Gamely

Instalación

Ha sido generado como estático con todas las rutas relativas a la url de donde se sirve. Por tanto, abriendo solo el HTML en local no funcionará. Para ver la aplicación hay varias opciones:

1. Acceder a la página donde he subido el estático: https://ivanortegaalba.github.io/gamely/
2. Hacer git clone del proyect, y desde la carpeta raiz del mismo ejecutar: `npm install` y posteriormente `npm start` y ya estaría accesible desde localhost:8000
3. Descargar una extensión para Chrome como Web Server 200 OK! que simplemente seleccionando el index.html cargará la página en local.

Datos sobre el desarrollo:

- Horas invertidas: 20h
- Prácticas usadas:
    - Maquetación parcial siguiendo BEM y ITCSS con el pre-processador SASS.

      A mitad pensé en mostar una comparativa diferente con styled-component.

      El SASS no está siendo usado, solo está ahí para mostrar que mi forma de trabajar con el.
    - TDD (Jest) y Snapshot testing

    - Linting para accesibilidad y buenas prácticas en React a través de:
        - eslint-plugin-import
        - eslint-plugin-jsx-a11y Muy util para evitar transmitir errores de accesibilidad al real DOM.
        - eslint-plugin-react
- Principales ideas de desarrollo:
    - Mostrar mis capacidades de arquitectura de software y buena praxis en Javascript
    - Mostrar la capacidad de construir semantic HTML a través de React
    - Mostrar la capacidad de abstraer los estilos de la lógica de aplicación
    - Proveer diferentes perspectivas de architectura para modularizar componentes
      (véase el uso de componentes comunes customizables pero estandarizados)
    - Presentar nuevos conceptos de Frontend como CSS-in-JS o TDD basado en Snapshot Testing
    - Promover el buen uso del unit testing a distintos niveles en React: Componente, Reducer, Selectors, Distpachers…
    - Demostrar que no siempre rápido significa malo: Un projecto sólido en 20 horas
- Principales 3 puntos débiles y 3 puntos fuertes:
    - Débiles:
        - Estilo poco atractivo (Tenía pensamiento de hacer un guiño a King al estilo Candy Crush, pero por falta

de tiempo ha quedado como un prototipo)
        - Componentes customizados sin ser abstraidos
        - Dejar la accesibilidad para el final, y no poder haberla implementado al completo
    - Fuertes:
        - Optimización de rendimiento a partir de buena arquitectura
        - Testing
        - Introducir un nuevo paradigma de maquetacion basado en componentes sin pasar por el uso de clases en CSS.
- Principales TODO:
    - Uso de Flow para el tipado en JS. Solo he hecho uso de los propTypes.
    - Testear el HTML estático con el W3C validator. No fue posible por falta de tiempo y la lucha constante con subir el estático gh-pages.
    - Mejorar la apariencia de la web. Se ha quedado en un simple boceto por poner énfasis en la arquitectura de JS.
    - Mejorar la usabilidad concatenando los focus y ajustando mejor el grid a la pantalla
    - Crear los styled-components como: Grid, Flex, Card, etc.
    - Añadir el plugin de styled-component para mostrar el nombre del componente y subcomponente siguiendo la convención BEM.
      Ahora, solo con el hash, se hace dificil el debug.

Introducción

Para desarrollar esta plataforma, he decidido usar:

- React (CRA para ahorrar tiempo en la configuraciónd de Webpack)
- Redux
- Jest (testing)
- styled-component (CSS-in-JS)
- SASS (ha sido sustituido en última instancia por styled-component)

Javascript

Uso de única fuente de verdad para la interacción con el store.

Es uso común en Redux o en React el declarar funciones durante el render o dentro del mismo connect
cuando estamos declarando un componente container.
He replanteado esta práctica para evitar el duplicar código,
malgastar memoria declarando más de una vez las funciones o eventos, o inclusos añadir infinitos listeners sin necesidad.

Para ello he manejado conceptos como:

- Dispatchers: Contendrán todas las funciones que hagan usos del dispatch para lanzar una acción
- Selectors: Manejarán la lógica de cada uno de los reducers (véase

los que están declarados dentro del propio archivo del reducer) o de la interacción entre los mismos (véase los declarados en el index.js que hace la combinación final.)

Esta división tiene ventajas como:

- Fácil unit testing
- Una sola fuente de verdad para lanzar acciones u obtener información del store
- Fácil refáctor de reducers y estructura del store
- Abstracción del store en los containers

Abstracción de la API a través del Middleware

Uno de los grandes problemas de Redux son los side-effects que producen las acciones en las distintas partes del store
y que muchas veces son dificiles de controlar.
Una librería bastante famosa para evitar esto es redux-saga que consiste básicamente
en funciones generadoras que escuchan determinado tipo de acciones para controlar estos side-effects.

No obstante, uno de los principales usos qeu se le da a esta librería es hacer peticiones a la API.
Con este approach no evitas el tener que conocer donde estas llamando, de que forma, qeu recibes, como inyectarlo en el store, etc.
La forma de declarar una petición es a través de una acción:

```
{
    [CALL_API] = {
        types: ['GAMES_REQUEST', 'GAMES_SUCCESS', 'GAMES_FAIL
URE'],
        endpoint: `${API_URL}/games/`,
        schema: new schema.Array(new schema.Entity('games'))
    }
}
```

Con esto, el middleware:

1. Lanzará la petición fetch, a la vez que lanzará una acción con el type `requestType`:
2. Una vez la petición se resuelve:
    - Si la petición ha sido satisfactoria, se lanzará una acción con el tipo `successType` y con la respuesta **Normalizada** en base al esquema enviado en la acción.
      Esto es muy importante para evitar tener deslocalizadas las entidades dentro del store. Para mas información ver la librería normalizr
    - Si la petición ha sido fallida, se lanzará una accion de tipo `failureType` y además se añadirá un campo error con la información del error.
      Con esta arquitectura, es muy facil controlar toda la data recibida a través de peticiones fetch, ya que simplemente tenemos que distribuirla por el store usando los reducers que traten dichos tipos de acción.

> Además, usando la normalización, siempre guardaremos en un solo sitio la entidad, y podrémos distribuir las ids de dicho elemento a nuestro antojo, evitando así inconsistencia de datos.

TODO:

- [ ] **Add extraParams to the call API actions to support POST, PUT and PATCH requests**
- [ ] **Testear que los errores son lanzados**
- [ ] **Evitar solapamiento de requests. Si una request de la api ha sido lanzada dos veces, cancelar la primera**

Abstracción del localStorage a traves del Middleware

Al igual que el middleware para la interacción con la API, el middleware de la localStorage nos permite guardar ciertas partes del store en la memoria local cuando determinadas acciones son despachadas.
El uso es muy simple, para este caso, cuando un juego es guardado o borrado, actualizamos el store guardado en la memoria local.

Semantic testing and TDD

Durante el desarrollo de la aplicación, cada una de las funciones ha sido

desarrollada a la par de su tests.
Los componentes de react, a su excepción, primeramente fueron maquetados, y después distribuidos y testeados.

Además, he procurado describir y utilizar semanticamente cada una de las tests suites para poder explicar la
funcionalidad del código a través de los mismos tests.

No obstante, debido al ajustado tiempo, no han sido lo suficientemente revisados.

CSS

Estandarización de estilos. Del SASS al CSS-in-JS

Son muchos los problemas con los que tenemos que lidiar desarrollando con CSS: sobreescritura, parametrización, funciones, tooling, etc. Gracias a SASS podemos solventar algunos de ellos.

Otros como el evitar la sobreescritura y el famoso important han de usarse herramientas como CSS Modules o PostCSS.

En esta prueba, he intentado ir más hayá, y he usado styled-component.
Sigue la misma filosofía que CSS Modules, pero assignando directamente el className al componente.
De esta manera, podemos asegurar que ese CSS no va a ser sobreescrito

por otro.

Otra de las grandes ventajas, es que podemos ejecutar funciones dentro del string template,
por lo que podemos manejar el css directamente desde JS, evitando así tener que aprender
lenguajes nuevos, o pre procesadores. Véase en la implementación:

- mixins para añadir margins
- la parametrización del CSS en base a props del componente
- la gran utilización de código
- la posibilidad de testear el CSS desde Jest
- tematización en base a JSON configurable

Semantic HTML5

En React, es habitual abusar de div y span. En esta aplicación he usado elemento HTML5 que ofrecen más semantica.
No obstante no había mucho contenido, por lo que la mayor parte no lo he podido aprovechar.

Accesibilidad y compatibilidad

En cuanto a la accesibilidad, la propia guía de React te explica los problemas mas comunes acerca de accesibilidad en React.
Por haberlo dejado a lo último, no he tenido tiempo apenas de preocuparme por ello, por tanto me gustaría comentar un poco en la entrevista.

No obstante he realizado:

- Buen manejo del focus y el uso de la app sin raton
- Uso de rems en vez de pixels. Esto evita el problema de dimensionado y descolocación de elementos. Véase este artículo
- Responsive design. La aplicación es perfectamente usable en todo tipo de dispositivos

UX/UI

Respecto al diseño:

- He optado por un layout tipo Netflix, Google App Store, etc.
- Básicamente dos secciones colapsables y escrolables horizontalmente.
- He añadido una barra de busqueda simultanea en ambos paneles para que la busqueda sea efectiva, y el cambio de focus con el teclado ea rápido.
- He usado una paleta de colores con 3 tonos cada uno de ellos, jugando con eso establezco la tonalidad de los elementos del componente para poder ser tematizado facilmente.
- Los margins y paddings son standard. Han sido establecidos tamaños siguiendo la nomenclatura de tallas.