

O Application Programming Interface en inglés, es un conjunto de funciones, subrutinas y procedimientos, que ofrece una biblioteca para ser utilizada por otro software. Corresponde a una capa de abstracción, es decir, a un intermediario entre el cliente y una aplicación o software.

Imaginemos que vamos a una pizzería, y compramos una pizza con ingredientes elegidos por nosotros. Piensa en una API como la carta de ingredientes, donde te proveen de todas las

opciones de masas, quesos, entre otros, a agregar. Una vez que especificas los que quieres, la cocina se encarga de preparar la pizza y de entregarte el producto terminado. Nosotros no tenemos que saber exactamente cómo prepararon la pizza, solo nos importa recibir nuestro producto caliente y listo para consumir.

De manera análoga, una API lista una variedad de operaciones que los desarrolladores pueden usar, además de una descripción de lo que hacen. Pero éste no necesariamente necesita conocer cómo se realiza cada operación, solo saber que ya está disponible para usarse en su aplicación, y los parámetros que debe entregarle a la API.

REST

Representational State Transfer, es un estilo de arquitectura, para manejar un requerimiento desde el frontend de un servicio web, hacia el servidor. Una API REST, por otro lado, es un servicio web que utiliza la arquitectura REST para realizar la comunicación entre un servicio web, y un servidor.

Imaginemos ahora a un farmacéutico, que está usando su computador en la farmacia, para acceder al stock de un medicamento. La aplicación web a la que tendrá que entrar para realizar dicha consulta, utilizará un API REST para hacer la consulta a la base de datos de la farmacia, luego la aplicación parseará los datos, y se los presentará en el sitio web.

Las API REST utilizan peticiones HTTP (Protocolo de transferencia de hipertexto), para procesar los requerimientos de datos.

EXERCISE 3: LIFE CYCLE (BEFOREMOUNT - MOUNTED)

beforeMount:

Esta life cycle hook se llama justo antes de que el componente sea montado en el DOM, cuando la función “render” sea llamada por primera vez. Se puede utilizar para inicializar variables.

Mounted():

En esta fase ya se generó el DOM, pero fue reemplazado con el VDOM, y nos puede servir para inicializar librerías externas. En esta fase del ciclo de vida, ya podemos manipular el DOM.

```
<template>
  <div>
    <div class="ct-chart ct-golden-section" id="chart1"></div>
  </div>
</template>
<script>
import Chartist from 'chartist'
export default {
  name:"LifeCycleComponent",
  data: function(){
    return {
      number:0,
      labels:[1,2,3,4],
      series:[[100,120,180,200]],
    }
  },
  //methods, lifecycles
  beforeMount(){
    this.number=5;
    //no tenemos acceso al dom
    console.log(this.$el,'beforemount')
  },
  mounted(){
    console.log(this.$el,'mounted')
    new Chartist.Line('#chart1',{
      labels:this.labels,
      series:this.series,
    })
  }}
</script>
<style scoped>
  div{
    border: 1px solid blue;
    width:60%;
    margin:0 auto;
  }
</style>
```

EXERCISE 4: LIFECYCLE (BEFOREUPDATE - UPDATED)

BeforeUpdate:

Una vez montado el componente, si se hace algún cambio en la data, éste gatillará a beforeUpdate. Es un paso antes a renderizar los datos en el DOM.

Updated():

Cuando el DOM se actualiza, es gatillado Updated. Debemos recordar que cualquier dato que esté dentro de data, y que tenga algún cambio de valor, gatillará ambos métodos.

```
<template>
  <div>
    <h1 ref="counter">{{number}}</h1>
    <button @click="number +=1">sumar</button>
  </div>
</template>
<script>
export default {
  name:"LifeCycleComponent",
  data: function(){
    return {
      number:0,

    }
  },
  //methods,
  //lifecycles
  mounted(){
    console.log(this.number,'mounted');
  },
  beforeUpdate(){
    console.log(this.number,'beforeUpdate');
    console.log(this.$refs.counter.textContent);//0
  },
  updated(){
    console.log(this.number,'updated');
    console.log(this.$refs.counter.textContent);//1
  }
}
</script>
<style scoped>
  div{
    border: 1px solid blue;
    width:60%;
    margin:0 auto;
  }
</style>
```

EXERCISE 5: LIFECYCLE (BEFOREDESTROY - DESTROYED)

BeforeDestroy:

Este lifecycle hook, es llamado antes de destruir el componente. Esto nos puede servir para limpiar ciertos eventos generados por librerías externas, y emitir algún event (emit).

Destroyed:

Esta función es llamada después de que se ha destruido una instancia de Vue. Cuando se llama a este life cycle, todas las directivas de la instancia de Vue se han desvinculado.

```
<template>
  <div>
    <h1 ref="counter">{{number}}</h1>
    <button @click="number +=1">sumar</button>

  </div>
</template>

<script>

export default {
  name:"LifeCycleComponent",
  data: function(){
    return {
      number:0,

    }
  },
  //methods,
  //lifecycles
  beforeDestroy(){
    console.log("antes de eliminarse");
  },
  destroyed(){
    console.log("componente eliminado")
  }
}
</script>
<style scoped>
  div{
    border: 1px solid blue;
    width:60%;
    margin:0 auto;
  }
</style>
```