Avance 2

En este avance se van a crear los eventos que van a cargar dinámicamente los datos en el html y a manejar errores, además de modularizar la aplicación con una arquitectura.

Lo primero que se requiere es pasar el id de cada receta al URL y que, cada que cambie el **id**, cambie el producto del **div** principal **recipe.**

1. Agrega al final del archivo **controller.js** un evento con **addEventListener** a la ventana y pásale como parámetros el evento **hashchange y showRecipe**.
2. Con esto, ahora se llamará a la función **showRecipe** cada vez que el hash cambie, por tanto, se puede eliminar la llamada a la función **showRecipe**.
3. Para poder probar la funcionalidad, en el archivo **index.html**, en el área de resultados, crea un par de enlaces con hash de diferentes recetas, parecidos a estos ejemplos:

```
<a href="#5ed6604591c37cdc054bc886">Recipe 1</a>
<a href="#5ed6604591c37cdc054bccde">Recipe 2</a>
```

4. Si observas en la URL, cuando haces clic en cada uno de los enlaces, el hash cambia, sin embargo, no pasa nada en la página. Para que esto se vea reflejado en la página, haz lo siguiente:
    a. En la función **showRecipe**, dentro del try, declara una variable **id** y asígnale el método **window.location.hash**.
    b. Imprime en la consola el resultado.
    c. Como no se requiere leer desde el primer carácter, agrégale el método slice(1) al final de la función del inciso a.
    d. Modifica en la función de búsqueda la URL estática para que ahora reciba la URL dinámico, pasándole la variable ${id}.
5. Hasta aquí todo parece funcionar, pero si se copia la URL completa y la pones en una ventana del navegador nueva, no va a tener el funcionamiento adecuado. Para crear la funcionalidad esperada, haz lo siguiente:
    a. Además del evento agregado al final del archivo **controller.js**, crea uno adicional, pero pásale como parámetro el evento **load** y **showRecipe**.
    b. Como las funciones son iguales y se podrían tener más, optimiza el código creando un arreglo con los elementos ‘haschange’,’load’ y aplícale el método foreach, el cual recibirá como parámetro una función flecha que, a su vez, recibe el evento (ev) y en el cuerpo de la función solo agrégale el evento **addEventListener** que recibirá como parámetros el evento (**ev**) y la función del controlador **showRecipe**.

Avance 2

6. Ahora, si no le pasas ningún id, te muestra un error y se queda esperando eternamente. Para solucionarlo, después de la declaración de la variable id, valida si la variable id no existe, cuando esto suceda, solamente dale un return.

Es momento de organizar el código desarrollado. Para ello, se va a utilizar la arquitectura MVC, dicho en otras palabras, vamos a modularizar el código.

1. Modelo: aquí se escribirá el modelo completo y tendrá objetos para la receta, la búsqueda y los marcadores y una función para cargar las recetas. Para realizar esta funcionalidad, haz lo siguiente:
    a. En la carpeta js, crea el archivo model.js y dentro del cuerpo del objeto agrega lo siguiente:
        i. Un objeto state que, a su vez, tendrá dentro de él un objeto recipe vacío.
        ii. Exporta este estado para que lo pueda visualizar el controlador.
        iii. Crea la función asíncrona loadRecipe que será la encargada de obtener la receta de la API.
        iv. Exporta la función loadRecipe.
        v. Toma el archivo controller.js y coloca dentro del cuerpo de la función loadRecipe lo siguiente:
            1) Dentro del try:
                a. La declaración de **res**.
                b. La declaración de data.
                c. La validación del estado de res.
                d. El objeto recibe:
                    i. La desestructuración de recipe.
                    ii. La impresión en consola de recipe.
            2) Dentro del catch que recibe como parámetro el error (err):
                a. Envía a una alerta el error.
            3) Dentro del modelo, realiza las siguientes modificaciones para ajustar el código:
                a. Pasa como parámetro a la función asíncrona la variable id.
                b. Declara como const el objeto recipe.
                c. Antepón el objeto state al recipe desestructurado.
                d. En el console.log, también registra a recipe como objeto de state.
    b. En el archivo **controller.js**, realiza las siguientes modificaciones:
        i. Importa todas las funciones como **model** desde el archivo model.js.
        ii. Manda a llamar la función **model.loadRecipe** con id como parámetro justo en donde se cortó el código que llevas al archivo **model.js** (como es una función asíncrona, recuerda colocar el await).
        iii. Declara una constante **recipe** desestructurada y asígnale **model.state**.

Avance 2

c. Prueba hasta aquí tu código, este debe continuar trabajando como hasta antes de la modularización.

2. Para separar la vista del controlador, haz lo siguiente:
   a. Dentro de la carpeta js, ahora crea la carpeta **views** y dentro de ella crea el archivo RecipeView.js para la vista de las recetas.
   b. Dentro del archivo **RecipeView**, realiza lo siguiente:
      i. Crea la clase **RecipeView** con las siguientes características:
         1. Declara un elemento privado **parentElement** y trae del archivo **controller.js** el valor del **recipeContainer**.
         2. Declara el elemento privado data.
         3. Declara el método **render** y pásale como parámetro **data.**
         4. Dentro de **render**, declara **this.#data** e iguálalos al parámetro que se acaba de recibir.
         5. Crea la constante markup e instancia el método **generateMarkup** con **this.#generateMarkup.**
         6. Declara un método privado **generateMarkup** y coloca dentro de su cuerpo todo lo que tenga la declaración de la variable **markup** en el archivo controller.js, pero regrésalo en un return.
         7. Cambia todas las variables **recipe.x** por **this.#data.x**.
         8. Quita el **innerHTML** y el **insertAdjacentHTML** de **#generateMarkup**.
         9. Para el limpiar el **parentElement**:
            a. Crea el método **#clean** y limpia al elemento utilizando this.#parentElement.innerHTML = '';.
            b. Instáncialo desde el **render** con **this.#clear**;

         10) Para el **insertAdjacentHTML** puedes utilizar la misma instrucción en el **render**, solo referenciando al **parentElement** de la siguiente manera: this.#parentElement.insertAdjacentHTML('afterbegin', markup);.

         11) Trae del archivo **controller.js** la función **renderSpinner** y colócala dentro de la clase **RecipeView**:
            a. Conviértela en un método público.
            b. Elimina el parámetro que se le pasa.
            c. Cambia el **parentEl** por **this.#parentElement**.
            d. Corta la línea que importa los íconos del archivo **controller.js** y pégala en la parte superior de **RecipeView.js.**
            e. Modifica la instancia del **renderSpinner** indicando que proviene de **renderView** y elimina el parámetro.

Avance 2

ii. Exporta por defecto no la clase directamente, sino una instancia de la clase utilizando **new RecipeView** para mantener la privacidad de sus elementos.

c. Dentro del archivo **controller.js**, realiza las siguientes modificaciones:
   i. Importa **recipeView** desde **./view/recipeview.js.**
   ii. Cambia el nombre de la función **showRecipe** por **controlRecipes**.
   iii. Elimina la instrucción **const** { recipe }= model.state;, ya no se considera necesaria.
   iv. Instancia **recipeView.render** y pásale como parámetro model.state.recipe en el lugar donde se cortó el código que se colocó en la vista.

d. Es momento de corregir un aspecto visual para que en los ingredientes, en lugar de presentar 0.5 coups, diga ½ coups. Para ello:
   i. Instala con ndm la librería **fractional.**
   ii. En el archivo **recipeView**, importa **Fraction** desde fractional utilizando la desestructuración.
   iii. Instancia **Fraction** en el campo cantidad de la siguiente manera:
      1) Valida con un operador ternario que exista un valor para la cantidad, si existe, entonces:
         new Fraction(ing.quantity).toString();
      2) Si no existe, envía una cadena vacía.

e. Prueba tu aplicación y debería seguir funcionando correctamente.

3. Crea un archivo de configuración que contendrá todas las variables que se utilizarán durante todo el proyecto y que son responsables de definir algunos datos importantes:
   a. Crea el archivo **config.js.**
   b. Declara y exporta la constante **API_URL** y asígnale la URL de la API que se está consumiendo (hasta antes del **id**).
   c. En el archivo **model.js**, importa la constante API_URL (colócalo entre llaves, ya que se van a importar otras variables) desde el archivo **config.js.**
   d. Modifica la variable **res** y utiliza la variable **API_URL.**

4. Crea un archivo en donde se almacenen funciones que se reutilizarán una y otra vez en el proyecto:
   a. En la carpeta **js**, crea un archivo **helpers.js** y agrega lo siguiente:
      i. Toma la declaración de **res**, la declaración de data y la validación del parámetro **ok** de **res** del archivo **model.js** de **loadRecipe** y llévalas al nuevo archivo **helpers.js**.
      ii. Crea y exporta la función asíncrona **getJSON** que recibe un URL como parámetro
      iv. La definición de **data** queda igual.
      v. La validación queda igual.
      vi. Haz que la función retorne el valor de data.

Avance 2

vii. En el catch, como lo hemos hecho en otras funciones, recibe el parámetro error y utiliza la instrucción **throw error**.

viii. Dentro del archivo **model.js**, sustituye la declaración de la función asíncrona **data** (no olvides el await), pásale como parámetro la URL armada con el **API_URL** y el **id**.

ix. Optimiza el manejo de errores en la función **loadRecipe** y, en lugar de mandar un alert, envía un mensaje de error personalizado a la consola, podría ser del tipo:

console.log(`${err} 💥💥💥💥`);

x. Prueba el nuevo funcionamiento del error.

b. Mueve la función **timeout**, que básicamente lo que hace es devolver una promesa que rechazará después de cierta cantidad de tiempo:

i. Modifica la función **getJSON**. Dentro de su cuerpo, declara una constante **fetchPro** y asígnale la función de búsqueda que recibe el URL como parámetro.

ii. Modifica la función **res** para que quede de esta manera:

```
const res = await Promise.race([fetchPro,timeout(TIMEOUT_SEC)]);
```

iii. Como podrás ver, hace uso de un TIMEOUT_SEC, este debes declararlo en el archivo **config.js** y asignarle el valor de 5.

Consideraciones

Instructor: Posiblemente a algunos de los aprendedores les cueste trabajo utilizar las funciones flecha, refuerza este tema.

Aprendedor: Asegúrate de saber utilizar las funciones flecha, ya que son las más fáciles de utilizar en los proyectos.