

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

A falta de algunos ajuste finales, este será el aspecto que tendrá (los ficheros *html* y *css* se proporcionarán como parte del enunciado de la PEC3):



Esto implica a grandes rasgos dos áreas de trabajo principales en la PEC2 sobre los datos como ya hemos comentado:

- adquisición: llamadas asíncronas a la API para recuperar los datos
- tratamiento: filtrado, ajuste y conversión de los datos recibidos

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 o no presente los datos solicitados a la API por consola 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 y presente los datos solicitados a la API por consola tendrá una nota mínima de 5 y será su optimización y corrección en las formas lo que determinará la puntuación entre 5 y 10.
- Se valorará la legibilidad y sencillez del código sobre la sofisticación.

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 continua (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.

Modificaciones sobre el entorno de trabajo

Cualquiera de los métodos de trabajo asíncrono es aceptado para resolver la práctica pero se recomienda encarecidamente usar *async / await* puesto que es la API de JavaScript más nueva y la que nos evita el *callback hell* de las promesas. Además, *async / await* permiten una legibilidad del código superior al convertir lo que es asíncrono en esencia en síncrono o secuencial a nivel de código.

Para evitar problemas con *eslint* y *babel* debemos asegurar que estas líneas existan en nuestros ficheros de configuración.

package.json

```
"browserslist": [  
  "last 1 Chrome versions"  
]
```

.eslintrc.json

```
{  
  "extends": ["eslint:recommended", "prettier"],  
  "parserOptions": {  
    "ecmaVersion": 2018,  
  },  
}
```

```
    "sourceType": "module"
  },
  "env": {
    "es6": true,
    "browser": true,
    "node": true,
    "jest": true
  }
}
```

Registro en la API

La gran mayoría de APIs públicas requieren el registro de los usuarios para poder empezar a utilizarla. Esto se debe a que las APIs ofrecen datos de forma gratuita sobre infraestructura (servidores) que no son gratuitos para los propietarios de las APIs.

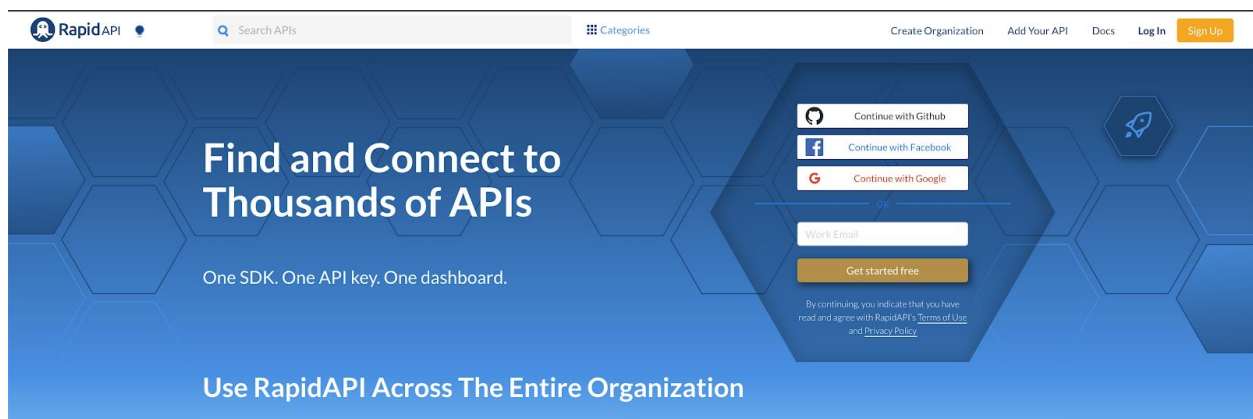
Con el registro de los usuarios se consigue el control de acceso mediante tokens identificativos, limitar el uso de la misma en base a peticiones y, sobretodo, permitir un servicio premium de pago, pruebas el servicio y si sirve a tus propósitos comerciales pagas por ella. En nuestro caso la opción gratuita es más que suficiente.

Aquí encontraréis los enlaces con los que trabajaremos:

- Página oficial de la API: <https://hearthstoneapi.com/>
- Página de la API para probar endpoints: <https://rapidapi.com/omgvamp/api/hearthstone>
- Repositorio de imágenes de cartas: <https://hearthstonejson.com/docs/images.html>

Nota: los enlaces de imágenes que vienen de la API de datos no siempre funcionan correctamente. Aprovecharemos ese hecho para incluir el trabajo con una segunda fuente de datos aunque sea de forma testimonial.

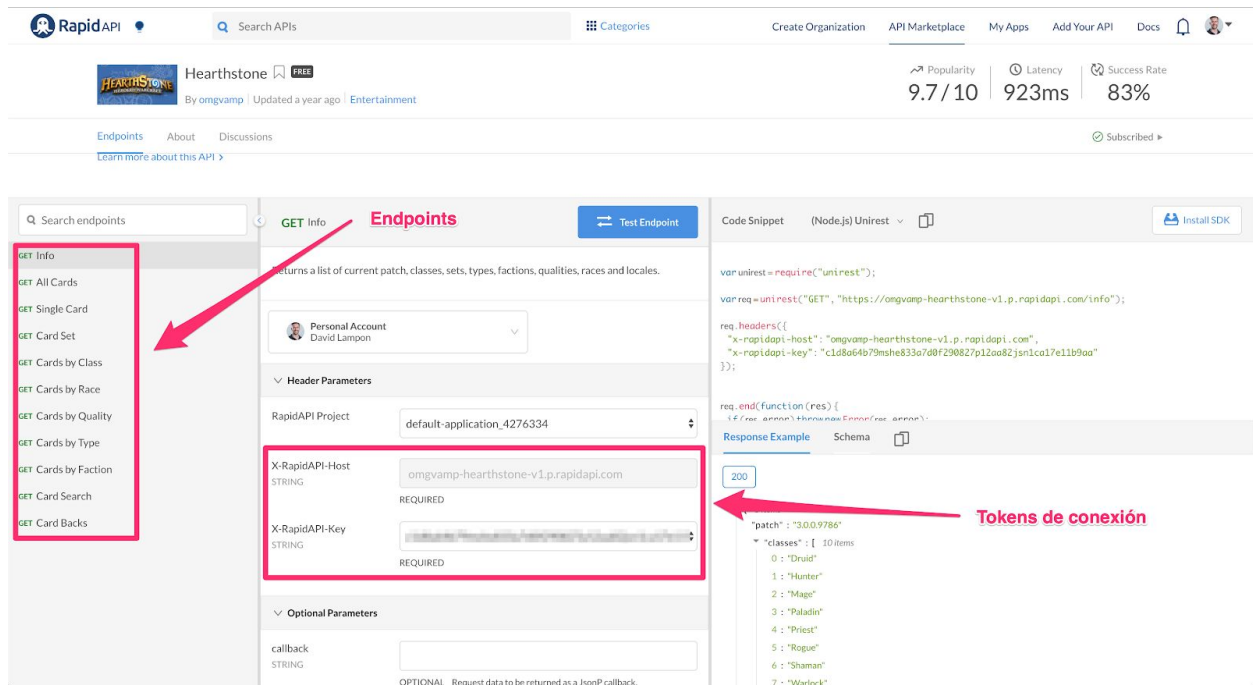
Debéis registraros para poder empezar a usar la API desde: <https://rapidapi.com/>



Nota: Personalmente, nos gusta utilizar nuestros perfiles de GitHub para registrarnos en todos los portales y plataformas relacionadas con temas de desarrollo.

Con esta información ya podremos consumir datos de la API y hacer llamadas a sus diferentes endpoints.

Ahora podéis acceder a la página de la API de HearthStone y consultar endpoints. Nos interesa especialmente obtener los tokens para usarlos como parámetros de configuración en nuestro código.



Consulta de endpoints

Los endpoints de una API son cada una de las direcciones o rutas de las que podemos obtener datos. Conceptualmente lo que haremos es acceder desde nuestro código a una dirección web mediante la API *fetch* de JavaScript y recuperaremos los datos de cada una de ellas.

Esto es bastante automático una vez hecho una vez pero la primera vez puede ser complicado conseguirlo. Para facilitar este paso, proporcionamos el aspecto que debe tener el código que accede a un endpoint. Os recomendamos copiar este código en su propio módulo **api.js** para empezar a trabajar con él:

```
const headers = new Headers();
headers.append('x-rapidapi-host', 'omgvamp-hearthstone-v1.p.rapidapi.com');
```

```
headers.append('x-rapidapi-key', 'vuestro_token');

async function getEndpoint(url) {
  try {
    const response = await fetch(url, { method: 'GET', headers });
    const apiData = await response.json();

    return apiData;
  } catch (err) {
    console.log('fetch failed', err);
  }
}
```

Donde *url* es cada uno de los endpoints de la API. Podéis revisar el listado en este enlace como vimos en la imagen más arriba:

- <https://rapidapi.com/omgvamp/api/hearthstone>

Cómo arrancar el desarrollo

Todos aquellos que tengáis claro cómo plantear la aplicación, adelante. Hay muchas maneras y todas ellas válidas. Para los que tengáis dudas, nuestra propuesta es:

- Crear la carpeta **/pec2**
- Crear un punto de entrada **/pec2/index.js**
- Dentro de **/pec2/index.js** crear una función llamada **init**:

```
export default function init() {
  console.log("Saludos desde la PEC2");
}
```

- Importar en **index.js** en la raíz del proyecto:

```
import init from './pec2';

console.log('Bienvenido a JS para programadores');
```



```
init();
```

Ahora ya podemos trabajar en nuestro punto de entrada de la PEC2 y usar *init* como la función que inicializará y realizará todas las peticiones llamando a otras funciones.

Cómo enfocar el desarrollo

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

- Crear clases que representen cada uno de los elementos de nuestra lógica de negocio. Antes de empezar a programar estaría bien tener clara las estructuras necesarias para modelar la aplicación. Propuesta:
 - *Card*: para cada una de las cartas
 - *DeckBuilder*: para el mazo creado, las cartas y consultas realizadas.
- Las clases se definirán cada una en su propio fichero dentro de la carpeta **pec2/Classes** con ficheros cuyo nombre empezara en mayúscula (p.e. *Card.js*)
- Para mantener nuestro fichero principal lo más limpio posible moveremos toda la lógica a funciones auxiliares en **pec2/utils** (por ejemplo)

Qué funcionalidades implementar

La funcionalidad que queremos implementar va muy ligada a la representación visual de nuestra aplicación:



Hay dos consultas para que esta UI se cargue correctamente:

1. Consulta de información general del juego: de aquí podemos obtener todos los tipos de las cartas (entre otros datos). Podéis obtener esta información desde este endpoint:
<https://omgvamp-hearthstone-v1.p.rapidapi.com/info>
2. Consulta de cartas por clase: para obtener las cartas por clase (p.e. *Hunter*) debemos usar el endpoint: <https://omgvamp-hearthstone-v1.p.rapidapi.com/cards/classes/Hunter>

Será necesario consultar diferentes endpoints para diferentes tipos de búsqueda así como añadir el parámetro. En la PEC2 debemos ser nosotros dentro del código los que definamos qué clase se va a buscar cuando implementemos esta funcionalidad. Será en la PEC3 cuando añadamos dinamismo a la aplicación permitiendo que sea el usuario el que seleccione la clase.

Hay una tercera que deberemos hacer para mostrar la carta: al recibir los datos de la carta uno de sus propiedades es `cardId` (p.e. `ICCA08_022`) y para poder construir la ruta de su imagen deberemos recuperar nuestra otra fuente de datos:

<https://hearthstonejson.com/docs/images.html>

Y construir rutas de imágenes usando el `cardID` del tipo:

https://art.hearthstonejson.com/v1/render/latest/enUS/256x/ICCA08_022.png

Donde debéis incluir al `cardId` el prefijo de la url y la extensión `png`.

Cómo presentar los datos

Para valorar el éxito de la práctica deberemos presentar por consola en las dev tools del navegador los datos finales que manejaremos en nuestro sistema (en la PEC3 los renderizaremos en el navegador).

Concretando en la línea de la propuesta que os hacía antes deberíais implementar una instancia de la clase **DeckBuilder** con:

- propiedad(es) (arrays u objeto de arrays) que almacenen los valores de las diferentes propiedades de las cartas (los valores que rellenarán los selectores de la barra lateral izquierda en la web) p.e. `clase = ["Hunter", "Druid", "Mage"....]` - esto es recomendable hacerlo al instanciar la clase (instanciar `DeckBuilder` al recuperar los datos de la consulta al endpoint)
- el resultado de una consulta a la api como una matriz de instancias de la clase **Card**, por ejemplo `getCardsByClass('Hunter')` (se recomienda pensar la mejor manera de almacenar esos datos para no tener que repetir consultas realizadas previamente)

- el resultado de invocar el método/función `getCardById(cardId)` para recuperar datos de una carta en concreto

Nota: recordad construir correctamente la url de la imagen en base a la API de imágenes

Y presentar por consola:

- Cada una de las arrays de selectores
- Resultado de `getCardsByClass('Hunter')`
- Resultado de `getCardById(cardId)`

Nota: decidid una clase de entre las existentes en los selectores devueltos por la API y así mismo decidid una id de carta entre las existentes de la consulta `getCardsByClass`

Pensando en la PEC3

Si valoramos nuestra aplicación web como un total (PEC2 y PEC3) hay algunas preguntas que nos pueden ayudar a hacer un planteamiento suficientemente genérico y enfocado a interacción con el usuario para facilitar nuestro trabajo en la PEC3:

- Por sencillez de código os recomendaría guardar todos los endpoints en un objeto dentro del fichero `config.js` e importarlo en los ficheros que hagan falta. La dominio de todos los endpoints es siempre la misma y solo cambia la ruta.
- Será necesaria una estructura de datos para almacenar las cartas que el usuario quiera ir añadiendo a su mazo. Ese mazo tendrá dos funcionalidades añadir y quitar cartas.
- No todas las cartas tienen las mismas propiedades, esto es importante a la hora de definir la clase carta y tener todas las opciones en cuenta para no perder ninguna propiedad.
- Las consultas a la API no deben hacerse por duplicado, debemos guardar de algún modo los datos en estructuras locales de datos y saber qué peticiones se han hecho para no colapsar la API y ser más inmediatos presentando datos.
- Pueden existir varios selectores activos a la vez y ningún endpoint nos permite hacer ese filtrado en base una consulta, deberemos realizar una primera consulta a un endpoint (primer selector) y filtrar localmente los demás selectores.