Índice

1. Entorno de trabajo

En esta primera unidad se configurará el entorno de trabajo y se describirán todos los elementos que se utilizarán durante el desarrollo del curso. El alumnado no está obligado a utilizar las herramientas que aquí se recomiendan aunque siempre será más fácil seguir las explicaciones de este modo.

1.1. Navegadores

La función principal del navegador es obtener documentos **HTML** e interpretarlos para mostrarlos en pantalla. En la actualidad, no solamente descargan este tipo de documentos sino que muestran con el documento sus imágenes, sonidos e incluso vídeos *streaming* en diferentes formatos y protocolos.

Algunos de los navegadores web más populares se incluyen en lo que se denomina una *Suite*. Estas *suite* disponen de varios programas integrados para leer noticias de *Usenet* y correo electrónico mediante los protocolos **NNTP**, **IMAP** y **POP**.

Los primeros navegadores web sólo soportaban una versión muy simple de **HTML**. El rápido desarrollo de los navegadores web propietarios condujo al desarrollo de lenguajes no estándares de **HTML** y a problemas de interoperabilidad en la web. Los más modernos (como *Google Chrome*, *Mozilla*, *Netscape*, *Opera* e *Internet Explorer* / *Microsoft Edge*) soportan los estándares **HTML** y **XHTML** comenzando con **HTML 4.01**, los cuales deberían visualizarse de la misma manera en todos ellos.

Para acceder a estos recursos, se utiliza un identificador único llamado **URL** (*Uniform Resource Locator*).

El formato general de una URL es **protocolo://máquina/directorio/archivo**.

- Si no se especifica el directorio, toma como directorio el raíz.
- Si no se especifica el fichero, toma alguno de los nombres por defecto. Usualmente estos nombres por defecto son similares a **index.html** o **index.php**.

Por ejemplo **https://www.google.es** donde se accede al recurso **www.google.es** usando el protocolo **https**.

La comunicación entre el servidor web y el navegador se realiza mediante el protocolo **HTTP**, aunque la mayoría de los navegadores soportan otros protocolos como **FTP** y **HTTPS** (una versión cifrada de **HTTP** basada en *Secure Socket Layer* o **SSL**).

Principales navegadores

- *Microsoft Edge* (antiguo *Internet Explorer*)
- *Mozilla Firefox*
- *Google Chrome*
- *Safari*
- *Opera*

En este manual se hará referencia a *Mozilla Firefox* o *Google Chrome*. El motivo de usar estos es la gran cantidad de herramientas para depuración que poseen incluso en su versión estándar. Para la mayoría de acciones con este será suficiente. En el caso de *Firefox* está disponible una versión que amplía las herramientas de desarrollo ***Firefox Developer Edition***.

Es una práctica imprescindible para un buen programador web comprobar que cualquier desarrollo funcione correctamente en los principales navegadores.

1.2. Manual de referencia

Elemento imprescindible a la hora de aprender un lenguaje es su referencia. Para ello, *Mozilla* mantiene la MDN (*Mozilla Developer Network*) con una extensa referencia de *JavaScript* llena de ejemplos, `aquí` que también está traducida (aunque en muchas ocasiones sin tanto contenido) `aquí` . A lo largo del manual se hará referencia estos contenidos para ampliar la información.

1.3. La consola web

La mayoría de los navegadores incorporan de manera nativa herramientas para facilitar el desarrollo, aunque una de las que más se utilizarán en este manual será la consola web. El uso de extensiones de navegador y *plugins* permite ampliar las funcionalidades de esta consola, pero no serán necesarios para el correcto desarrollo de este manual.

Tanto en *Mozilla Firefox* como en *Google Chrome* se accede a la consola web pulsando la tecla `F12` .

En el desarrollo de este manual se usará el objeto **`console`** para mostrar resultados (también se podría usar **`alert`** pero es más molesto). Este objeto no es parte del lenguaje, pero sí del entorno y está presente en la mayoría de los navegadores ya sea en las herramientas del desarrollador o en el inspector web.

Así pues, dentro de la consola se podrá visualizar los mensajes que se envíen así como ejecutar comandos *JavaScript*.

Los métodos que más se van a usar con la API de **`console`** son:

- **`log()`**, muestra por la consola todos los parámetros recibidos, ya sean cadenas u objetos,
- **`info()`**, **`warn()`** y **`error()`** muestran mensajes representados con varios colores,
- **`dir()`**, enumera los objetos recibidos e imprime todas sus propiedades,
- **`assert()`**, permite comprobar si se cumple una aserción booleana,
- **`time()`** y **`timeEnd()`**, calcula el tiempo empleado entre las dos instrucciones.

Por ejemplo, si se ejecuta el siguiente código:

```
console.log([1,2,3]);
```

se obtendrá:

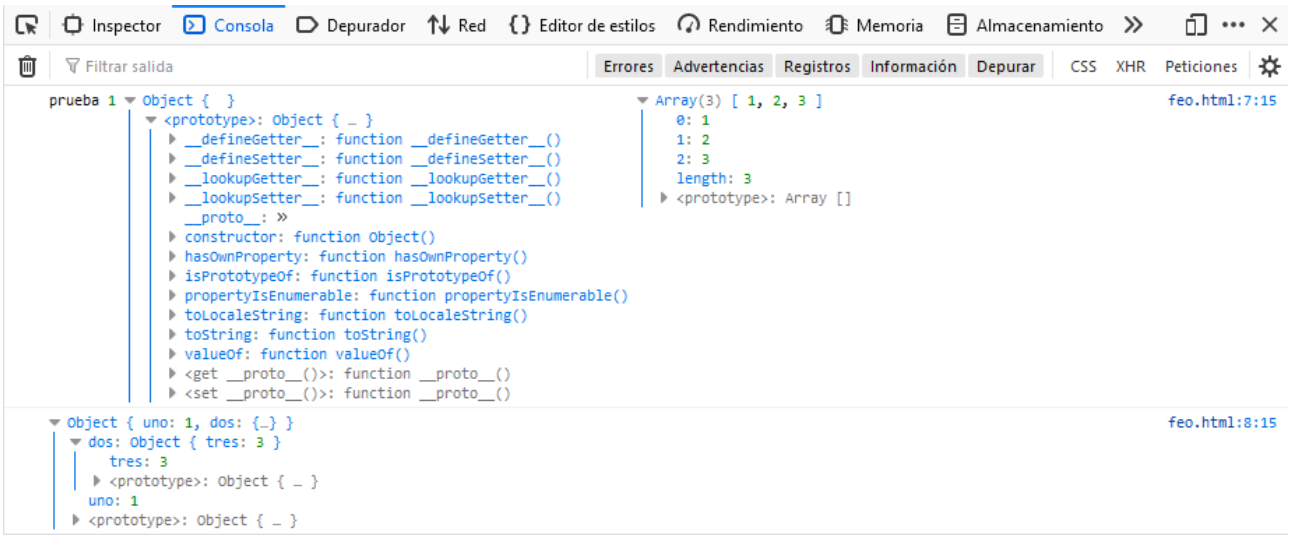

Figura 1: Aspecto de la consola de un navegador web

Más información del uso de **console** aquí .

Además de **console**, existen otros métodos para interactuar con el usuario a través del navegador:

- **alert()** permite mostrar al usuario información literal o el contenido de variables en una ventana independiente. La ventana contendrá la información a mostrar y el botón Aceptar .

```
alert("Hola mundo");
```

- **confirm()** se activa un cuadro de diálogo que contiene los botones Aceptar y Cancelar . Cuando el usuario pulsa el botón Aceptar este método devuelve el valor **true**; el botón Cancelar devuelve el valor **false**. Con ayuda de este método el usuario puede decidir e influir de ese modo directamente en el programa.

```
let respuesta;
respuesta=confirm ("¿Desea cancelar la suscripción?");
alert("Usted ha contestado que "+respuesta);
```

- **prompt()** abre un cuadro de diálogo en pantalla en el que se pide al usuario que introduzca algún dato. Si se pulsa el botón Cancelar el valor de devolución es **false/null**. Pulsando en Aceptar se obtiene la cadena de caracteres introducida y se guarda para su posterior procesamiento. El segundo parámetro de **prompt** (**Alicante** en este caso) es su **valor por defecto** y es opcional.

```
let provincia;
provincia=prompt("Introduzca la provincia ","Alicante");
alert("Usted ha introducido la siguiente información "+provincia);
console.log("Operación realizada con éxito");
```

1.4. Editores de código (entornos de desarrollo)

Existen diversos entornos de desarrollo, desde los más sencillos (*Brackets*, *Notepad++*, *Sublime* o *Visual Studio Code*) a interfaces más complejas como *Aptana* o *Eclipse*. En este manual se usará ***Visual Studio Code***. Es muy potente y posee un gran ecosistema de *plugins* para ampliar su funcionalidad.

Aquí algunos manuales libres de uso para aprender más sobre ***Visual Studio Code*** en castellano: para su instalación aquí y para su personalización aquí .

1.5. Editores *online*

Para hacer pequeñas pruebas el hecho de tener que crear un documento **HTML** que enlace a un documento *JavaScript* se puede hacer tedioso. Para ello existen diferentes "parques" donde jugar con código. Dos de los más conocidos son ***JSBin*** (**http://jsbin.com**) y ***JSFiddle*** (**http://jsfiddle.net**).

Ambos permiten probar código en caliente e interactuar con **HTML**, **CSS** y usar bibliotecas de terceros como *jQuery*. Es una buena alternativa a un editor pero para probar pequeñas porciones de código.

1.6. Control de versiones

Durante el curso se utilizarán repositorios *Git* tanto para la entrega de prácticas como para facilitaros el disponer de un repositorio con control de versiones.

Para instalar *Git*:

- en GNU/*Linux Ubuntu*:

```
sudo apt-get update
sudo apt-get install git
```

- en *Microsoft Windows*: instalación basada en "siguientes" aquí .

Para usar *Git* en *Visual Studio Code* se recomienda repasar las guías contenidas aquí y **sobre todo** aquí . Aquí un ejemplo del uso de *Git* dentro de *Visual Studio Code*.

1.7. Servidor *NodeJS*

Al trabajar en un entorno cliente no será necesario la instalación de un servidor que interprete el código que se escribe ya que esta tarea la realizará el propio navegador web. Bien es cierto que *JavaScript* no sólo sirve para la implementación de páginas web, sino que es posible realizar aplicaciones de servidor que se comunique directamente con bases de datos o incluso realizar aplicaciones de escritorio independientes.

A pesar de que en este curso no se utilizarán estas utilidades, siempre es buena idea conocer una tecnología escalable y potente como esta.

Para instalar NodeJS en *Microsoft Windows* pincha aquí , para hacerlo a través de **nvm** pincha aquí y para hacerlo en *GNU/Linux Ubuntu* o *Debian* es necesario utilizar estos comandos:

```
curl -fsSL https://deb.nodesource.com/setup_16.x | sudo -E bash -
sudo apt-get install -y nodejs
```

Aunque en la mayoría de distribuciones funciona la instalación tradicional:

```
sudo apt-get install npm nodejs
```

Tras la instalación desde el terminal del sistema operativo es posible ejecutar ficheros *JavaScript* anteponiendo **node** al nombre del fichero a ejecutar:

```
node ./MiPrimerFichero.js
```

Todo el resultado de ese fichero aparecerá en el terminal y no en un navegador. Es interesante conocer que *NodeJS* es un intérprete muy versátil. Esta versatilidad está generada por los módulos instalables que se le pueden aplicar. Existen muchos que añaden características como ejecutar acciones en el sistema operativo, interpretar el código en *Android* o añadir funcionalidades a los proyectos. Todos los módulos con los que viene instalado *NodeJs* se encuentran descritos en su documentación oficial aquí , aunque esto no se estudiará en este módulo.

1.8. *Clean Code*

Las técnicas de código limpio aparecieron por primera vez en el libro "*Clean Code: A Handbook of Agile Software Craftsmanship*", lanzado en 2008 y escrito por **Robert Cecil Martin**. Con sus largos años de experiencia encontró que el problema principal en el desarrollo de *software* era precisamente el mantenimiento, es decir, un código mal escrito en su primera versión puede funcionar, pero va a generar grandes problemas en el futuro. De hecho, se estima que la proporción media de lectura y escritura de código fuente es de 10 a 1. Esto significa que las personas pasan más tiempo intentando entender los códigos existentes que creando código nuevo.

Clean Code no es un conjunto de reglas estrictas que hay que cumplir si quieres tener el carné de programador, sino una serie de principios que ayudan a producir código **intuitivo**, **escalable**, **accesible** y fácil de **modificar**.

En este contexto, intuitivo significa que cualquier desarrollador pueda entenderlo de inmediato. Por ejemplo, la secuencia de ejecución de todo el programa sigue una lógica y tiene una estructura sencilla, es decir, la tarea de cada clase, función, método y variable es comprensible a primera vista.

Un código se considera fácil de modificar cuando es flexible y escalable, lo que también ayuda a corregir los posibles errores que pueda tener. El código limpio es muy fácil de mantener ya que las clases y los métodos son reducidos y, si es posible, tienen una sola tarea clara. Además, siempre tienen el resultado esperado y ha sido sometido a pruebas unitarias.

Las **ventajas** de este tipo de programación son obvias: se vuelve **independiente del desarrollador/a** que lo ha creado. Cualquier persona puede trabajar con él y evita problemas como los que conlleva el código heredado (código personalizado e improvisado por quien lo creó). Su mantenimiento también se simplifica porque los errores son más fáciles de buscar y corregir. A priori, todo son ventajas.

1.8.1. Principios generales

Crear *Clean Code* implica tener en cuenta ciertos principios fundamentales durante el desarrollo del *software*. No se trata de seguir unas instrucciones concretas que indican cómo programar en ciertas situaciones, sino más bien de una reflexión sobre el propio trabajo. Su significado despierta polémica en la comunidad del desarrollo: lo que alguien considera "limpio", puede ser "sucio" para otros, por tanto, la limpieza del código acaba siendo algo subjetivo.

- **Los nombres son importantes**

  El código no solo debe funcionar y ser interpretado por la máquina que lo ejecuta, sino que también debe ser comprensible para el humano medio, especialmente si se trabaja en proyectos colaborativos. La legibilidad del código siempre es más importante que su concisión. No tiene sentido escribir un código conciso si el resto no lo entiende.

  Un ejemplo de creación de código legible sería nombrar las variables que siempre deben ser comprensibles. Por ejemplo, no es posible entender la siguiente variable sin una explicación:

  ```
  var feo;
  ```

  Sin embargo, con el siguiente nombre, la misma variable se explica por sí sola:

  ```
  var elapsedTimeInDays;
  ```

  Esto también ocurre con el nombre de las **funciones**, **clases**, **métodos** o **componentes** que se desarrollen:

  - deben ser autoexplicativos, precisos y entregar la idea de lo que hacen,
  - el tamaño no importa. Si la función o parámetro necesita de un nombre largo para demostrar lo que realmente representa, es lo que debes hacer.

- **Regla del *boy scout***

  Existe un principio que afirma que si sales del área en la que estás acampando, debes dejarla más limpia que cuando la encontraste. Si se lleva esta regla al mundo de la programación se puede adaptar como dejar el código más limpio de lo que estaba antes de editarlo.

- **KISS (*Keep It Simple, Stupid*)**

  El ser humano está acostumbrado a pensar de manera narrativa, así que el código funciona de la misma forma. Es una historia y sus autores deben preocuparse por el modo en el que la historia es presentada. Para estructurar un código limpio es necesario crear funciones simples, claras y pequeñas.

  El código debe ser lo más sencillo posible evitando cualquier complejidad innecesaria. En programación nunca hay una única manera de resolver un problema, por lo que el desarrollo que sigue el principio KISS debe preguntarse si se podría dar con una solución más simple que la que se ha encontrado.

  Existen dos reglas para crear código KISS:

  - las funciones deben ser pequeñas,
  - estas deben ser aún más pequeñas.

- **DRY (*Don't Repeat Yourself*)**

  Es una concreción de KISS: cada función debe tener una representación única y, por lo tanto, inequívoca dentro del sistema general del *Clean Code*. Lo contrario de DRY es **WET** (*We Enjoy Typing*) que es cuando el código contiene duplicaciones innecesarias.

  Por ejemplo, el nombre de usuario y la contraseña aparecen dos veces en el código para utilizarlos en diferentes acciones. En lugar de programarlos por separado, ambos procesos pueden agruparse en una sola función. De esta manera, el código WET (húmedo) con sus redundancias se convertirá en código DRY (seco).

- **Eliminar lo innecesario: YAGNI (*You Aren't Gonna Need It*)**

  Se basa en la idea de que sólo se debe añadir funciones al código cuando sea estrictamente necesario. Está íntimamente relacionado con los métodos del desarrollo ágil de *software*. De acuerdo con este principio, en lugar de comenzar a programar partiendo de un concepto general, la arquitectura del *software* se desarrolla paso a paso para poder reaccionar a cada problema de forma dinámica.

- **Comentar solamente lo necesario**

  Los comentarios pueden hacerse, sin embargo, deben ser necesarios. Hay que tener claro que **los comentarios mienten** y esto tiene una explicación lógica. Lo que ocurre es que, mientras el código es modificado, los comentarios no se tocan nunca. Estos son olvidados, y por lo tanto, no retratan la funcionalidad real del nuevo código que seguro que ha cambiado varias veces desde su escritura inicial.

- **Tratamiento de errores y pruebas**

  Las **cosas pueden salir mal** y el **usuario medio de tu código es idiota**, por lo que hay que escribir código que contemple todas las posibilidades a las que el código puede enfrentarse, es decir, tratar las excepciones de forma correcta para tener el control ante cualquier situación.

  Además, un código sólo se considera limpio después de ser válido a través de pruebas que deben diseñarse en función del código escrito. Sólo cuando las pruebas confirmen que el código es válido será cuando de verdad lo sea.

El *Clean Code* es un concepto que soluciona con eficacia uno de los principales problemas que gran parte de los proyectos de desarrollo: el mantenimiento.

1.9. Programación reactiva con *React*

React es una biblioteca *JavaScript* centrada en el desarrollo de interfaces de usuario. También es un excelente aliado para hacer aplicaciones web, SPA (*Single Page Application*) o incluso aplicaciones para móviles. Para ello, **React** dispone un completo ecosistema de módulos, herramientas y componentes capaces de ayudar al desarrollo de aplicaciones avanzadas con poco esfuerzo. Es una biblioteca desarrollada inicialmente por *Facebook, es software* libre y a partir de su liberación acapara una creciente comunidad de desarrolladores.

El objetivo de **React** es desarrollar aplicaciones web de una manera más ordenada y con menos código que si usas *JavaScript* puro o bibliotecas como *jQuery* centradas en la manipulación del **DOM**. Permite que las vistas se asocien con los datos, de modo que si cambian los datos, también cambian las vistas.

Está basado en componentes. Una interfaz de usuario es básicamente creada a partir de un componente el cual encapsula el funcionamiento y la presentación. Unos componentes se basan, además, en otros para solucionar necesidades más complejas en aplicaciones. La mejor parte de todo es que estos componentes son reutilizables entre desarrollos. Por este motivo se encuentra una amplia comunidad que libera sus propios componentes para que cualquier persona los pueda usar en sus proyectos. Por tanto, antes de desarrollar algo en **React** conviene ver si alguien ya ha publicado un componente que haga lo que necesita el proyecto.

Es una biblioteca completa, adecuada en muchos tipos de proyectos y que permite un desarrollo ágil, ordenado y con una arquitectura fácilmente sostenible.

Figura 2: Logotipo oficial de **React**.

1.10. Hoja de ruta re *React*

React es un ecosistema muy amplio y es muy sencillo perderse entre tanta funcionalidad. Cuando se está iniciando en esta tecnología y se necesita una funcionalidad concreta, a veces buscar en internet puede suponer un peligro. Existen muchas soluciones para resolver un mismo problema y en ocasiones se acaba "matando moscas a cañonazos", es decir, utilizando una potente biblioteca para resolver un problema que *React* implementa de forma nativa.

Con el fin de ordenar un poco todo este contenido y de centrar el desarrollo de esta nueva tecnología sin sobresaltos, *Kamran Ahmed* (**@kamranahmedse**) ha creado una hoja de ruta que es muy recomendable seguir para adentrarse en ella.

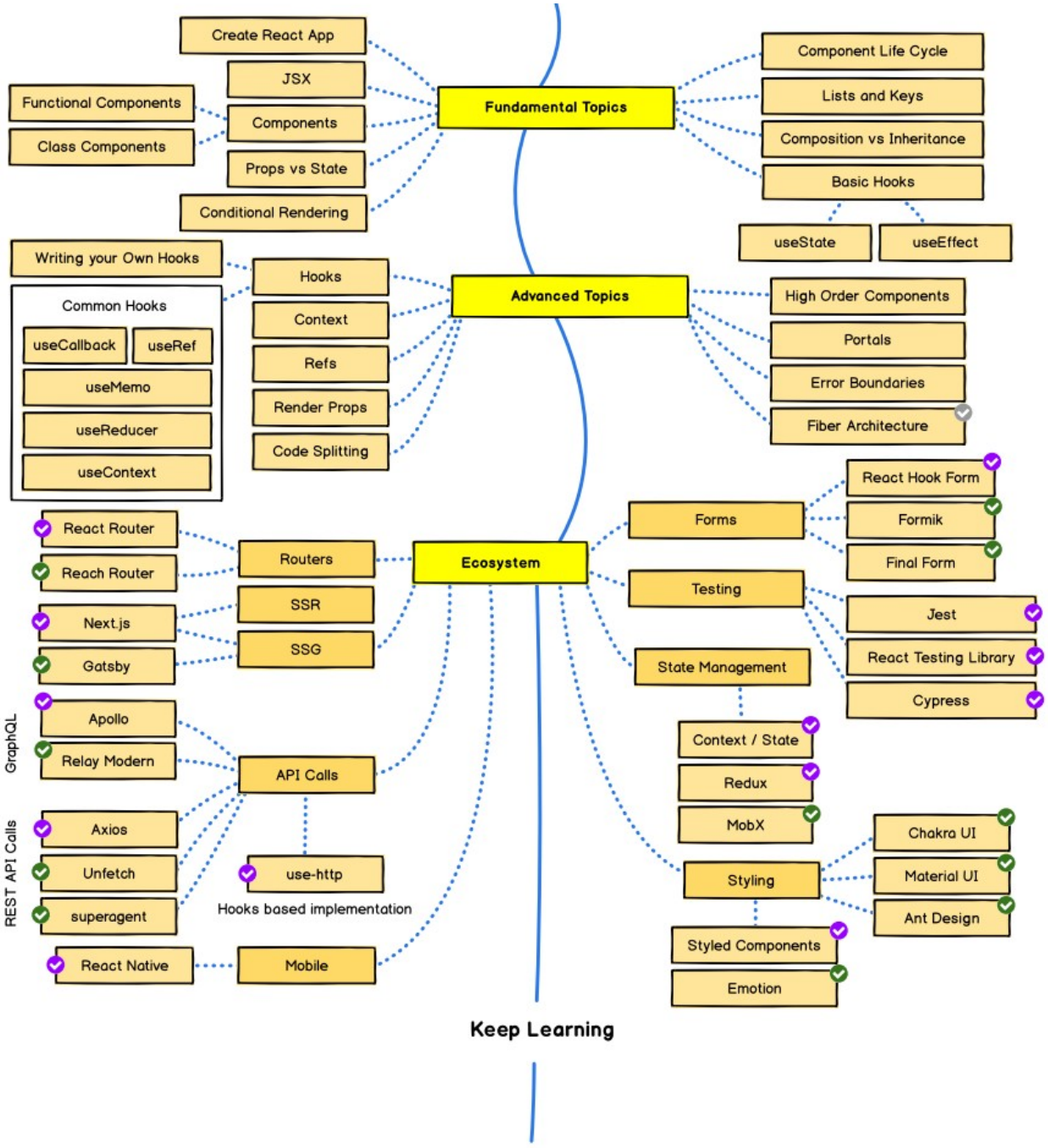

Figura 3: Hoja de ruta de **React**

No es recomendable utilizar una herramienta de un nivel si antes no se han dominado, o al menos conocido, la mayoría de las utilidades del nivel anterior. En este manual se estudiarán todos los temas básicos (*Fundamentals Basics*) que permitirá conocer la base del funcionamiento de **React**. Además se trabajará con herramientas de la parte avanzada (*Advanced topics*) como *hooks* avanzados y contextos; y otras que forman parte del ecosistema (*Ecosystem*) como la administración de estados avanzada y las rutas.

1.11. Creación de un nuevo proyecto

Antes de la instalación es necesario limpiar la caché de **npm** para asegurar que se obtiene la versión más reciente del *software*. Para ello se utiliza es siguiente comando:

```
npm cache clean –force
```

Después se instalará la herramienta ***Vite*** que permitirá la creación de un proyecto **React** a través de un asistente facilitando la configuración inicial del proyecto:

```
npm install -g vite
```

Una vez instalado en global (modificador **-g**) hay que situarse en la carpeta en donde se va a localizar el proyecto. Una vez allí, se lanza el siguiente comando para crear una aplicación:

```
npm create vite@latest nombre-de-la-app
```

Hay que tener presente que no es posible utilizar mayúsculas y evitar el uso de caracteres especiales en el nombre de la aplicación. **nombre-de-la-app** contendrá la ruta de la carpeta en donde se guardan los archivos de la aplicación.

Es posible que solicite la instalación de un paquete denominado **create-vite@X.X.X** . Se trata de un paquete necesario para mostrar el asistente que ayudará a la creación del proyecto. Tras esto, en el terminal aparecerá un corto asistente que permitirá elegir cuál será la tecnología del proyecto (la opción **Vanilla** implica usar sólo código *JavaScript* nativo), y el lenguaje que se usará en él: *TypeScript* o *JavaScript*.

Tras la creación del proyecto, el propio terminal informa de los siguientes pasos a realizar:

- moverse hasta la carpeta del proyecto creado,
- instalar todas las dependencias del proyecto y
- lanzar el proyecto en un servidor local.

```
cd nombre-de-la-app
npm install
npm run dev
```

Esto creará un servidor web con las bibliotecas necesarias para empezar a trabajar en el proyecto. La aplicación es accesible desde la red local lo que permite acceder al servidor desde un teléfono móvil o tableta para ir comprobando en tiempo real el desarrollo en este tipo de dispositivos.

Además de este comando es conveniente familiarizarse con los siguientes:

- **npm run build**, que genera el código final del proyecto. No es un compilador sino un transpilador que interpreta y optimiza el código del proyecto a código de producción,
- **npm run preview**, vista previa local del proyecto antes de su construcción,
- **npm run dev**, que inicia el servidor de desarrollo.

Para todo esto, también es posible utilizar el gestor de paquetes **yarn** creado por *Facebook* que optimiza varias de las características de **npm**, aunque en este manual se seguirá trabajando con el último.

Hay que recordar que cuando se utilice **Git** para el control de versiones y se realice un **clone** en un entorno de trabajo nuevo será necesario instalar de nuevo todas las bibliotecas en ese nuevo

equipo. Para ello se utiliza el fichero `package.json` (donde están indicadas las dependencias del proyecto) y el comando

```
npm install
```

Esto instalará todas las dependencias del proyecto para poder continuar trabajando. Huelga decir que estas bibliotecas no es conveniente almacenarlas en el repositorio de **Git** por el elevado espacio que suelen ocupar.

Para instalar otras bibliotecas al proyecto bastará con utilizar el gestor de paquetes **npm** como de costumbre. Por ejemplo, para instalar las bibliotecas de *Firestore*

```
npm install firebase
```

De este modo el proyecto accederá de forma local a las bibliotecas necesarias.

1.12. Descripción de los ficheros

La instalación del proyecto trae consigo la construcción de una estructura de carpetas que es necesario conocer:

- `dist`, alberga los ficheros de producción de la aplicación. Estos ficheros se construyen interpretando el código de la carpeta `src` a través del comando **`npm run build`**,
- `node_modules`, contiene las dependencias del proyecto (bibliotecas asociadas al proyecto),
- `public`, carpeta raíz de nuestro servidor donde se encuentra el `index.html` de la aplicación,
- `src`, contiene el código del proyecto **React**,
- `package.json`, que contiene información del proyecto, así como enumera sus dependencias tanto para desarrollo como para producción,
- `.gitignore`, que es el archivo para indicar a **Git** que ignore ciertos archivos durante la sincronización del proyecto,
- `package-lock.json`, archivo que no se debe editar generado por **npm** con información sobre el proyecto y las dependencias que contiene.

Para empezar con **React** no se necesita más que entrar en la carpeta `src` y empezar a editar su código a través del fichero `App.jsx` , que será el componente índice de la aplicación que se va a construir.

El fichero `main.jsx` no es un componente, sino que es el va a dibujar toda la aplicación en el **DOM** del navegador. Para ello utiliza el método **`createRoot`** del objeto **`ReactDOM`** del siguiente modo:

```
ReactDOM.createRoot(lugar-donde-se-va-a-dobujar).render(que-se-va-a-dibujar);
```

De este modo se introduce el componente **`<App />`** en el **DOM** del navegador, y ese es el motivo por lo que este componente debe ser el índice de la aplicación, para mantener el código simple y accesible.

A medida que se vaya desarrollando la aplicación se irán creando nuevos componentes para realizar tareas más específicas o instalando componentes de terceros que permitan realizar tareas sin necesidad de invertir tiempo en programarlas de nuevo. Todo esto se realizará con el gestor de paquetes **npm.**