Práctica y Trabajos Tutelados de Programación Avanzada (2019-2020)

Memoria

Diego Varela Candal diego.vcandal@udc.es

Adrián Bueno Leiro adrian.bueno.leiro@udc.es

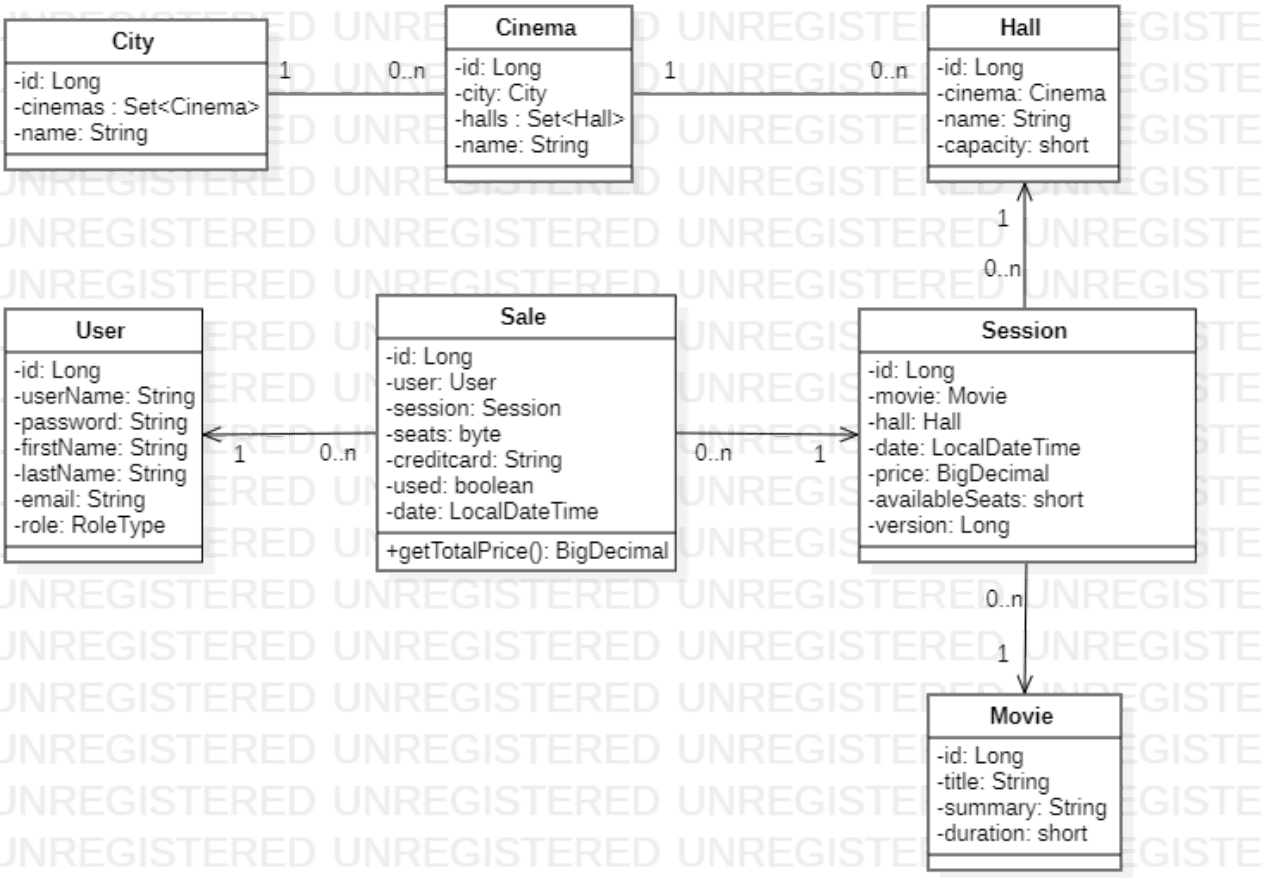
Anxo Cristobo Fabeiro anxo.cristobo@udc.es

1. Introducción

Para esta práctica en la iteración actual se implementan los dos trabajos tutelados, además de la parte opcional. Todos los diagramas se encuentran actualizados a esta última versión. Algunos detalles concretos se explicarán en los apartados correspondientes.

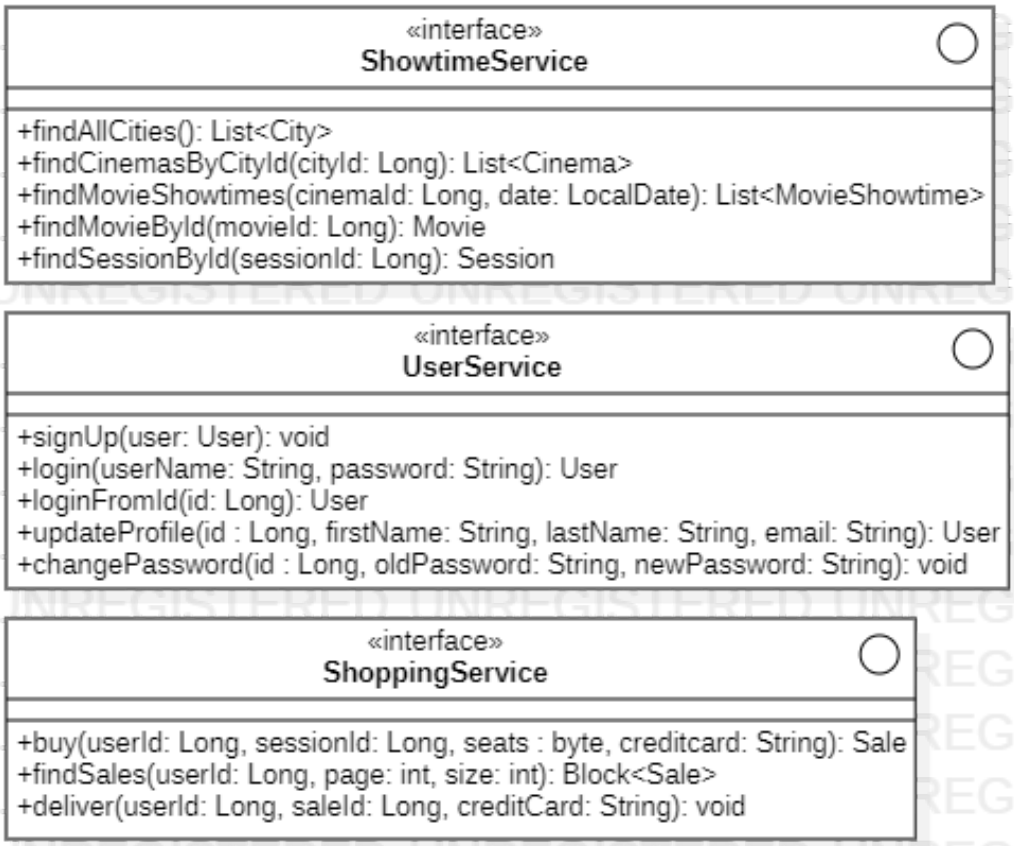
2. Backend

2.1. Capa de acceso a datos



Para este apartado se ha decidido definir varias relaciones como unidireccionales, en concreto en las que hemos considerado que pueden llegar a tener una gran cantidad de elementos en una parte. En cambio para las relaciones de City, Cinema y Hall se declaran bidireccionales, ya que no consideramos que vayan a tener esta cantidad de elementos.

2.2. Capa de lógica de negocio



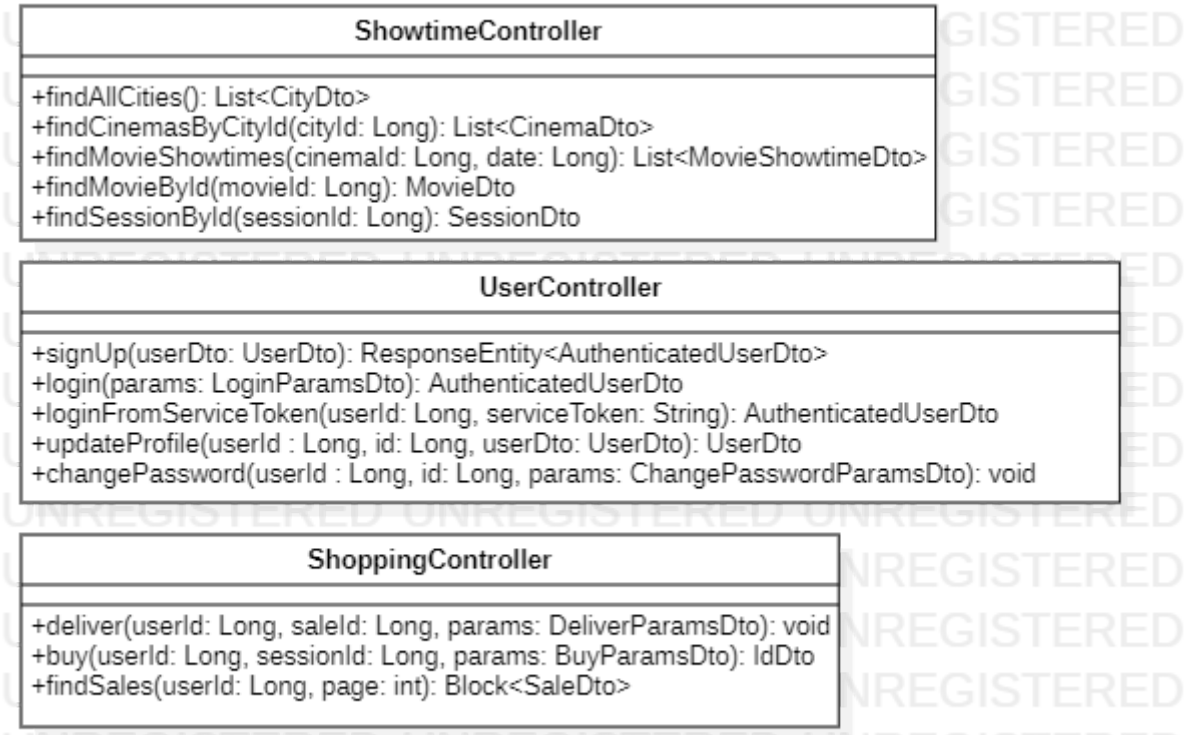
Se han definido 3 servicios locales:

- **ShowtimeService** → Búsqueda de cartelera, información de películas y sesiones, y recuperar las listas de cines y ciudades.
- **UserService** → Registro de usuarios.
- **ShoppingService** → Comprar entradas, visualización de compras y entregas de entradas.

Además para gestionar la cartelera se ha definido un DTO, *MovieShowtime*, con la película y su lista de sesiones disponibles para un día concreto.

Por último mencionar que el para el caso de uso 'Buy', en 'ShoppingService', podrían darse problemas surgidos del nivel de aislamiento usado. Por ejemplo si dos usuarios intentan comprar la última entrada "justo" en el mismo momento, una de las transacciones seguramente se vea afectada por "second lost update", y ambos acabasen con la última entrada.
Para evitar este tipo de problemas se aplica la técnica de **Optimistic Locking** sobre la entidad **'Session'**, añadiendo el atributo/propiedad 'version' a la entidad, y añadiendo la columna a la tabla.

2.3. Capa servicios REST



Definimos 3 controladores, cada uno asignado a su correspondiente servicio. Para cada uno se definen los siguientes recursos que recibirán las peticiones:

- **ShowtimeController**
  - **/showtimes/cities** → GET. Método *findAllCities*.
  - **/showtimes/cinemas** → GET. Método *findCinemasByCityId*. Entradas: cityId [param]
  - **/showtimes/movies/{id}** → GET. Método *findMovieById*. Entradas: id [path]
  - **/showtimes/sessions/{id}** → GET. Método *findSessionById*. Entradas: id [path]
  - **/showtimes/cinemas/{id}/movieShowtimes** → GET. Método *findMovieShowtimes*. Entradas: id [path], date [param]

- **UserController**
  - **/users/signUp** → POST. Método *signUp*. Entradas: UserDto [body]
  - **/users/login** → POST. Método *login*. Entradas: LoginParamsDto [body]

- **/users/loginFromServiceToken** → POST. Método *loginFromServiceToken*. Entradas: userId [jwt]
- **/users/{id}** → PUT. Método *updateProfile*. Entradas: userId [jwt], id [path], UserDto [body]
- **/users/{id}/changePassword** → POST. Método *changePassword*. Entradas: userId [jwt], id [path], ChangePasswordParamsDto [body]

- **ShoppingController**
  - **/shopping/sales/{saleId}/deliverTickets** → POST. Método *deliver*. Entradas: userId [jwt], saleId [path], DeliverParamsDto [body]
  - **/shopping/sessions/{sessionId}/buy** → POST. Método *buy*. Entradas: userId [jwt], sessionId [path], BuyParamsDto [body]
  - **/shopping/sales** → GET. Método *findSales*. Entradas: userId [jwt], page [param]

3. Frontend

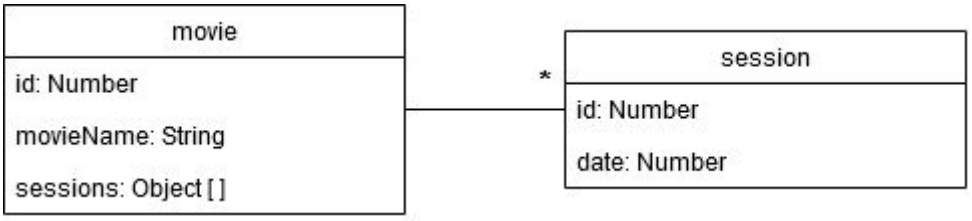
3.1. Módulos

A continuación se describen todos los módulos definidos, mostrando los correspondientes diagramas de estado.

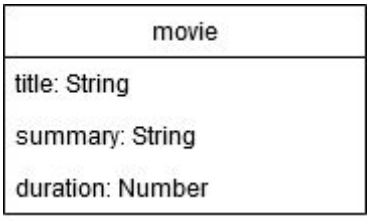
3.1.1. Showtimes

Este módulo se encarga de gestionar la búsqueda de cartelera, información de películas y sesiones, y recuperar las listas de cines y ciudades. A continuación se define el diagrama del estado general y los diagramas con el detalle de cada uno de sus componentes, mostrando sus atributos y su tipo de dato en JavaScript:

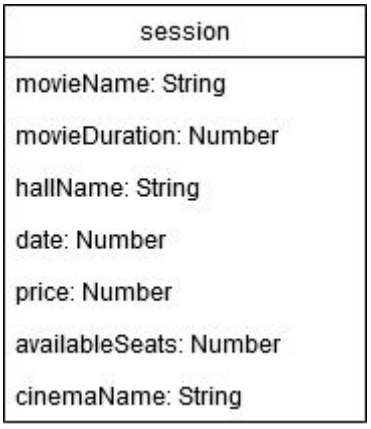
- **movies** → Se trata de un array de objetos que representan las películas con sus sesiones con las que se construirá la cartelera. El atributo sessions a su vez representa un array de objetos con estas sesiones, que también se muestra a continuación.



- **movie**→ Se trata objeto que representa los detalles de la película que se está visualizando en ese momento.

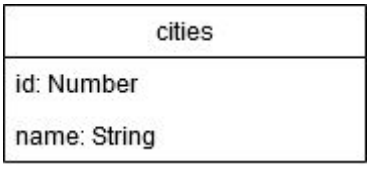


- **session** → Se trata objeto que representa los detalles de la sesión que se está visualizando en ese momento.

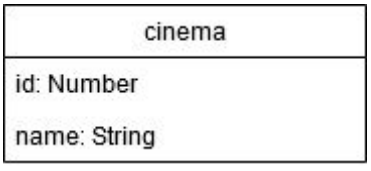


- **showtimesDate**→ La fecha seleccionada para ver la cartelera de ese dia.

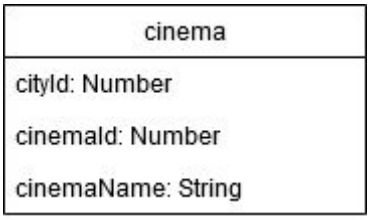
- **cities** → Es un array de objetos que representan las ciudades existentes en el sistema con sus datos.



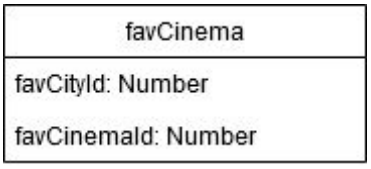
- **cinemas** → Array de objetos que representan las cines, de una ciudad determinada, existentes en el sistema con sus datos.



- **cinema** → Objeto que representa al cine actualmente seleccionado para ver su cartelera.

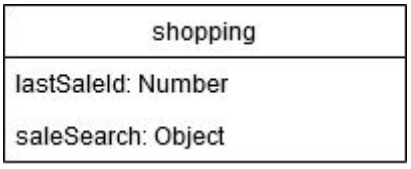


- **favCinema** → Objeto que representa el cine marcado como favorito, con sus datos correspondientes (se explica en detalle en el siguiente apartado).



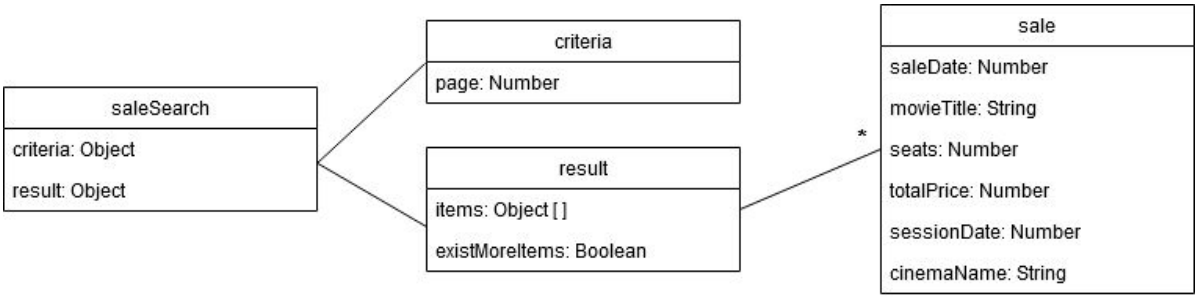
3.1.2. Shopping

Este módulo se encarga de gestionar la compra entradas, visualización de compras y entregas de entradas. A continuación se define el diagrama del estado general y los diagramas con el detalle de cada uno de sus componentes, mostrando sus atributos y su tipo de dato en JavaScript:



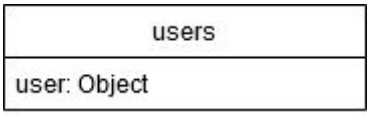
- **lastSaleId**→ El identificador de la última compra realizada.

- **saleSearch**→ Objeto que contiene el resultado de la búsqueda del histórico de compras del usuario. Donde las compras encontradas se guardan en 'items', en el objeto 'result'.

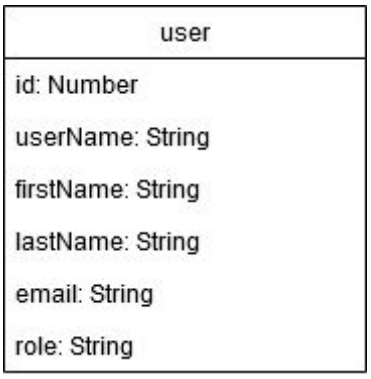


3.1.3. Users

Este módulo se encarga de gestionar las funcionalidades relacionadas con los usuarios como registro, login, cambio de contraseña y actualizar perfil. A continuación se define el diagrama del estado general y los diagramas con el detalle de cada uno de sus componentes, mostrando sus atributos y su tipo de dato en JavaScript:



- **user**→ El objeto que representa al usuario actual.



3.2. Otros detalles de la implementación

- Implementación de la **selección de cine favorito** en el almacenamiento local:

    - Para realizar este caso de uso creamos dos componentes nuevos, en este caso botones, para controlar la selección como favorito y el borrado de favorito. A la hora de guardar simplemente se guarda el identificador de cine y ciudad junto al nombre de la ciudad, para poder cargar toda la información necesaria al abrir la aplicación. El botón de borrado únicamente borra la key.

      Se define en el estado un nuevo atributo para tener la información del cine favorito, usandolo para seleccionar el cine y ciudad adecuados en el selector y para gestionar qué botón mostrar.

      La key empleada para almacenar los datos es *'PAProject_pa17_favCinema'*, la cual se buscará siempre que se acceda a la aplicación, y si se recupera algo, con los atributos necesarios se invoca a la acción definida que controla la búsqueda de los cines a partir de la ciudad donde se encuentra el cine favorito, seleccionarlo como cine favorito en el estado y buscar la cartelera asociada.

4. Trabajos tutelados

4.1. Backend: BatchSize

Para este trabajo nos hemos limitado a seguir las explicaciones proporcionadas aplicando la estrategia para la cartelera y el histórico de compras. De esta forma para la cartelera, se necesita la anotación para la entidad **Movie**, y en el histórico de compras, es necesaria para **Session**, **Movie**, **Hall** y **Cinema**, ya que se necesitan recuperar ciertos datos de estas entidades para los dos casos de uso. Esto se debe a lo siguiente:

- Cuando obtenemos la cartelera se obtiene una lista de sesiones para el cine y horas marcadas. Cuando creamos los objetos 'movieShowtime' (almacenan la película y las sesiones que se le encontraron) guardamos las sesiones agrupandolas por su id de película, hasta aquí no se produce ningún problema ya que este id se recupera con las sesiones.

  El problema viene cuando intentamos acceder al título de las películas para añadirlo a los datos del DTO correspondiente, ya que estaríamos realizando una consulta por cada película, incurriendo en el problema de las n+1 consultas.

- De forma similar para el histórico de compras, cuando añadamos los datos al DTO tendremos que acceder las sesiones (y a su película), y al hall (para acceder a su cine). Sería el mismo problema que antes, incluso peor, ya que estamos accediendo a 4 entidades distintas por cada Sale, y resultando en 4 consultas select, 4*n + 1 consultas.

En los dos casos se estaría accediendo a las entidades relacionadas, pudiendo resultar en un gran cantidad de consultas adicionales. Por lo que para estas entidades se emplea la anotación @BatchSize, intentando reducir el problema todo lo posible, en ambos casos:

- En el caso de la cartelera, cuando se acceda a los datos de una película, con la anotación se marca que al cargar el proxy de la película en la misma transacción se pueden cargar hasta otros 10 proxies más, evitando una cantidad de consultas excesivas, y reduciendo el problema a n+1 / 10 consultas.
- De la misma forma para el histórico de compras, cada vez que se acceda a las entidades relacionadas se cargaran hasta 10 proxies de cada una, reduciendo considerablemente el problema, al igual que antes.

Además decidimos dejar el mismo tamaño (10) del ejemplo. Aunque para la práctica actual posiblemente no lleguemos a necesitar ese tamaño, consideramos que ese valor o alguno cercano sería adecuado en un entorno más real.

4.2. Frontend: pruebas unitarias

Se implementan los tests en frontend para el caso de uso 'Compra de Tickets', implementarlas para el componente que muestra el formularios **(`modules/shopping/components/BuySessionTicketsForm.test.js`)**, la acción de compra correspondiente **(`modules/shopping/action.test.js`)**, y el comportamiento del reductor **(`modules/shopping/reducer.test.js`)** tras la compra.

Para poder implementar los test ha sido necesario realizar una serie de cambios al código actual.

- El primero trata de añadir las dependencias de las librerías usadas al archivo **`'frontend/package.json'`**, en este caso se añadirían con 'devDependencies:{ }', para ser ejecutadas en el modo desarrollo.

- Se añade el archivo **`'frontend/package.json'`** que contiene el setup global de los test implementados, en este caso importa una librería de Jest.

- A la accion 'buyCompleted' de '**`src/modules/shopping/actions.js'`** se le añade un export para poder invocarla desde fuera.

Como se explicó se realizan tests para los tres apartados mencionados. Como no se tiene información acerca de los errores del backend (ya que no se ejecuta para estas pruebas), por lo que todos los posibles casos se representan en uno. Para cada caso:

- **`BuySessionTicketsForm.test.js`** → Se prueba el componente del formulario. Se realizan dos test, uno para el caso que la compra se completa con éxito, y otro en el que se producen errores en el backend. También se define una función para renderizar el componente con un estado inicial, con una store donde su función dispatch se implementa con un mock.

    - En el caso de compra exitosa definimos la función mock que reemplaza a la acción original 'buy', siempre se devuelve la función onSuccess() recibida por parámetros, y además pudiendo 'espiar' las llamadas que se hagan sobre esa función.

      Renderizamos el componente y obtenemos los inputs y el botón de compra. A estos inputs se les cambia el valor actual por uno que representa un caso real. Y comprobamos que ese valor se cambia correctamente.

A continuación generamos el evento 'click' del botón, ejecutando la acción buy. Como esta función se limita a ejecutar la función onSuccess() únicamente se cambia de página, no se realizara ningun cambio de estado ya que la función dispatch de la store se reemplaza con un mock.

Por último se comprueban que los parámetros de la acción buy, con mock, son los correctos. Y que el navegador cambio correctamente de página.

- En el caso de compra no exitosa con errores de backend las explicaciones son practicamente identicas. Para la acción buy en este caso siempre se devuelve el error, por lo que comprobamos que esos errores se muestren correctamente y no se cambia de página.

- **`action.test.js`** → Se prueba la acción de compra. Como antes se realizan dos tests, en caso exitoso y en caso de errores, y se define una store con mock. Para cada caso:

  - En el caso de éxito, se mockea la función de compra en el backend por una que siempre devuelve onSuccess() pasándole como argumento el id de compra. Las funciones onSuccess() y onErrors() se mockean sin implementación, y se ejecuta la acción de buy con los datos necesarios y ambas funciones.

    Esta acción ejecuta la función mock del backend. Al backend le llega la función onSuccess definida en la acción, que cuando se le pasa un id por parámetro, se ejecuta dispatch(buyCompleted(id)) para cambiar el estado.

    Esta acción se ejecuta a través del mock de la store. Se comprueban que los parámetros que recibe la función en backend son correctos, y que la acción esperada (buyCompleted(id)) también es la correcta.

  - En el caso de errores en el backend, las explicación son idénticas. La función de compra mockeada en backend devuelve siempre onErrors(), y cuando se ejecute la acción de compra no generará ninguna otra acción, siempre se ejecuta onErrors(). Se comprueba que esto es así, y que la función onErrors() se ejecuta con el parámetro esperado.

- **`reducer.test.js`** → Solo se comprueba el comportamiento del reductor ante la acción de compra. Se produce un nuevo estado con la función reducer(), pasandole el estado inicial (en este caso vacío ya que no influye en el resultado), y la acción de compra. Por último se comprueba que el identificador de la última compra (en el estado) es el esperado.

5. Problemas conocidos

Para el momento actual de la práctica no hemos encontrado ningún error presente en el código.