Presentación

El enfoque que tenemos en la asignatura para abordar la explicación del desarrollo frontend es tener un hilo conductor que nos permita enlazar la obtención de datos, su procesamiento, el renderizado en el navegador y la interacción con el usuario.

El origen de datos de nuestra aplicación es una API abierta y gratuita, por motivos académicos, cuya temática determina ese hilo conductor. Este trimestre desarrollaremos una API de HearthStone:

HearthStone es un juego de cartas digitales basado en los personajes de World of Warcraft. Ambos juegos han sido desarrollados por la compañía norteamericana Blizzard y ambos son digitales (no hay un formato físico del juego).

Las mecánicas del juego y el conocimiento del mismo son irrelevantes para desarrollar con éxito las prácticas. Sin embargo, para los neófitos, decir que se trata de un juego de uno contra uno en la que cada uno de los jugadores tienen 20 puntos de vida y deben conseguir bajar a 0 al oponente. Cada uno tiene un mazo de cartas, barajado, por lo que el orden de llegada de las cartas es aleatorio. Al empezar la partida se toman X número de cartas y empieza el turno del primer jugador. Este debe usar los diferentes tipos de cartas y las sinergias entre ellas para acabar con la vida del oponente. Múltiples estrategias y combinaciones son posibles pero todas ellas poseen puntos fuertes y débiles en base de la pericia del oponente y la elección de su propio mazo.

La aplicación web que vamos a desarrollar representará:

- Una barra lateral izquierda con selectores de las propiedades por las que se pueden categorizar las cartas
- Un cuerpo central con la presentación de las cartas que cumplen los selectores de la barra lateral izquierda
- Una barra lateral derecha con:
    - En la parte superior el mazo preparado
    - En la parte inferior los detalles de la carta del mazo sobre la que se hace hover

Este es el aspecto que tendrá (los ficheros *html* y *css* se proporcionarán como parte del enunciado de la PEC3):

Partiendo del trabajo realizado en la PEC2 de adquisición y tratamiento de datos nos queda por implementar:

- Representación en el DOM del modelo de datos
- Interacción con el usuario

Formato de entrega

A través del campus deberéis entregar un único fichero zip con el contenido de los archivos de desarrollo.

Nota: En esta PEC no es necesario hacer testing aunque si se hace será valorado positivamente. En cualquier caso, y en contra de las buenas prácticas de desarrollo, os recomendamos centraros en resolver el ejercicio y posponer el testing como última tarea, en caso de que queráis llevarlo a cabo.

Consultas

En caso de que tengáis que consultar algo mediante el foro o por correo electrónico, seguid estos pasos: (1) copiad vuestro código a una de las plataformas online que os indicamos abajo, y (2) enviad el link del código del ejercicio. Estos pasos os evitarán problemas con el correo de la **UOC**, que elimina los ficheros *js* para evitar inyectar código malicioso y la tediosa mala indentación al copiar/pegar el código en el correo electrónico. Usad cualquier de estos dos enlaces y la comunicación e interacción será mucho más sencilla:

- Codepen (para sencillos snippets de código)
- CodeSandBox (para ejercicios más complejos)

En caso de publicar algún código en el foro, éste debe estar relacionado con consultas genéricas y no directamente soluciones a los ejercicios. Hacer accesible soluciones de ejercicios a otros compañeros, aunque pueda no tener una intención directa, se considerará copia y se penalizará académicamente a nivel de asignatura.

Puntuación

A modo de orientación sobre la puntuación:

- Un ejercicio que no funcione correctamente, no renderice los datos en el navegador y no tenga implementadas las interacciones tendrá una nota máxima de 5 y y será su optimización y corrección lo que determinará la puntuación entre 0 y 5.
- Un ejercicio que funcione correctamente, renderice los datos en el navegador y tenga implementadas las interacciones tendrá una nota mínima de 5 y será su optimización y corrección en las formas lo que determinará la puntuación entre 5 y 8,5.

- Un ejercicio que cumpla lo anterior y además pueda gestionar correctamente la presentación de datos y las combinaciones de filtros sin tener que almacenar inicialmente todas las cartas tendrá una nota superior al 8,5 y de 10 como máximo.
- Se valorará la legibilidad y sencillez del código sobre la sofisticación (mirar apartado **Cuando aplican varios filtros**)

Propiedad intelectual y plagio

La Normativa académica de la UOC dispone que el proceso de evaluación se cimenta en el trabajo personal del estudiante y presupone la autenticidad de la autoría y la originalidad de los ejercicios realizados.

La ausencia de originalidad en la autoría o el mal uso de las condiciones en las que se realiza la evaluación de la asignatura es una infracción que puede tener consecuencias académicas graves.

El estudiante será calificado con un suspenso (D/0) si se detecta falta de originalidad en la autoría de alguna prueba de evaluación contínua (PEC) o final (PEF), sea porque haya utilizado material o dispositivos no autorizados, sea porque ha copiado textualmente de internet, o ha copiado apuntes, de PEC, de materiales, manuales o artículos (sin la cita correspondiente) o de otro estudiante, o por cualquier otra conducta irregular.

Cómo arrancar el desarrollo

En este momento tenemos dos opciones:

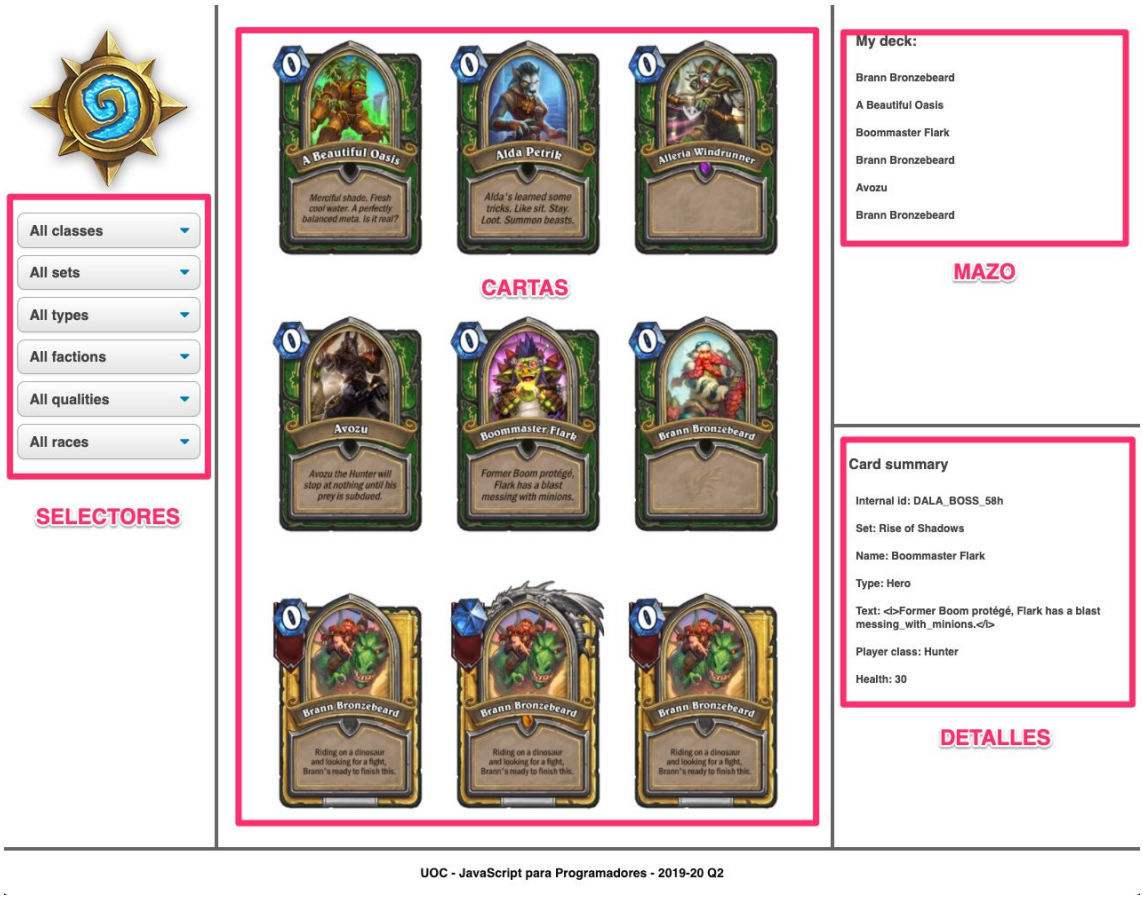
- Partir del ejercicio presentado de la PEC2 en el Campus
- Partir de la solución propuesta

Valorar cual de las dos opciones es más conveniente queda a discreción del alumno. Sin embargo cabe decir que aunque hay muchas aproximaciones válidas al ejercicio aquí se va a pautar el desarrollo esperado de la PEC3 partiendo de la base de la solución propuesta:

- Crear la carpeta **/pec3**
- Copiar todo el desarrollo realizado en la **/pec2** a la **/pec3** o bien copiar el contenido con la solución proporcionada

Qué funcionalidades implementar

Sin ánimo de encorsetar demasiado el desarrollo os proponemos la siguiente aproximación:



- El primer paso es el más sencillo, debemos crear los restantes métodos de obtención de datos en base al tipo de selector. Ya tenemos: getCardsByClass, los demás son prácticamente iguales, nos faltan:
    - getCardsBySet
    - getCardsByType
    - getCardsByFaction
    - getCardsByQuality
    - getCardsByRace
- Asegurarse de que las peticiones ya realizadas no se repiten y los datos primero se buscan localmente (DeckBuilderSingleton.cards)

- Construir una función que renderice los selectores en base a los datos obtenidos del endpoint de info
- Construir una función que renderice las cartas (básicamente la imagen)
- Añadir una propiedad a Deckbuilder para almacenar las cartas que vaya añadiendo el usuario a su mazo
- Adjuntar los eventListeners adecuados a:
    - Los selectores: para lanzar más consultas de cartas o realizar el filtrado
    - Las cartas: para poder añadirlas al mazo del usuario
    - La carta seleccionada del mazo: para ver sus detalles

JavaScript y el DOM

La interacción entre JavaScript y el DOM se realiza en dos direcciones:

- De JS a DOM: mediante renderizado de datos
- Del DOM a JS: mediante eventos de interacción

Renderizado

En nuestros ficheros de JavaScript tenemos el modelo de datos almacenado localmente pero para que éste tenga valor para el usuario debe quedar plasmado en el navegador. A este proceso lo llamamos renderizado de datos. Lo bueno de haber decidido un modelo de datos relacionado con la lógica de negocio de la aplicación es que a la hora de renderizar ya tenemos modelado cada uno de los elementos que vamos a representar.

Para construir elementos mediante JavaScript primero necesitamos saber su estructura HTML asociada, por ejemplo, imaginemos uno de los selectores:

```
<select name="classes" class="select-css">
    <option value="Death Knight">Death Knight</option>
    <option value="Druid">Druid</option>
    <option value="Hunter">Hunter</option>
    <option value="Mage">Mage</option>
    <option value="Paladin">Paladin</option>
    <option value="Priest">Priest</option>
    <option value="Rogue">Rogue</option>
    <option value="Shaman">Shaman</option>
    <option value="Warlock">Warlock</option>
    <option value="Warrior">Warrior</option>
    <option value="Dream">Dream</option>
```

```
    <option value="Neutral">Neutral</option>
    <option value="Demon Hunter">Demon Hunter</option>
</select>
```

En este ejemplo vemos que la estructura es fácilmente convertible a un bucle y solo debemos cambiar el contenido de cada uno de las etiquetas option así como su atributo value.

Podríamos construir algo de ese estilo de este modo:

```
function createSelector(options) {
  const selectEl = document.createElement('select');
  selectEl.name = 'classes';
  selectEl.className = 'select-css';

 options.forEach(option => {
    const optionEl = document.createElement('option');
    const optionText = document.createTextNode(option);

    optionEl.value = option;
    optionEl.appendChild(optionText);
    selectEl.appendChild(optionEl);
  });

  return selectEl;
}
```

Para que esto funcione debemos pasarle como options cada una de las categorías recibidas del endpoint info. Pero con esto solo tenemos localmente creado un selector, para poder insertarlo en el DOM debemos:

1. Seleccionar un nodo destino como padre de nuestro elemento generado
2. Insertar nuestro fragmento o elemento en el nodo obtenido

Para obtener un nodo de nuestro fichero HTML debemos realizar una consulta al objeto document que representa el árbol de elementos de nuestra página:

```
const sidebarSelectors =
document.querySelector('#hearthStone_sidebarSelectors');
```

Nota: en el fichero proporcionado ese nodo ya existe y es el que se espera que se utilice para renderizar los selectores.

Para insertar el elemento que hemos generado en JavaScript debemos hacer:

```
const classes = DeckBuilderSingleton.info.classes;
sidebarSelectors.appendChild(createSelector(classes));
```

Llegados a este punto deberiais:

- Realizar un bucle para pasar por todos los selectores
- Idealmente renderizarlos una única vez (no una vez por selector sino todos juntos)
- Encontrar el modo de añadir una opción por defecto (*All, Todos, Default*) que implique eliminar el filtrado por esa categoría

Teniendo este listo se debería realizar el mismo proceso con las cartas, la carta seleccionada (que aún no ha seleccionado el usuario) y los detalles de la misma en cada una de las partes de la página web.

Nota: es recomendable acabar el tema de renderizado antes de pasar al de interacción. Podéis usar opciones de los selectores fijos (class = hunter), ids de cartas concretos y todo lo que sea necesario (como en la PEC2) para obtener datos con los que trabajar y una vez todo esto funcione pasar al siguiente punto.

Interacción

Una vez tenemos todos los elementos representados en pantalla o si somos capaces de presentarlos es cuando vamos a poder empezar a realizar eventos para mostrar unos datos u otros a modo de respuesta a las peticiones del usuario.

Debemos pensar qué elemento recibirá la interacción y valorar si le añadimos directamente un evento o si lo hacemos a un elemento superior. Si pensamos en el ejemplo de los selectores tenemos dos opciones:

- Añadir un evento a cada selector (6 en total)
- Añadir un evento al padre que los contiene y usar delegación de eventos

Por temas de escalabilidad de aplicaciones es más óptimo la segunda opción. Si de repente aparece una nueva categoría no debemos hacer nada más que asegurarnos de que el evento ya gestiona ese caso correctamente que es mucho menos costoso que añadir un nuevo evento

por cada nuevo selector. Además, en nuestra aplicación la plantilla tiene una serie de nodos estáticos (antes de recibir datos) ya existentes, los nodos padres que albergarán nuestro contenido renderizado, lo que facilita buscar donde añadir los eventListeners ya que eso evita tener que hacerlo de forma dinámica en JavaScript.

Sin embargo esta decisión implica añadir una lógica de gestión para saber quien ha emitido el evento, es decir, con qué elemento ha interactuado el usuario.

Cada interacción del usuario dispara un evento que recorre el árbol html desde el origen del evento hasta la etiqueta html. Si tenemos un event listener en algún punto del camino podremos interceptarlo.

De nuevo, imaginemos el caso de los selectores, en este caso primero necesitamos recuperar del DOM el elemento al que le vamos a adjuntar el event listener:

```
const sidebarSelectors =
document.querySelector('#hearthStone_sidebarSelectors');
```

Y le añadimos un escuchador para el tipo “change” y un callback a ejecutar en ese caso:

```
sidebarSelectors.addEventListener('change', getCardsBySelector);
```

Veamos qué aspecto tiene el callback:

```
function getCardsBySelector(event) {
  const { value } = event.target;

  console.log("Value: ", value");
}
```

Como veis al callback le llega el objeto event que representa el evento que se ha lanzado en el DOM. Si el event listener se está ejecutando es porque es del tipo “change”.

Una vez dentro de la función buscamos “event.target” que representa el elemento original que desencadenó el evento dentro del DOM, en el caso de los selectores, puede ser cada una de las opciones sobre las que se puede hacer click en el desplegable. Si recordáis el código HTML de las opciones:

```
<option value="Hunter">Hunter</option>
```

El value es el atributo que hemos dado a cada uno de los options y que ahora podemos recuperar.

En este caso solo nos falta saber qué selector contiene “Hunter” como valor de uno de sus options. Esto puede averiguarse inspeccionando el objeto event o event.target más profundamente. Con esos datos ya podemos lanzar la petición de recuperación de datos de nuestro modelo ya bien sea en local si ya existen o contra la API si no se han solicitado previamente.

Debemos realizar el mismo procedimiento y reflexión con las cartas renderizadas para poder añadirlas al mazo del usuario y así mismo sobre las cartas de dicho mazo para mostrar sus detalles.

Hay ocasiones en las que no resulta tan sencillo como usar el value en un option por ello existen los data atributos que son atributos genéricos que se pueden añadir a cualquier elemento html:

```
<article
  id="electriccars"
  data-columns="3"
  data-index-number="12314"
  data-parent="cars">
...
</article>
```

```
var article = document.getElementById('electriccars');

article.dataset.columns // "3"
article.dataset.indexNumber // "12314"
article.dataset.parent // "cars"
```

Fuente: https://developer.mozilla.org/es/docs/Learn/HTML/como/Usando_atributos_de_datos

Estados de carga

Cada vez que el usuario realice una petición y los datos no existan localmente realizaremos una petición a la API. Mientras los datos se solicitan debemos darle al usuario algún tipo de respuesta y señal de que la aplicación se encuentra en movimiento y está realizando su operación. Para ello debéis preparar estados de carga (en el lugar central, donde se renderizan las cartas) y presentar un spinner o algún otro elemento que dé esa información visual al usuario.

Hay miles de snippets de código para representar spinners. Os dejo uno pero usad el que creáis más conveniente:

https://projects.lukehaas.me/css-loaders/

Para controlar el renderizado de contenido real o el spinner debéis tener una variable que represente condicionalmente uno u otro en base al estado de la aplicación. La instancia singleton de DeckBuilder puede ser un buen lugar para tenerla a modo de propiedad.

Cuando aplican varios filtros

La mayor complicación de esta práctica radica en la gestión correcta de los filtros en cada momento y lanzar las peticiones adecuadas sin solicitar los datos a la API si ya se han pedido antes.

Inicialmente, ningún filtro está seleccionado. Mi propuesta es que elijáis una de estas dos opciones:

- Cargar la primera clase por defecto: lanzar esa petición, mostrar las cartas y dejar preseleccionada esa clase en el selector de clase
- No lanzar ninguna petición y mostrar un mensaje en el cuerpo central indicando al usuario que realice su selección

En cualquier de los dos casos existirá una primera petición, supongamos por clase “Hunter”. Si sobre estas cartas se aplican más filtros de selección como faction “Horde”, deberemos filtrar las cartas actuales que mostramos para representar sólo las que cumplan esa condición adicional. Si se aplicara un nuevo filtro debería pasar el set previo esa nueva condición y así con todas.

Si el usuario deselecciona todos los selectores en el mismo orden que los ha seleccionado no hay ningún problema pero qué sucede si cuando llega al set inicial con clase “Hunter” y faction “Horde” deselecciona clase a “All” y se queda solo con faction “Horde”? El grupo base de cartas ya no nos sirve, por lo que habrá que lanzar una nueva petición para faction “Horde” y presentar todas las cartas de Horde porque hasta ahora solo teníamos las que cumplian clase “Hunter” y faction “Horde” y nos faltarían todas las del resto de clases.

Este es el caso sencillo de los difíciles, pero que pasa con todas las opciones de más de dos filtros?

Esta parte de la práctica es la que requiere una mayor reflexión a nivel de adquisición y representación de datos para cubrir todas las casuísticas sin repetir llamadas a la API pero a la vez representar siempre los datos de forma consistente.

Puesto que este punto se le puede atascar a mucha gente os propongo que inicialmente realicéis las llamadas para obtener las cartas de todas las clases. Eso os permitirá tener todas las cartas en local y filtrar sin tener que valorar qué sets de cartas están o no. Los que quieran aspirar a la máxima nota deberán resolver de forma exitosa esta última complejidad.