El ejercicio consiste en hacer una *demo* de una aplicación, *tiempoApp*, para mostrar las temperaturas del día actual de una ciudad que se introduce en ① por su **nombre** o **id**. Al pulsar *enter* en ① o el botón ②, los datos se muestran en ③ Además, contará con una tarea que actualizará los datos de forma automática (cada 5 minutos) siempre que en el l*ocalStorage* cuente con la información de la ciudad. En ④, por medio de un *timer* que se actualiza cada medio minuto, informará de cuánto tiempo ha pasado desde la última actualización. *Los números identificativos de las imágenes aparecen como comentarios en el código html para localizar las zonas de trabajo más fácilmente*

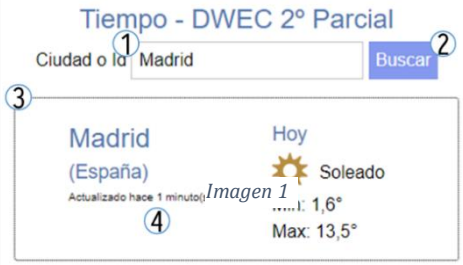

Imagen 1

Obligatorio: No cumplir estas normas acarreará la nulidad de la prueba.
El formato IIFE de la solución con ausencia de variables globales salvo el nombre de la variable de la app, `tiempoApp`. La **sola modificación del archivo** `tiempoApp.js`. **El uso de los métodos y variables indicados** (se pueden agregar métodos y variables auxiliares). La **operatividad** hasta el punto **4**. El uso de las estructuras de los datos que devuelve el servidor (presentes también, para consulta, en **datos.js** como variable globales `datos_json` y `ciudades_csv`). El uso de la cláusula `'use strict'`. El método `pedirDatos(peticion,callback)` se utilizará para pedir los datos al servidor. El argumento de `callback` serán los datos pedidos en formato texto. (ver comentarios en el archivo `tiempoApp.js` entregado).

A implementar

0.Nada más lanzarse la página se llama al método iniciar de `timerApp` con la dirección del servidor como argumento.

1.El método `iniciar` comprobará su argumento. Si existe, sustituirá la `url` por defecto del servidor por la del argumento. Agregará los eventos necesarios para la gestión de los controles ① (`enter`) y ② (`click`) al elemento con `id=buscar`. Su gestión correrá a cargo del método `gestionarEventosEntrada`. Creará los *timer*: `ultimaActualizacion` y `refrescarDatos` (puntos 6 y 7). Pedirá al servidor el archivo `ciudades.csv` que se gestionará en `convertirCSVaObjetos`.

2.Cuando se reciba `ciudades.csv`, se debe convertir en un array de objetos (uno por cada línea válida). En la primera línea se encuentran las llaves de los objetos que se guardarán en la variable `ciudades`. Solo se admitirán registros con datos en todos los campos. Hay que quitar los espacios iniciales y finales de cada registro. Se deben pasar los datos de los registros por expresiones regulares para validarlos. La validación de los valores de los campos `Nombre` y `País` es: deben empezar por una letra mayúscula seguida de 1 o más letras en minúsculas. Éstas pueden incluir las letras del alfabeto español (acentos, ñ,…), además de espacios. La validación del valor del campo `id` será: debe empezar por dos letras en mayúsculas (alfabeto inglés), seguidas de un guion y 2 dígitos, seguidos de otro guion y 3 dígitos. Las expresiones regulares se deben guardar en las variables que aparecen en el archivo `tiempoApp.js` entregado.

3.Un método, `gestionarEventosEntrada`, que, en el caso de que se haya pulsado la tecla *enter* ① o el botón ②, llame a `gestionarEntrada(valor)` donde `valor` es el dato introducido en ① (nombre de la ciudad o el id de la misma). Esta última comprobará que el valor cumple con la correspondiente expresión regular. Si es así, se buscará la existencia de la ciudad en el array generado en el punto anterior por medio de `buscarRegistro` (punto 5), buscando por `Nombre` o por `id`. Si existe la ciudad, se hará una petición al servidor, con el id de la misma, de sus temperaturas. La petición tendrá la forma de `datos?id=valor`. El gestor de la respuesta será el método `mostrarTemperaturas` (punto 4). Si existieran más de una ciudad, se cogerá la primera. Se guardará TODA la información de la ciudad, no sus temperaturas, por medio de `localeStorage` (punto 8). Tanto si el valor introducido no cumple la expresión regular como si no se encuentra una ciudad, se informará del error al usuario por medio de un `alert`.

Imagen 2


4.Una vez recibidos los datos de las temperaturas en `mostrarTemperaturas`, se buscarán las del día actual mediante `buscarRegistro` (punto 5). Si se encuentran, se mostrarán como la imagen 1. Si no se encuentran, como la imagen 2. La estructura html para mostrar los datos en ③ se encuentra como comentario en el archivo `tiempoApp.js` entregado. Una vez construida, se agregará como hijo a la etiqueta con `id=info`. Si ya tuviera información anterior, se eliminará, nodo a nodo, antes de insertar la nueva información. Para la creación las etiquetas html se deben emplear, exclusivamente, los métodos de `document: createElement` y `createTexNode`. Para la inserción/sustitución de la estructura, para la asignación de atributos, etc, también se deben utilizar métodos, no propiedades.

5.Un método, `buscarRegistro(objetos, campo, valor)`, que reciba en el primer argumento el array de objetos donde buscar; en el segundo, el campo por el que buscar en los objetos; y, en el tercero, el valor de este campo. Deberá devolver un array de objetos con los coincidentes. Si no hay ninguno, el array se devolverá vacío.

6.Un método, `ultimaActualizacion`, que se ejecute cada ½ segundo (y cada vez que se reciban temperaturas) que calcule el número de minutos ④ desde que se realizó la última petición válida de temperaturas. Los cálculos se harán con `Date`

7.Un método, `refrescarDatos`, que se ejecute cada 5 minutos (y nada más tener el array de ciudades construido) que compruebe que existen datos de una ciudad en `localeStorage` y, si es así, realice la petición de temperaturas de esa ciudad.

8.Un método, `almacenarCiudad`, con una ciudad por argumento, que almacene en un objeto, `tiempo`, en el `localeStorage`.

9.Un método, `propiedadCiudad`, con una propiedad de un objeto `ciudad` por argumento, que recupere y devuelva su valor si existe la propiedad en el objeto `tiempo`, en el `localeStorage`; en caso contrario, devolverá `undefined`.