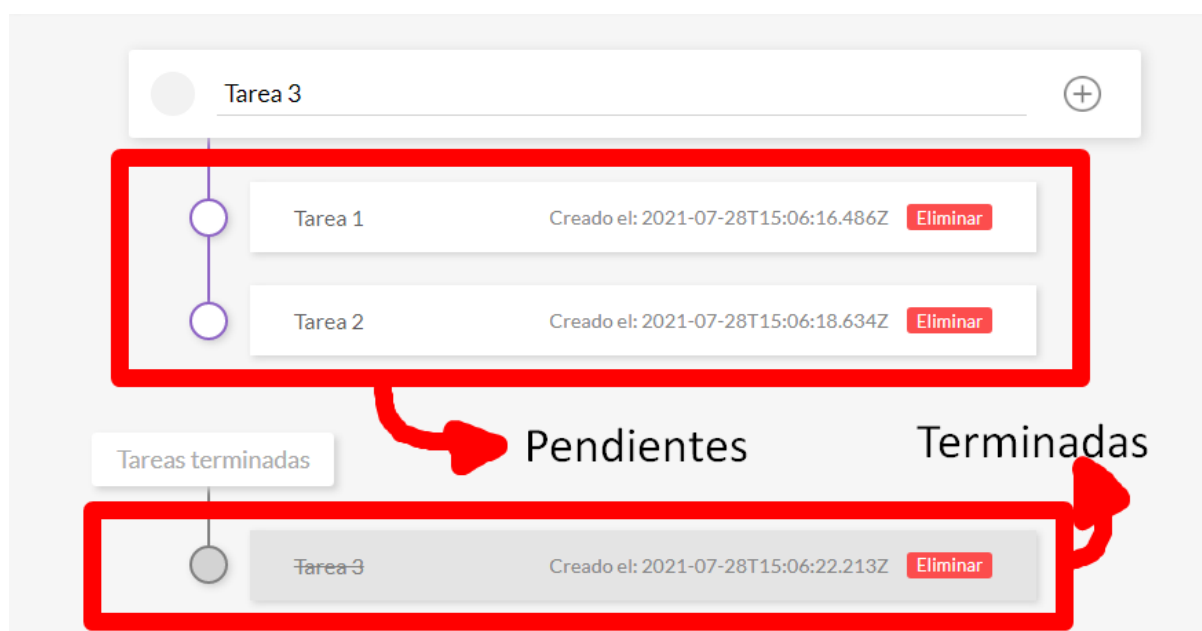


Informe proyecto ToDo

Puntos a analizar:

- Ver tareas pendientes.
- Ver tareas terminadas.
- Marcar una tarea como terminada.
- Crear nuevas tareas.
- Ingresar a nuestra cuenta (*login*).
- Ver la fecha de creación de una tarea.

Ver tareas pendientes y ver tareas terminadas



Funcionamiento en el código

Para ver las tareas, ya sean pendientes o terminadas, de un usuario tenemos que hacer un pedido de tipo GET a la API.

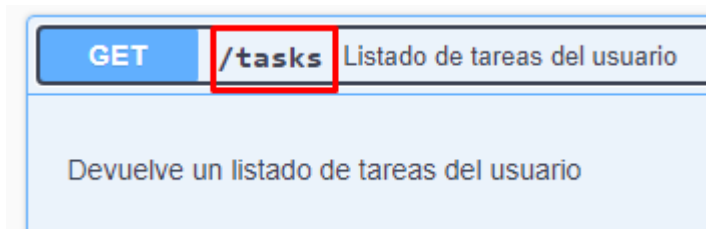
```

1  class RequestManager {
2      static baseUrl = 'https://ctd-todo-api.herokuapp.com/v1';
3
4      static getToken() {
5          return localStorage.getItem('token');
6      }
7
8      static get(endPoint) {
9          console.log('obtenido por request manager')
10         return fetch(RequestManager.baseUrl + endPoint, {
11             headers: {
12                 authorization: RequestManager.getToken()
13             }
14         }).then( data => {
15             return data.json();
16         })
17     }

```

Aquí podemos ver una clase llamada **RequestManager** la cual contiene un atributo y varios métodos estáticos (en la screenshot se alcanza a ver un método solo). En este instante nos interesa la función que se encuentra dentro del rectángulo rojo. Como podemos ver en la línea 8, la función es estática, recibe el nombre de “get” para simplificarnos lo que hace y recibe únicamente un parámetro el cual tiene un nombre acorde ya que la misma va a recibir un endPoint. Luego, en la línea 10 usamos la función **fetch** la cual recibe 2 parámetros:

- Por un lado, le pasamos la url base de la API la cual declaramos en la línea 2 de manera global ya que la seguiremos usando. Además, concatenamos a la URL el parámetro con el objetivo de poder decidir qué queremos hacer con la API. En este caso, al momento de usar dicha función tendríamos que pasarle en formato de string “/tasks” tal como nos indica la documentación de la API.



- Por otro lado, el fetch recibe como segundo parámetro las configuraciones o información necesaria para interactuar con la API. Al ser método GET, no hace falta aclararlo ya que es el que está por defecto. Lo que sí hace falta aclarar, es el token del usuario del cual queremos obtener las tareas. El mismo va como valor de “authorization” dentro de “headers”. En la línea 12 vemos cómo obtenemos el mismo mediante otra función propia de la clase RequestManager cuya declaración se encuentra desde la línea 4 a la 6, donde lo obtenemos mediante el método propio (getItem) de localStorage.

El fetch retorna un objeto de tipo respuesta y debido a su asincronismo es que debemos utilizar el método **.then** en las líneas 14 a 16 ya que la información que atrapa este es de tipo JSON y nosotros la necesitamos en formato JS. Entonces, la respuesta del fetch entra como parámetro del then, y luego en la línea 15 específicamente hacemos esta conversión utilizando el **.json()** propio de fetch.

Hasta aquí hemos visto solamente la arquitectura de la función que utilizaremos pero todavía no la hemos visto en funcionamiento.

En otro archivo, donde tenemos las funciones que interactúan directamente con la aplicación tenemos la función **getTareas()**...

```
68  ✓ function getTareas() {
69  ✓   |   RequestManager.get('/tasks').then(tareas => {
70  |   |       crearTareas(tareas);
71  |   |   })
72  |   }
```

La misma invoca a la función de la clase que previamente vimos, pasando como parámetro tal como mencionamos el endpoint correspondiente. Sin embargo, la función de RequestManager nos retorna el objeto parseado, nos falta decidir qué haremos con dicha información. Como vemos en las líneas 69 y 70, la respuesta que recibe el then pasa como parámetro de la función **crearTareas()**. La presentamos:

```
23  function crearTareas(tareas) {
24  |   document.querySelector('ul.tareas-pendientes').innerHTML = '';
25  |   document.querySelector('ul.tareas-terminadas').innerHTML = '';
26  |
27  |   tareas.forEach(tarea => {
28  |   |   renderizarTarea(tarea)
29  |   |   })
30  |   }
```

Esta función se encarga de renderizar las tareas mediante la función **renderizarTarea()** la cual explicaremos brevemente, pero antes, vale aclarar lo que sucede en las líneas 24 y 25; aquí se vacían los campos donde se encuentran las mismas, ya sean pendientes o terminadas, para evitar la duplicidad de información. Volviendo a las líneas 27 a 29, teniendo en cuenta que **tareas**, la información que ingresa como parámetro, es una lista de tareas, la recorremos con un **forEach** y mediante la función de la línea 28 renderizamos una por una.

```

32 function renderizarTarea(tarea) {
33   const template = `
34     <li class="tarea animar-entrada">
35       <div class="not-done" onclick='completarTarea(${tarea.id}, ${tarea.completed})'></div>
36       <div class="descripcion">
37         <p class="nombre">${tarea.description}</p>
38         <p class="timestamp">Creado el: ${tarea.createdAt}</p>
39         <button class="eliminar" onclick='eliminarTarea(${tarea.id})'>Eliminar</button>
40       </div>
41     </li>
42   `;
43
44   const contenedorTareas = document.querySelector('ul.tareas-pendientes');
45   const contenedorTareasCompletas = document.querySelector('ul.tareas-terminadas');
46   if (!tarea.completed) {
47     contenedorTareas.innerHTML += template;
48   } else {
49     contenedorTareasCompletas.innerHTML += template;
50   }
51 }

```

La función **renderizarTarea** recibe un parámetro. Dicho parámetro es un objeto que contiene las características de una tarea: un identificador único (id), si está completada o no con un booleano (completed), una descripción (description) y una fecha de creación (createdAt). Esta información la podemos ver en la documentación de la API:

Responses	
Code	Description
200	Operación exitosa
	Example Value Model
	<pre>[{ "id": 1, "description": "Aprender Javascript", "completed": false, "userId": 1, "createdAt": "2021-06-30T22:53:09.549Z" }]</pre>
401	Requiere Autorización
500	Error del servidor

Dentro de dicha función creamos una constante para utilizar un template para luego añadir en el html y en el mismo utilizamos template strings para poder insertar la información que trae el parámetro. En la imagen debajo este template va desde la línea 33 a la 42.

Luego en la línea 44 creamos una constante en la cual se almacena el **ul.tareas-pendientes** (elemento contenedor de las tareas a crear) mediante la propiedad `querySelector` propia de `document`.

Por último, vimos todo el mecanismo de cómo obtener las tareas pero nunca vimos el llamado a la misma, es decir, nunca hizo nada hasta ahora.

```
1  window.onload = () => {  
2    getTareas();  
3  
4    document.forms.agregarTarea.addEventListener( 'submit', event => {  
5      event.preventDefault();  
6      agregarTarea()  
7    });  
8  }
```

La instanciamos dentro del **window.onload** para que esta se ejecute una vez que haya cargado el documento así evitamos cualquier conflicto con el DOM (Document Object Model).

Marcar una tarea como terminada

Para esto tenemos una función denominada **completarTarea()**. Esta recibe 2 parámetros, el primero es el id de la tarea y el segundo es el estado en que se encuentra (completed), recordemos que este era un booleano.

```
73  function completarTarea(id, completed) {  
74    const body = {completed: !completed};  
75    RequestManager.put('/tasks/'+id,body)  
76      .then(tarea => {  
77        getTareas();  
78      }).catch(err => {  
79        console.log(err)  
80      })  
81  }
```

En la línea 74 declaramos una constante llamada `body` y le asignamos a la propiedad pasada por parámetro el estado contrario al que se encontraba. Es decir, al estar negado (**!completed**) si el parámetro entró por `true`, cambia a `false` y viceversa. Luego, invocamos al método **put** de `RequestManager` que brevemente

explicaremos pasándole por parámetros en primer lugar el endpoint, que se forma del string /tasks/ como escribimos concatenado al id que se pasa por parámetro a la función y por otro lado, el body que declaramos en la línea superior para realizar la modificación que queremos. En adición, utilizamos un `.then` para atrapar el objeto de tipo respuesta e instanciamos a `getTareas()` que vimos previamente lo que hace, con el objetivo de actualizar las tareas y visualizar el cambio que se hizo. Finalmente, usamos `.catch()` para atrapar cualquier error que pueda suceder y así poder visualizarlo por consola.

Analicemos qué tiene el método `put`...

```
32     static put(endpoint, body) {
33         return fetch(this.baseUrl + endpoint, {
34             method: 'PUT',
35             headers: {
36                 authorization: RequestManager.getToken(),
37                 "Content-Type": "application/json"
38             },
39             body: JSON.stringify(body)
40         }).then( data => {
41             return data.json();
42         })
43     }
```

Como habíamos mencionado, `put` espera que pasemos por parámetro un endpoint y un body que va a traer la información que se quiere modificar. Por su lado, `fetch`, recibe 2 parámetros nuevamente y la lógica se mantiene, es decir, primero se invoca a la variable `baseUrl` que está dentro de la clase, esto lo referenciamos con la palabra **this**. A esta URL le concatenamos el correspondiente endpoint que nos indica la documentación de la API y como segundo parámetro pasamos las configuraciones:

- En la línea 34 indicamos que el método a utilizar va a ser `PUT`, por lo tanto indicamos que queremos realizar una modificación.
- De la línea 35 a la 38 tipeamos la información necesaria que tiene que viajar en los headers, en este caso necesitamos el token nuevamente y en la línea 37 aclaramos que estaremos enviando un archivo de tipo `JSON`.
- Por último, en la línea 39 tenemos que pasarle el body que entra por parámetro y es la información a modificar. Como nosotros pasamos un objeto en formato `js`, utilizamos **`JSON.stringify()`** para convertir ese `js` a `JSON` y así, la API podrá entender nuestra información enviada.

Recordamos que el `.then` luego del `fetch` tiene como objetivo reconvertir el lenguaje, es decir, traducir de `JSON` a `js`.

Otra vez vimos cómo funciona pero no en qué momento se instancia...

```
31 function renderizarTarea(tarea) {
32   const template = `
33     <li class="tarea animar-entrada">
34       <div class="not-done" onclick='completarTarea(${tarea.id}, ${tarea.completed})'></div>
35       <div class="descripcion">
36         <p class="nombre">${tarea.description}</p>
37         <p class="timestamp">Creado el: ${tarea.createdAt}</p>
38         <button class="eliminar" onclick='eliminarTarea(${tarea.id})'>Eliminar</button>
39       </div>
40     </li>
41   `;
}
```

Volvemos a traer la función renderizar tarea y antes habíamos mencionado que utilizabamos un template string para poder utilizar variables dentro. Como se puede ver, en la línea 34 colocamos el atributo **onclick=""** que se va a renderizar en el HTML y nos va a indicar que cuando hagamos click en ese div se va a llamar a la función `completarTarea()` la cual recibe como parámetro el id de la tarea (`tarea.id`) y el estado en que se encuentra (`tarea.completed`).

Crear nuevas tareas

Una vez logueado, es decir, teniendo un token el cual nos permita estar en la página de "lista-tareas.html", podemos agregar una tarea completando el campo correspondiente(donde dice "Descripción de la tarea") y haciendo click en el siguiente ícono marcado a continuación para que la misma se renderize y se vea como pendiente.



Para esto, necesitamos entender qué hace el método `post` de la clase `RequestManager`.

```
19 static post(endPoint, body) {
20   return fetch(this.baseUrl + endPoint, {
21     method: 'POST',
22     headers: {
23       authorization: RequestManager.getToken(),
24       "Content-Type": "application/json"
25     },
26     body: JSON.stringify(body)
27   }).then( data => {
28     return data.json();
29   })
30 }
```

Su sintaxis y funcionamiento es igual al explicado anteriormente con la excepción de que en la línea 21 cambiamos el método a POST, indicando a la API que queremos enviar datos al servidor.

```
52  function agregarTarea() {
53      const descripcion = document.forms.agregarTarea.descripcionNuevaTarea.value;
54      const body = {
55          description: descripcion,
56          completed: false
57      }
58
59      RequestManager.post('/tasks', body)
60          .then( tarea => {
61              renderizarTarea(tarea);
62          }).catch( err => {
63              console.log(err);
64          })
65  }
```

La función **agregarTarea()** va a ser la encargada de justamente agregarnos la tarea. En la línea 53 capturamos el valor del campo donde antes habíamos escrito “Descripción de la tarea” y luego en la línea 54, declaramos el body con los datos necesarios requeridos por la API para realizar el POST, los cuales son la descripción que va a ser capturada por el campo y tendrá el valor que el usuario decida y el estado que lo inicializamos en false ya que el propósito de la app es que se creen tareas incompletas con el objetivo de realizarlas luego, por ende, deben inicializarse en false para renderizarse en la sección de tareas-pendientes. A partir de la línea 59, invocamos al post de la API y le pasamos el body previamente mencionado por parámetro. Si no hubo errores, el .then va a atrapar la respuesta del post y decirle que ese resultado lo renderize.

Ingresar a nuestra cuenta (login)

Aunque sea un tanto obvio, vale aclarar que para ingresar a una cuenta, debemos crearla primero. Analicemos el login...


```

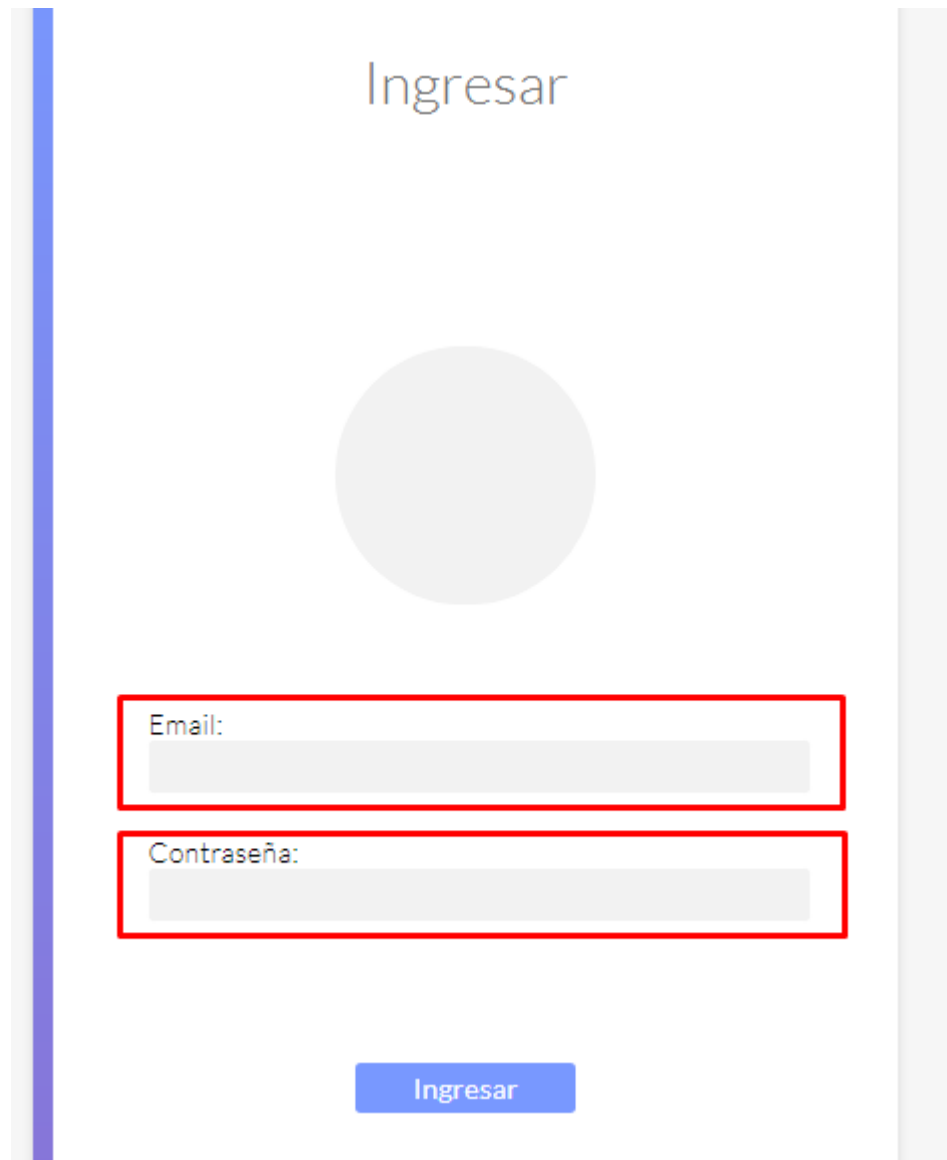
1  window.onload = () => {
2
3      const form = document.forms.formLogin;
4
5      form.addEventListener('submit', e => {
6          e.preventDefault();
7
8          const email = form.email.value;
9          const password = form.contrasenia.value;
10
11         console.log(form)
12         console.log(email)
13         console.log(password)
14
15         const url = 'https://ctd-todo-api.herokuapp.com/v1';
16         fetch(`${url}/users/login`, {
17             method: 'POST',
18             headers: {
19                 "Content-Type": "application/json"
20             },
21             body: JSON.stringify({
22                 email,
23                 password
24             })
25         }).then(datos => {
26             return datos.json();
27         }).then(datos => {
28             localStorage.setItem('token', datos.jwt);
29             location.href = './lista-tareas.html'
30         })
31     })
32 }
33
34 }

```

Primero que nada, encerramos todo dentro del window.onload para evitar conflictos con el DOM ya que desde js capturamos elementos del mismo y primero necesitamos que se terminen de cargar sino lo más probable es que nos arroje algún error.

En la línea 3 capturamos el formulario con el método **.forms** propio de document y en caso de que hubiera más de uno o en el futuro se quisiera agregar otro en el HTML, especificamos que queremos obtener el que tiene como **name** "formLogin". En la línea 5 le agregamos un evento submit al campo mencionado y pasamos por parámetro el evento (e) y así poder evitar que se envíe al hacer click con el método **.preventDefault()**.

En las líneas 8 y 9 capturamos los valores de los campos:



The image shows a login form with the title "Ingresar" at the top. Below the title is a large, light gray circle. Underneath the circle are two input fields. The first field is labeled "Email:" and the second is labeled "Contraseña:". Both input fields are highlighted with red rectangles. Below the input fields is a blue button with the text "Ingresar". The entire form is centered on a white background with a light gray border.

En la línea 15 declaramos la constante con la url base que necesitamos y a partir de la línea 16 hasta la 30 tenemos el fetch. En primer lugar, con template string pasamos la url declarada arriba y le agregamos manualmente el endpoint que nos indica la API. Como segundo parámetro del fetch, pasamos las configuraciones e información necesaria:

- Aclaramos que vamos a utilizar el método POST.
- Que estaremos enviando información en formato JSON.
- En el body pasamos los valores capturados previamente y estos los convertimos a formato JSON mediante el método correspondiente.

- Como siempre, el fetch nos retorna un objeto de tipo respuesta pero primero debemos parsearlo a js con el primer `.then` mediante `.json()`
- Seguidamente, realizamos lo que queremos con la información de respuesta. En este caso, la API nos devuelve el token que nos corresponde a nuestro usuario y así poder ingresar. Para que este funcione debemos agregarlo al `localStorage` como valor de la key `token`. En la línea 28, realizamos esto mediante el método `.setItem()` propio de `localStorage` y como primer parámetro pasamos la key a la que le queremos agregar el valor, y como segundo parámetro, accedemos al `jwt` dentro del objeto de respuesta del `fetch` quién nos trae el valor de nuestro token.
- Por último, luego de haber seteado nuestro token en el `localStorage` ya podemos ingresar a la página sin problemas. En la línea 29 esta tarea la llevará a cabo el atributo `href` de `location` al cual le asignamos el path hacia la página donde queremos acceder.

Ver la fecha de creación de una tarea

```
31 function renderizarTarea(tarea) {
32   const template = `
33     <li class="tarea animar-entrada">
34       <div class="not-done" onclick='completarTarea(${tarea.id}, ${tarea.completed})'></div>
35       <div class="descripcion">
36         <p class="nombre">${tarea.description}</p>
37         <p class="timestamp">Creado el: ${tarea.createdAt}</p>
38         <button class="eliminar" onclick='eliminarTarea(${tarea.id})'>Eliminar</button>
39       </div>
40     </li>
41   `;
42 }
```

En la función `renderizar tarea`, en la línea 37, como ventaja de los `template strings`, podemos acceder a la propiedad `createdAt` de `tarea` que es pasada por parámetro y esta nos devuelve su valor, que efectivamente es la fecha de creación de la tarea.