Universitat Oberta de Catalunya

Aula

M4.253 - Program. JavaScript programadores aula 1

Tooling

Inicio:	Fin:	Solución:
19/02/20	**01/03/20**	Solución programada **02/03/20**

Presentación

Tooling es el término anglosajón usado para describir un **conjunto de herramientas** de forma genérica.

Cuando hablamos del tooling dentro del ecosistema de **JavaScript** nos referimos a todas aquellas utilidades auxiliares que nos servirán de apoyo y ayuda durante el desarrollo de nuestros proyectos.

Gran parte de las acciones que llevamos a cabo durante la fase de desarrollo son repetitivas y mecánicas. Tener un entorno que automatice estos procesos nos permitirá minimizar el tiempo que les dedicamos y centrarnos de forma más eficiente y efectiva en el desarrollo de nuestra aplicación.

Vamos a realizar una serie de pasos de forma guiada para implementar el tooling de una aplicación moderna en **JavaScript** que será nuestro entorno común de trabajo para el resto del curso.

Analizaremos a continuación cada uno de los desafíos que se nos presentan, las tecnologías que entran en juego y los problemas que quedan resueltos por cada una de ellas.

Node.js y NPM

Históricamente JavaScript había sido un lenguaje ejecutado en el navegador. Los navegadores son aplicaciones de escritorio que en su albergan en su interior un engine que compila y ejecuta Javascript.

Sin embargo, Ryan Dahl cambió este paradigma el 8 de Noviembre de 2009 con su trabajo y presentación de node.js en la JSConf .

Node.js es **un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome** , dicho de forma muy simple, una emulación de lo que sucede en el navegador sin el renderizado visual (no existe el objeto window).

Este hecho fue disruptivo ya que permitió que JavaScript, uno de los lenguajes de programación más populares y con una curva de entrada más baja, pasara a ser, simultáneamente, un lenguaje de cliente y de servidor permitiendo que muchos ingenieros de front-end pudieran dar el salto a back-end sin tener que cambiar de lenguaje.

Node.js permitió el desarrollo de muchas aplicaciones no destinadas directamente a los navegadores y facilitó la aparición de un ecosistema de utilidades a las que se les llama paquetes (packages).

Al instalar node.js en nuestro ordenador se instala también un gestor de paquetes: el **npm, node package manager** (gestor de paquetes de node). Esta utilidad nos permite (entre otras cosas):

- Crear nuestros propios paquetes de node (nuestras aplicaciones)
- Instalar otros paquetes en nuestra aplicación

Para poder tener una idea del tipo de aplicaciones a las que nos referimos podemos pasarnos por npmjs.com , un repositorio de paquetes de node.js, una especie de supermercado de utilidades. Podemos, por ejemplo, instalar React , la famosa librería de **Facebook** , mediante las instrucciones presentadas en esta página .

Instalación de node.js y npm

Ahora que sabemos qué es y para qué sirven node.js y npm podemos dirigirnos a su página oficial para descargarnos el binario más adecuado para nuestra plataforma de desarrollo.

Se nos presentan dos opciones: última versión o LTS (Long Term Support) siendo ésta última la opción más segura y aconsejable en nuestro caso ya que no tenemos necesidad de hacer uso de las últimas novedades ni versiones experimentales con poco soporte.

Para comprobar que la instalación ha sido correcta debemos abrir un terminal y ejecutar los siguientes comandos:

```
node -v
```

```
npm -v
```

Obteniendo como respuesta de cada uno de estos comandos la versión de node y de npm respectivamente. En el caso de haber instalado la versión LTS estos valores serán **10.15.1** para node y **6.4.1** para npm.

Aviso: el porcentaje de usuarios que utilizan un entorno Linux/Mac en el desarrollo de aplicaciones JavaScript supera de forma notable a los que usan entornos Windows. De aquí en adelante los comandos presentados serán todos en formato Linux/Mac. Los conceptos serán los mismos e independientes de la plataforma pero los usuarios de Windows quizás deban investigar en el caso de que los comandos no sean directamente ejecutables en la consola.

Creando la base de la aplicación

Nuestros ejercicios, en el caso de esta asignatura hablamos de las PECs, no son necesariamente aplicaciones de escritorio. Sin embargo, el tooling alrededor de nuestra aplicación sí lo será. Queremos poder hacer uso de algunas de las funcionalidades que podemos encontrar en la librería de paquetes de npm para nuestro proyecto y, como dijimos, automatizar y optimizar nuestros procesos en fase de desarrollo.

Para arrancar la creación de nuestro proyecto debemos buscar una carpeta en nuestro disco duro que nos parezca apropiado. En mi caso personal suele encontrarse en la ruta ` \Projects ` en la raíz del disco duro (cuyo formato será diferente para los usuarios de Windows y Linux/Mac).

Una vez tengamos ubicada nuestra carpeta de proyectos crearemos una nueva carpeta destinada a albergar nuestra aplicación y navegamos hacia ella:

```
mkdir PEC1 && cd PEC1
```

Una vez en su interior llamamos al inicializador de paquetes de node.js:

```
npm init
```

Podemos pasar por todas las preguntas que nos solicita el prompt con ` enter ` para aceptar los valores por defecto. Esto que puede parecer tremendamente sofisticado únicamente genera un

fichero llamado ` package.json ` en la raíz de nuestro proyecto con los campos y valores que hemos determinado.

El formato JSON (JavaScript Object Notation) es un estándar de la industria, no exclusivo de JavaScript, para representar objetos. Los objetos son pares de key/value envueltos entre los caracteres ` {} `.

Si damos un vistazo a nuestro ` package.json ` veremos algo parecido a esto:

```
{
"name" : "my-tooling" ,
"version" : "1.0.0" ,
"description" : "" ,
"main" : "index.js" ,
"scripts" : {
   "test" : "echo \"Error: no test specified\" && exit 1"
},
"author" : "" David < "dlampon@uoc.edu>" ,
"license" : "ISC"
}
```

Esta es la firma de nuestro paquete. Todos los campos son auto descriptivos excepto dos que pueden requerir de algo más de explicación:

- ` main `: es el punto de entrada de nuestra aplicación, normalmente ` html ` para una aplicación de navegador e ` index.js ` para una SPA o aplicaciones de escritorio o servidor.
- ` scripts `: es un objeto en el que podemos definir todos los comandos asociados a nuestro proyecto y que podrán ser ejecutados desde la raíz usando el comando ` npm run `

Todo este proceso automatizado mediante `npm init` podría haberse hecho de forma manual creando el fichero con el comando `touch package.json` y definido en su interior el objeto con las propiedades de nuestro proyecto.

Instalando una dependencia

Aunque hasta ahora solo tengamos una carpeta con un fichero de firma del paquete lo que hemos habilitado, entre otras muchas cosas, es la posibilidad de instalar dependencias.

Una dependencia es un paquete npm externo (www.npmjs.com) o de nuestro propio repositorio interno (como hacen muchas empresas) o bien local, de otro paquete/aplicación que hayamos desarrollado.

El primer paquete que vamos a instalar es Parcel.js usando el comando:

```
npm install parcel-bundler --save-dev
```

Esto dará paso al proceso de instalación. Veremos una barra de proceso en la que se descargará tanto el propio ` parcel ` como sus propias dependencias que quedarán instalados en la carpeta ` node_modules `. Aquí es donde residirán todas las dependencias de nuestro proyecto (y las subdependencias de nuestras dependencias).

Todos los ficheros deben estar en local para poder hacer uso de estas utilidades de node. Sin embargo, trabajar de forma conjunta con otros miembros del equipo o en proyectos open source podría ser bastante tedioso de tener que compartir esta carpeta ya que suele ser bastante pesada.

Lo brillante de este sistema de dependencias es que solo necesitamos indicar en nuestro fichero de firma del proyecto las dependencias y su versión para que todos los que trabajen en el mismo proyecto tengan el mismo estado de la aplicación.

Al recibir nuestro proyecto con el fichero ` package.json ` podremos tener las mismas dependencias (todas ellas) ejecutando el comando:

```
npm install
```

Haced la prueba:

- Borrad la carpeta ` node_modules ` con el comando:

```
rm -rf ./node_modules
```

- Ahora estáis en el mismo estado que alguien al que le paséis vuestro ` package.json ` por primer vez y ejecutad:

```
npm install
```

Los mismos ficheros, vuestras dependencias y la subdependencias de éstas, se han vuelto a instalar como antes de borrar la carpeta ` node_modules `. Es un sistema muy potente que facilita el trabajo

en equipo, la sincronización de los cambios del código y las versiones de los paquetes, muy importante para el trabajo con gestores de versiones tipo git .

Se espera que en las entregas de las PECs ***no se incluya la carpeta node_modules*** puesto que la carga y descarga de los ejercicios aumenta de forma innecesaria

Si revisamos nuestro ` package.json ` veremos que ha aparecido un nuevo par de valores:

```
"devDependencies" : {
  "parcel-bundler" : "^1.11.0"
}
```

Hay dos tipos de dependencias ` dependencies ` y ` devDependencies `. Las primeras son las que necesitaremos tanto en desarrollo como en producción como podrías ser alguna librería tipo ` react ` o ` d3 `. Las segundas son las dependencias de desarrollo como puede ser la librería de testing ` jest `. No necesitaremos testing en nuestra versión de producción expuesta al público pero sí lo necesitaremos en fase de desarrollo para poder testear nuestra aplicación.

¿Por qué necesitamos ` *parcel* ` para nuestro proyecto?

Lo que nos ofrece Parcel (entre otras cosas) es:

- Live server para nuestras aplicaciones
- Hot reload de nuestros cambios
- Bundling rápido de JS
- Poder usar módulos de ES6 (import/export)

Todo esto puede no significar mucho ahora mismo pero volveremos a estos puntos más adelante según empecemos a desarrollar y aclararemos el significado de cada uno de ellos.

Nuestro primer script

Los scripts son comandos de bash propios de cada proyecto que podemos definir para abstraer partes de la lógica asociada a los procesos de desarrollo. Vamos a crear un script para no tener que indicar el nombre de fichero cada vez que queramos ejecutar ` parcel index.html `.

Vamos a añadir esta línea en nuestro fichero ` package.json `:

```
  "scripts"  : {
    "dev" : "parcel index.html"  ,
    "test" : "echo \"Error: no test specified\" && exit 1"
},
```

Para poder ejecutar el comando ` dev ` debemos hacerlo de este modo:

```
npm run dev
```

Es muy discutible el valor aportado por este script y cambiar el uso de ` parcel index.html ` por el de ` npm run dev `. Son más o menos el mismo número de caracteres pero los beneficios son:

- No necesitamos que parcel sea una dependencia global del sistema
- Hemos abstraído el punto de entrada de la aplicación a un comando del proyecto
- Hemos generalizado la carga del proyecto a un comando ` dev ` de npm que es un estándar

El uso de scripts es mucho más valioso cuando los comandos tienen muchas opciones o flags adicionales. Éste es, por tanto, un ejemplo sencillo del uso que puede darse a los scripts.

Primeros pasos con nuestra aplicación

Todo lo que hemos hecho ahora ha sido definir nuestro tooling y la base de nuestra aplicación. Vamos a añadir tres ficheros de html, css y js para entender cómo nos va a ayudar `parcel` en fase de desarrollo.

Ha llegado el momento de empezar a escribir código y toca decidir qué editor usar. Mi consejo y recomendación es usar Visual Studio Code . Es mi editor predilecto y el que uso a nivel profesional. Cualquier otra opción que os resulte cómoda, útil y os permita sacar vuestro trabajo adelante es igualmente válida. No debemos olvidar que, a fin de cuentas, se trata solo de una herramienta más y lo único importante es que nos permita ser productivos.

Abrimos nuestro editor y creamos el fichero `index.html` con este contenido:

```
< html >
< head >
  < title >JS para programadorestitle>
  < link rel=  "stylesheet" type= "text/css" href= "styles.css"  />
head>
< body >
  < div id=  "main"  >div>
  < script src=  "./index.js"  >script>
body>
>
```

Creamos el fichero `styles.css` con este contenido:

```
#main  {
width :  100%  ;
height :  100%  ;
background-color : orange;
}
```

Creamos el fichero ` index.js ` con este contenido:

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

Estos tres archivos son una de las representaciones más básicas de una aplicación de navegador con la estructura en html, los estilos en css y la interactividad proporcionada por JavaScript.

Volveremos a nuestro terminal y ejecutaremos:

```
npm run dev
```

Con esto le estamos diciendo a ` parcel `que cargue el fichero index.html, recupere todas sus dependencias, cree una carpeta de distribución y la sirva en un servidor local en el puerto ` 1234 `. Veremos en el terminal el mensaje:

```
   Server running at http://localhost:1234
# Built in 14ms.
```

Si accedemos a esa dirección en nuestro navegador podremos ver el contenido de nuestra página.

Para poder ver el mensaje del `console.log` en nuestro fichero de ***JavaScript*** debemos abrir las developer tools de nuestro navegador Chrome , Firefox , Edge y acceder a la pestaña de `console`

Han sucedido procesos bastante sofisticados que han quedado escondidas a nuestros ojos para que podamos ver el resultado de la creación de nuestros archivos renderizados en el navegador.

Sin embargo, lo que puede resultarnos más útil para la realización de las prácticas es el hecho de que mientras no cerremos la consola en la que hemos ejecutado ` parcel ` podremos modificar cualquiera de los ficheros y los cambios se recargarán automáticamente en el navegador.

Probad a añadir texto en el html, cambiar alguno de los estilos o modificar el texto del ` console.log ` y veréis que el navegador se autorefresca y recarga los cambios.

Es recomendable tener una distribución de pantalla partida en la que una mitad la ocupe el navegador con `localhost:1234` y la otra mitad vuestro editor.

Preparando el entorno de testing

Para poder implementar testing en nuestra aplicación y poder, en mayor o menor medida, trabajar con metodologías TDD vamos a instalar la dependencia jest , desarrollada por los ingenieros de **Facebook** mediante el comando:

```
npm install -D jest
```

Veremos que nuestro fichero `package.json` contiene una nueva dependencia de desarrollo:

```
  "devDependencies" : {
    "jest" : "^24.1.0" ,
    "parcel" : "^1.11.0"
}
```

Si ejecutamos nuestro script de `test` que venía por defecto al instalar la aplicación mediante:

```
npm test
```

Obtendremos un mensaje de error puesto que es lo que queda especificado en el fichero el valor con ` echo \"Error: no test specified\" && exit 1 `. Modificaremos su valor para llamar al binario de la librería de testing:

```
"scripts"  : {
   "dev" : "parcel index.html" ,
   "test" : "jest --passWithNoTests --silent"
},
```

El flag `--passWithNoTests` evita que se muestre un mensaje de error por consola en caso de que no haya ningún test implementado. El flag `--silent` evita que se muestren errores adicionales no relacionados con los tests fallidos.

Ejecutamos este comando para ver la librería de testing en funcionamiento:

```
npm test
```

Primeros pasos con TDD

Test Driven Development es una práctica de desarrollo en la que se definen en forma de test qué funcionalidades debe pasar un programa (o una parte del mismo) en para ser válido y posteriormente se implementa el código que cumple ese requisito.

Vamos a ver un ejemplo sencillo. Debemos crear un fichero ` sum.js ` y otro ` sum.spec.js `.

Es la parte `spec` la que define que se trata de un archivo de tests. ***Jest*** recogerá todos y cada uno de los ficheros `spec` y pasará los tests uno por uno y mostrará el reporte por consola.

En el fichero ` sum ` vamos a añadir la siguiente función:

```
function sum (a, b) {
return a + b;
}
module .exports = sum;
```

En el fichero ` spec ` vamos a añadir el siguiente test:

```
const sum = require ( './sum' );
test( 'adds 1 + 2 to equal 4' , () => {
expect(sum( 1 , 2 )).toBe( 4 );
});
```

Si ahora ejecutamos el comando de test veremos que los tests se muestran un error. Según la filosofía TDD para que un test sea válido primero hemos de asegurarnos que falla en un caso incorrecto para luego corregirlo, ver qué pasa con éxito y seguir iterando. Modificamos el contenido del test por su valor correcto:

```
const sum = require ( './sum' );
test( 'adds 1 + 2 to equal 3' , () => {
expect(sum( 1 , 2 )).toBe( 3 );
});
```

Ahora, al ejecutar el comando de test, veremos que los resultados son correctos.

Módulos ES6

No vamos a entrar ahora mismo a explicar lo que es ES6, ni un módulo ni las diferencias entre los módulos ES6 y ES5 pero es evidente, viendo el funcionamiento de los test, su propósito y cómo deben ser usados.

`ES5 vs ES6` y la teoría de `módulos` son dos temas que veremos más adelante

Vamos a mejorar nuestro tooling permitiendo que en los ficheros de test podamos usar la nomenclatura de módulos ` ES6 ` (import/export) en lugar de los ` ES5/CommonJS ` (module.exports/ require).

` Parcel ` nos permite usarlos a nivel de nuestra aplicación pero no para los tests ya que no son parte directa ni se usan en la aplicación final. Por ello debemos realizar uso de un nuevo paquete para poder habilitar esta funcionalidad para nuestros tests.

```
npm install -D babel-jest @babel/core @babel/preset-env
```

Nuestro fichero ` package.json ` tendrá nuevas entradas en su apartado de dependencias de desarrollo:

```
"devDependencies" : {
   "@babel/core" : "^7.2.2" ,
   "@babel/preset-env" : "^7.3.1" ,
   "babel-jest" : "^24.1.0" ,
   "jest" : "^24.1.0" ,
   "parcel" : "^1.11.0"
}
```

Babel es un transpilador entre versiones de ECMAScript (la especificación de JavaScript). Nos permite escribir ES6 y automáticamente hará una conversión a ES5 para que node.js pueda ejecutarlo como si fuera el mismo código (node no soporta de forma nativa los módulos ES6).

babel-jest , @babel/core , @babel/preset-env

Con estos paquetes ya tenemos instaladas las funcionalidades para poder usar la sintaxis de módulos en nuestros tests de aplicación. Sin embargo necesitamos indicar mediante un archivo adicional de configuración cómo vamos a hacer esa conversión. Debemos crear un nuevo fichero ` babel.config.js ` con este contenido:

```
module .exports = {
presets: [
 [
    '@babel/preset-env' ,
  {
   targets: {
    node:  'current'
   }
  }
 ]
]
};
```

Ahora cambiaremos el formato de nuestro fichero ` sum.js ` por:

```
export default function sum (a, b) {
return a + b;
}
```

Y el de ` sum.spec.js ` por:

```
import sum from './sum';
test('adds 1 + 2 to equal 3', () => {
expect(sum(1, 2)).toBe(3);
});
```

Procederemos a ejecutar de nuevo los tests viendo cómo a pesar del cambio de sintaxis no se nos presenta ningún de error de ejecución ya que hemos habilitado el uso de las nuevas funcionalidades en nuestros tests como lo hace ` parcel ` en nuestros ficheros de aplicación.

Estilo de código y autocorrección

Un problema habitual es que al trabajar en equipo los estilos de código sean diferentes: hay quien prefiere tabulaciones a espacios o las comillas simples a las dobles. Es conveniente llegar a acuerdos de estilos para evitar que cada usuario que toque un fichero modifique el fichero en base a sus preferencias personales.

Para poder unificar estilos vamos a tener que instalar dos utilidades para nuestro IDE. Esto nada tiene que ver con la configuración de nuestro proyecto sino con las funcionalidades de nuestro editor. Los que usemos el editor **VSCode** deberemos tener instaladas las extensiones ESLint y Prettier . Simplificando mucho las funcionalidades de ambos **ESLint** define el estilo de código y detecta las incongruencias con las reglas establecidas y **Prettier** las corrige.

Instalaremos estas dependencias de desarrollo mediante:

```
npm install -D eslint prettier
```

Obteniendo estos resultados en nuestro ` package.json `:

```
"devDependencies" : {
   "@babel/core" : "^7.2.2" ,
   "@babel/preset-env" : "^7.3.1" ,
   "babel-jest" : "^24.1.0" ,
   "eslint" : "^5.13.0" ,
   "jest" : "^24.1.0" ,
   "parcel" : "^1.11.0" ,
   "prettier" : "^1.16.4"
}
```

Ahora podríamos definir nuestras propias reglas pero lo más habitual es reutilizar conjuntos de reglas que ya han sido probadas y se han convertido en estándar o al menos en la base de todo proyecto y modificarlas o extenderlas a necesidad. Uno de estos códigos de estilo es el publicado y mantenido por airbnb . Para poder empezar a usarlas necesitaremos usar esta dependencia así como todas las asociadas. Usaremos en este caso el comando ` npx ` para simplificar la instalación de las dependencias asociadas:

```
npx install-peerdeps --dev eslint-config-airbnb
```

Necesitaremos dos dependencias más para evitar conflictos entre las **ESLint** y **Prettier** , inhabilitar el formateo de código de **ESLint** pero permitir mostrar errores mientras escribimos:

```
npm install -D eslint-config-prettier eslint-plugin-prettier
```

Obteniendo este objeto de dependencias de desarrollo:

```
"devDependencies" : {
   "@babel/core" : "^7.3.3" ,
   "@babel/preset-env" : "^7.3.1" ,
   "babel-jest" : "^24.1.0" ,
   "eslint" : "^5.3.0" ,
   "eslint-config-airbnb" : "^17.1.0" ,
   "eslint-config-prettier" : "^4.0.0" ,
   "eslint-plugin-import" : "^2.16.0" ,
   "eslint-plugin-jsx-a11y" : "^6.2.1" ,
   "eslint-plugin-prettier" : "^3.0.1" ,
   "eslint-plugin-react" : "^7.12.4" ,
   "jest" : "^24.1.0" ,
   "parcel" : "^1.11.0" ,
   "prettier" : "^1.16.4"
}
```

A continuación crearemos un archivo `.eslintrc.json` con este contenido:

```
{
"extends" : [ "airbnb" , "prettier" ],
"plugins" : [ "prettier" ],
"rules" : {
   "prettier/prettier" : [ "error" ]
},
}
```

Y otro ` .prettierrc ` con este contenido:

```
{
"printWidth" : 100 ,
"singleQuote" : true
}
```

Y por último habilitaremos que nuestro editor formatee el fichero que tengamos abierto cada vez que salvemos su contenido. Iremos al menú ` Code > Preferences > Settings ` y buscaremos la opción ` Format On save ` y la activaremos.

Con esto conseguimos, por ejemplo, que se eliminen de forma automática los espacios y líneas extra así como intercambiar las comillas dobles por las simples de forma automática garantizando sobretodo que al tratarse de una configuración de proyecto (habilitada por las extensiones añadidas al editor) todos los participante del proyecto (o en este caso del aula) tengan el mismo tipo de código y apliquen las mismas reglas de forma automática.

Resumen

Hemos conseguido paso a paso configurar un entorno de trabajo común para todos con funcionalidades muy potentes y avanzadas. Han quedado algunos temas a desarrollar, especialmente el tema de módulos, pero que requieren su propio espacio de desarrollo y explicación.

Lecturas complementarias y enlaces:

- Gestor de versiones de node
- El formato JSON

- Yarn
- Git: control de versiones
- Jest
- ¿Qué es TDD?
- Nuevas funcionalidades ES6
- Babel
- npx
- ESLint
- Prettier

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.