

PEC 2

Lenguajes de desarrollo front-end



Universitat Oberta
de Catalunya

Información relevante:

- Fecha límite de entrega: 24 de octubre.
- Peso en la nota de FC: 15%.



Contenido

Contenido	2
Información docente	3
Presentación	3
Objetivos	3
Enunciado	4
Ejercicio 1 – Callbacks/Promesas/async-await (2 puntos)	4
Ejercicio 2 – Arquitectura MVC usando VanillaJS (4 puntos)	6
Ejercicio 3 – Arrays (4 puntos)	10
Formato y fecha de entrega	12

Información docente

Presentación

Esta práctica se centra en conocer las mejoras en el lenguaje de JavaScript a lo largo de los últimos años basados en el estándar ECMAScript, proporcionando fundamentos del lenguaje Web que es utilizado por todos los frameworks de hoy en día.

Objetivos

Los objetivos que se desean lograr con el desarrollo de esta PEC son:

- **Desarrollar** código JavaScript **usando las características de ES6-ES12**.
- **Utilizar el patrón de arquitectura MVC** en el desarrollo de una aplicación Web.

Enunciado

Esta PEC contiene 3 ejercicios evaluables. Debes entregar la solución de los 3 ejercicios evaluables (ver el último apartado).



Debido a que las actividades están encadenadas (i.e. para hacer una se debe haber comprendido la anterior), **es altamente recomendable hacer las tareas y ejercicios en el orden en que aparecen en este enunciado.**

Antes de continuar debes:

- Haber leído los recursos teóricos disponibles en la PEC.
- Crea una carpeta PEC2 e inicializa git en esa ruta. Al igual que en la PEC1, debes crear un fichero README.md en la raíz de la carpeta que contenga que contendrá al menos esta información:
 - Login UOC
 - Nombre y apellidos del alumno.
 - Breve descripción de lo realizado en esta PEC, dificultades, mejoras realizadas, si hay que tener algo en cuenta a la hora de corregir/ejecutar la practica o cualquier aspecto que queráis destacar.

Ejercicio 1 – Callbacks/Promesas/async-await (2 puntos)

Antes de continuar debes:

Descargar el fichero **PEC2_Ej1.zip** adjunto en la PEC y descomprimirlo dentro de la carpeta PEC2.

Observa el contenido del fichero ejer1.js:

```
const findOne = (list, { key, value }, { onSuccess, onError }) => {
  setTimeout(() => {
    const element = list.find(element => element[key] === value);
    element ? onSuccess(element) : onError({ msg: 'ERROR: Element Not Found' });
  }, 2000);
};

const onSuccess = ({ name }) => console.log('user: ${name}');
const onError = ({ msg }) => console.log(msg);

const users = [
  {
    name: 'Carlos',
    rol: 'Teacher',
  },
  {
    name: 'Ana',
    rol: 'Boss',
  },
];

console.log('findOne success');
findOne(users, { key: 'name', value: 'Carlos' }, { onSuccess, onError });
console.log('findOne error');
findOne(users, { key: 'name', value: 'Fermín' }, { onSuccess, onError });

/*
findOne success
findOne error
//wait 2 seconds
user: Carlos
ERROR: Element Not Found
*/
```

- a. El código anterior hace uso de *callbacks* para realizar una tarea después de hacer la búsqueda en un *array* (*find*). Dentro de la carpeta PEC2_Ej1, crea un fichero *ejer1-a.js* y documenta en el código, que se está realizando en cada línea, haciendo hincapié en el *callback* y el valor mostrado por la pantalla. (0.5 puntos)
- b. Copia el fichero resultado del ejercicio anterior a un fichero *ejer1-b.js*. Después modifica el código anterior para eliminar los *callbacks* y, en su lugar, hacer uso de promesas (hay que crearlas en la función *findOne*) y consumirlas en el código principal (con las palabras reservadas *then* y *catch*). Documenta en el código, línea a línea, que cambios has realizado. (0.5 puntos)
- c. Copia el fichero resultado del ejercicio anterior a un fichero *ejer1-c.js*. Después modifica el código anterior para hacer uso de *async/await* (ten en cuenta que deberás separar en una función el uso de la función *async*. Es decir, podrás crear una función extra en caso de ser necesario). Explica línea a línea a línea, dentro del propio código, qué has modificado. (0.5 puntos)
- d. Copia el fichero resultado del ejercicio anterior a un fichero *ejer1-d.js*. Observa que el código obtenido tras aplicar *async/await* se ha vuelto secuencial. Escribe y documenta en el propio código, como sería usando promesas y *async/await* la versión paralela, es decir, que se ejecuten todas las llamadas a la función *findOne* en paralelo, sin necesidad de esperar a que concluya la primera para ejecutarse la segunda. (0.5 puntos)

Durante el desarrollo del ejercicio ejecuta los comandos *git* necesarios para añadir esta nueva carpeta, ficheros y los cambios que realices en ellos al repositorio local *git* de la PEC.

Ejercicio 2 – Arquitectura MVC usando VanillaJS (4 puntos)

En este ejercicio vamos a construir, usando todas las novedades de JavaScript, una aplicación SPA (*Single Page Application*) sin utilizar ningún framework, lo que se conoce como programar usando VanillaJS o JavaScript puro. Es decir, VanillaJS es la referencia para crear aplicaciones basadas en JavaScript que no hacen uso de ningún framework (p.ej. Angular o Vue). Eso sí, seguiremos la arquitectura de MVC (Modelo-Vista-Controlador). Esta práctica te permitirá conocer la arquitectura que siguen frameworks como Angular, pero utilizando como lenguaje base JavaScript.

En primer lugar, debemos comprender cómo se construye una aplicación usando el patrón de arquitectura MVC utilizando VanillaJS. Para ello, os facilitamos un proyecto ya terminado llamado `Ejer2-1-TODO` que consiste en una aplicación SPA que realiza las 4 operaciones elementales CRUD (Create, Read, Update y Delete) sobre una lista de tareas (TODO). Una vez comprendido el proyecto `Ejer2-1-TODO`, tendréis que hacer una aplicación similar llamada `Ejer2-2-expense-tracker`.

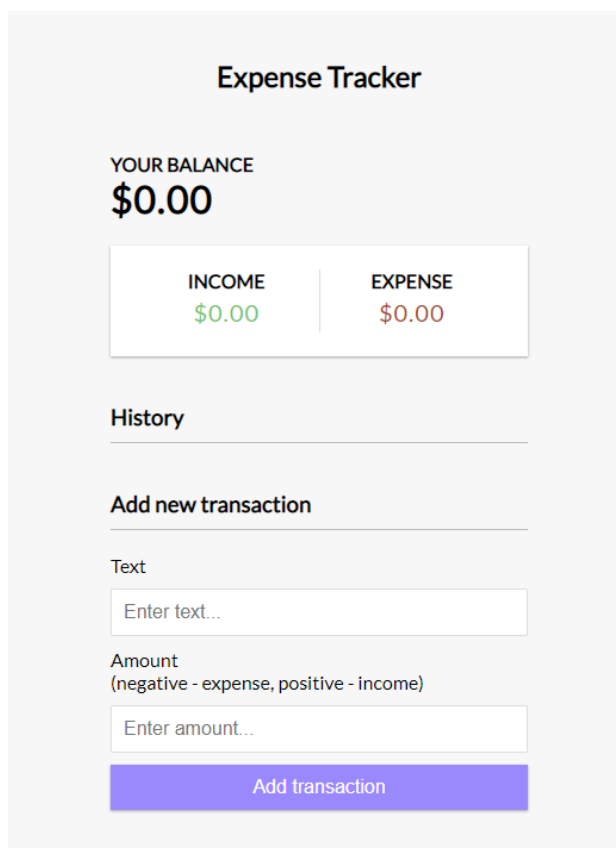
Antes de continuar debéis:

- Descargar el fichero `PEC2_Ej2.zip` adjunto en la PEC, que contiene los dos proyectos `Ejer2-1-TODO` y `Ejer2-2-expense-tracker`.
- Leer el recurso `P02.PEC2_Ejemplo_Arquitectura_MVC.pdf` que explica cómo está construido el proyecto `Ejer2-1-TODO`.

Una vez visto el ejemplo de una aplicación completa usando el patrón MVC **programada** en VanillaJS, debéis crear una aplicación MVC de control de gastos. Para ello debéis seguir los pasos del anterior código para crear un CRUD para el control de gastos. Es decir, se realizarán las operaciones de crear, borrar, editar y mostrar un registro de gastos.

En el código facilitado para comprender el MVC se realizan todas las operaciones sobre objetos TODO, en este caso, el dominio del problema ha cambiado para ser registros de gastos.

El código fuente inicial de la práctica (Ejer2-2-expense-tracker) genera la siguiente página:



The image shows a web application titled "Expense Tracker". It features a balance section at the top showing "YOUR BALANCE" as "\$0.00". Below this is a table with two columns: "INCOME" showing "\$0.00" in green and "EXPENSE" showing "\$0.00" in red. Underneath is a "History" section which is currently empty. At the bottom is a form titled "Add new transaction" with two input fields: "Text" (placeholder "Enter text...") and "Amount" (placeholder "Enter amount...", with a note "(negative - expense, positive - income)"). A purple "Add transaction" button is at the bottom of the form.

La información de los gastos que queremos gestionar es la siguiente:

- Texto del gasto
- Cantidad del gasto

Antes de empezar con el desarrollo de `Ejer2-2-expense-tracker`

Crea una carpeta `PEC2_Ej2`, dentro de esta carpeta crea un fichero texto que tenga como nombre `PEC2_Solucion_Ejercicio_2a.md` y responde a la siguiente pregunta sobre `Ejer2-1-TODO`.

- a. (0.50 puntos) Observa que se han creado funciones *handle* en el fichero controlador (`todo.controller.js`), las cuales son pasadas como parámetro. Esto es debido al problema con el cambio de contexto (*this*) que existe en JavaScript. Ahora mismo si no tienes muy claro que está sucediendo, revisa qué hacen las “fat-arrow” de ES6 sobre el objeto *this*, y prueba a cambiar el código para comprender qué está sucediendo cuando se modifica la siguiente línea:

```
this.view.bindAddTodo(this.handleAddTodo);
```

Por esta:

```
this.view.bindAddTodo(this.service.addTodo);
```

Responde, en un documento texto en el directorio de entrega a la siguiente pregunta:

¿Por qué es el valor de *this* es undefined?

A continuación, lo que se pide es que dentro de la carpeta `PEC2_Ej2`, crees una subcarpeta `Ejer2-2-expense-tracker` donde vas a ir modificando el código y estructura del proyecto para convertir un simple proyecto con html/css/js en una estructura MVC haciendo uso del potencial de ES6 y siguiendo el código de ejemplo (TODO) que se ha proporcionado.

En primer lugar, debes generar la estructura de directorios de nuestro proyecto (modelos-vistas-servicios-controladores) e ir pensando donde ubicar el contenido de los diferentes ficheros.

- b. (0.50 puntos) Construye las clases relativas a modelos, controladores, servicios, vistas y lanzador de la aplicación desde donde iras desarrollando la aplicación. En este punto sólo debes crear la estructura de ficheros que modelan nuestro problema. Es decir, organizar las clases relativas a modelos (`expense.model.js`), controladores (`expense.controller.js`), servicios (`expense.service.js`) y lanzadora (`app.js`).
- c. (0.50 puntos) Construye la clase modelo (anémico) que sea necesaria para esta aplicación.
- d. (1 punto) Construye la clase servicio que es la encargada de realizar todas las operaciones sobre una estructura de datos (donde se almacenará la información de todos los gastos).
- e. (0.50 puntos) Construye la clase vista que controlará todas las operaciones relativas a la vista.
- f. (0.50 puntos) Construye el controlador que es el encargado de poner en comunicación la vista con el servicio, en este proyecto.
- g. (0.50 puntos) El proyecto entregado no tiene implementado la actualización de un gasto concreto. Añade esta nueva funcionalidad.

Durante el desarrollo del ejercicio ejecuta los comandos `git` necesarios para añadir esta nueva carpeta, ficheros y los cambios que realices en ellos al repositorio local `git` de la PEC.

Ejercicio 3 – Arrays (4 puntos)

Utilizando los métodos nativos de `array` como son *filter*, *some*, *map*, *reduce*, debes completar los ficheros `core.js` que se suministran en el proyecto `PEC2_Ej3.zip` incluido en la PEC.

Antes de continuar debes:

Descargar fichero `PEC2_Ej3.zip` adjunto en la PEC

Para realizar el ejercicio solo debes modificar los ficheros `core.js`, **no se deben modificar los ficheros de tests `core.spec.js`**.

El objetivo es conseguir que la ejecución de los test de las pruebas definidas dé como resultado los tests `passed`.

Instrucciones:

- Instalar mocha con el comando `npm install -g mocha`.
- Completar el código del fichero `core.js` de cada subcarpeta
- Ejecutar cada uno de los test con la instrucción `mocha <nombre prueba>` por ejemplo `mocha a_map`

Tienes que completar el código del fichero `core.js` y conseguir que los test den como resultado `passed` para todas las carpetas suministradas en el zip.

- a. `map` – 0.5 puntos
- b. `filter` – 0.5 puntos
- c. `reduce` – 0.5 puntos
- d. `every` – 0.5 puntos
- e. `some` – 0.5 puntos
- f. `zoo` (ejercicio completo) – 1.5 puntos

Si necesitas más información, los siguientes enlaces disponen de abundante documentación sobre los métodos nativos de Array:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

https://www.w3schools.com/js/js_arrays.asp

Durante el desarrollo del ejercicio ejecuta los comandos `git` necesarios para añadir esta nueva carpeta, ficheros y los cambios que realices en ellos al repositorio local `git` de la PEC.

Formato y fecha de entrega

Tienes que entregar un fichero *.zip, cuyo nombre tiene que seguir este patrón: loginUOC_PEC2.zip. Por ejemplo: dgarciaso_PEC2.zip. Este fichero comprimido tiene que incluir los siguientes elementos:

La **carpeta PEC2**, y en su interior:

- Un fichero README.md en la raíz de la carpeta con la información indicada.
- La **carpeta oculta .git** con el contenido del repositorio local git.
- Una carpeta PEC2_Ej1 con el código completado y comentado siguiendo las peticiones/especificaciones del Ejercicio 1. Como lo que se solicita es modificar el código ejer1.js, debéis entregar un ejer1.js modificado para cada apartado de los solicitados. Es decir, debéis entregar cuatro ficheros: ejer1-a.js, ejer1-b.js, ejer1-c.js y ejer1-d.js. No es necesario entregar ni el original ejer1.js, ni el index.html, pero si es muy importante que todo el código que se entregue este comentado con los cambios realizados y el motivo de ellos.
- Una carpeta PEC2_Ej2:
 - Con el fichero PEC2_Solucion_Ejercicio_2a.md y responda a la pregunta formulada en el apartado A del Ejercicio 2.
 - Una subcarpeta Ejer2-2-expense-tracker con el código completado siguiendo las peticiones y especificaciones del Ejercicio 2
 - No es necesario entregar la carpeta ejemplo Ejer2-1-TODO
- Una carpeta PEC2_Ej3 con el código completado siguiendo las peticiones y especificaciones del Ejercicio 3.

Penalizaciones

- Entrega en otro formato que no sea el especificado (ej. en zip): **-0.75 puntos**
- Comprimir archivos dentro del zip: **-1 puntos.**
- Para cada ejercicio/apartado donde no se respete la nomenclatura exacta de las carpetas o ficheros indicados (símbolos, minúsculas, mayúsculas, etc.): **-0.75 puntos.**
- La no entrega del repositorio local git **-3 puntos**

El último día para entregar esta PEC es el **24 de octubre de 2021** hasta las **23:59**.
Cualquier PEC entregada más tarde será penalizada.