Lemoncode - Estructura Front

versión: 0.12

Fecha de publicación: 27 de Octubre de 2022

Sumario

Tópicos de esta guía:

Nombrando y creando carpetas

Prácticas que tenemos definidas

- Usar minúsculas.
- Separar palabras con guiones medios.
- Usar nombres cortos pero descriptivos.
- Salvo que lo tengamos claro no crear carpetas demasiado temprano.
- Uso de singulares y plurales.
- Uso de *barrel*.

Uso de minúsculas

Esto viene porque *Linux* es *case sensitive* y *Windows* no, aunque ya los IDE y herramientas de desarrollo lo suelen resaltar es posible que se nos pase este detalle y por ejemplo tener que el *build* local en máquinas Windows funciona, pero en CI en una máquina Linux, y este fallo es muy complicado de depurar ya que casi nadie tiene una máquina Linux / Mac OS en local.

Podéis ver en muchos proyectos *open source* que no siguen esta aproximación, nosotros lo entendemos así porque en el resto de Europa y en USA la mayoría de *Front Enders* desarrollan con Mac OS.

Separación de palabras

Nosotros solemos usar guiones medios (*kebab case*) porque nos resulta más legible.

Podría valer otro separador, pero es importante que sea consistente, si por ejemplo se usa guion bajo, que se use siempre guion bajo, no mezclar.

En el caso de los ficheros veremos que usamos dos separadores, el "-" y el ".", esto lo veremos más adelante, así como el razonamiento.

Nombres cortos pero descriptivos

En el caso de carpetas intentamos que los nombres sean cortos, ya que normalmente acompañarán a ficheros que cuelguen de la raíz, pero le damos importancia a que éstos describan lo que hacen.

En caso de que haga falta más de una palabra, usamos guiones medios para separarlas (mismo razonamiento que para nombre de ficheros).

Salvo que lo tengamos claro no crear carpetas demasiado temprano

Salvo que esté muy claro que es necesario crear una carpeta o estructura de carpetas (por que tengamos un armazón inicial definido, o tengamos claro que van a ser necesarias), preferimos crear la carpeta sólo cuando sea necesario.

Por ejemplo:

- Tenemos una carpeta `*components*`, de primeras vamos soltando componentes aquí. Conforme esta carpeta vaya creciendo en contenido, se irá haciendo más grande, por lo que hará falta refactorizarla y agrupar componentes comunes en subcarpetas.
- Tenemos debajo de *components*, un componente de *layout*, otro que *wrapea* un *input*, un *combobox*, una lista... esos componentes serán candidatos para ser agruparlos en la carpeta `components/form`, sin embargo la de layout mientras no crezca no lo planteamos.

Siguiendo esta aproximación, nos evitamos tener una estructura de carpetas de las que hay veces que sólo cuelga un fichero, este mal se llama "el principio de clarividencia del programador", a priori en muchos escenarios no sabemos por dónde va a crecer el proyecto.

Es malo tanto tener muchas carpetas y profundidad sin contenido, como tener poca profundidad de carpetas y muchos ficheros.

Uso de singulares y plurales

A nivel de carpetas, salvo que estemos hablando de adjetivos o términos no contables (*common*, *core*), utilizaremos el plural, ya que las carpetas suelen agrupar elementos (*components*, *scenes*, *layouts*).

Siguiendo esta aproximación mientras que es más natural usar el plural en las carpetas, en los ficheros usaremos siempre el singular.

Uso de ficheros barrel

Es buena práctica debajo de cada subcarpeta generar un fichero *index.ts* que sea el punto de entrada para los ficheros de esa carpeta, algo así como:

./index.ts

```
export * from "./kanban.container";
export * from "./providers/kanban.provider";
```

De esta manera:

- Las importaciones son más cortas.
- Si se renombra un fichero, no hay que cambiar las importaciones, sólo tocar el *barrel*.
- Si se renombra una carpeta, sólo hay que cambiar el *barrel*.
- Si se reagrupa en subcarpetas, sólo hay que cambiar el *barrel*.
- Tenemos una forma de indicar al programador que va a consumir elementos de nuestra carpeta, qué funcionalidad queremos hacer "*pública*". Si bien el desarrollador puede usar ruta completa, va a detectar que el desarrollador original no quería exponer el fichero por algún motivo.

Estructura de carpetas.

Cuando creamos una aplicación de tamaño medio, seguimos la aproximación de *pods*.

```
my-application/
├─ common/
├─ common-app/
├─ core/
├─ layout/
├─ pods/
├─ scenes/
```

Para proyectos más grandes, candidatos dividir en cargas bajo demanda o submódulos podemos añadir un nivel de carpetas más: *submodules*. Aquí se pueden elegir dos aproximaciones:

```
my-application/
├─ common/
├─ core/
├─ modules/
│  ├─ my-module/
│  │  ├─ common-app/
│  │  ├─ layout/
│  │  ├─ pods/
│  │  ├─ scenes/
```

Aquí, dependiendo de cómo se enfoque, cada submódulo podrían compartir *concerns* comunes.

Otra opción sería la siguiente:

```
my-application/
├─ modules/
│  ├─ my-module/
│  │  ├─ common/
│  │  ├─ common-app/
│  │  ├─ core/
│  │  ├─ layout/
│  │  ├─ pods/
│  │  ├─ scenes/
```

En este caso cada módulo es una isla independiente. Si necesitáramos compartir *concerns* comunes podríamos plantear crear una librería o una carpeta común o *core* a módulos.

common

Bajo esta carpeta se incluyen todos los componentes y funcionalidades que se pueden usar en cualquier parte de la aplicación, y que podrían ser reutilizables en otras aplicaciones (no tienen relación con el dominio de la aplicación), por ejemplo:

- Un componente de UI para mostrar un calendario.
- Una función para parsear un fichero CSV cualquiera.
- Unas expresiones regulares para validar un email o un formato dado.
- ...

La funcionalidad que se publique aquí, si se demuestra útil (ver más abajo principio de promoción), se puede extraer a una librería estándar para que otros proyectos de la empresa lo usen o incluso promocionarlo a *open source*.

common-app

Bajo esta carpeta se incluyen todos los componentes y funcionalidades que se pueden usar en cualquier parte de la aplicación, pero que NO son reutilizables en otras aplicaciones (SÍ tienen relación con el dominio de la aplicación), por ejemplo:

- Un panel de filtro de paciente de un hospital que usa campos específicos de la base de datos.
- Un listado de pacientes con una jerarquía de datos específica del dominio.
- ...

Es decir, son componentes / funcionalidades reusables que no se pueden reaprovechar en otras aplicaciones ya que están atadas al dominio del proyecto.

Esta carpeta es opcional y puede generar controversia ya que se puede incurrir en el riesgo de que la carpeta acabe siendo un cajón de sastre si no hay criterio de qué debe incluir.

core

En esta carpeta incluimos las funcionalidades que son transversales al proyecto, es decir ficheros que podemos usar a diferentes niveles de la aplicación y que guardan información, por ejemplo:

- Las rutas de navegación.
- El perfil del usuario / roles de seguridad.
- El tema de la aplicación.
- Contextos y proveedores de la aplicación.
- Cachés de datos comunes.
- ...

A veces nos puede costar distinguir qué debe entrar en la carpeta *common* y la *core*. Como regla para distinguirlo:

- La carpeta *common* está más orientada a componentes y funcionalidades independientes que podemos incrustar en la aplicación. Estas funcionalidades actúan como cajas negras independientes.
- La carpeta *core* almacena funcionalidades que se usan y están íntimamente ligadas a toda la aplicación, por ejemplo información de la aplicación que necesitamos consumir en diferentes niveles.

layout

En *layout* definimos las páginas maestras, es decir, el armazón de una página (*wireframe*). Por ejemplo, una ventana de aplicación puede tener la siguiente estructura:

```
--------------------------
|       Header           |
--------------------------
|                        |
|        body            |
|                        |
|                        |
--------------------------
|       Footer           |
|                        |
|                        |
--------------------------
```

Otro *layout* puede ser el de la ventana de *login* en el que simplemente centramos el contenido en horizontal y vertical.

Cada página de la aplicación (escena) elegirá qué *layout* puede usar y en el área de *body* aparecerá el contenido de la escena.

Esto se puede elaborar más:

- Un *layout* podría tener más de una zona para poder incluir contenido de página.
- Un *layout* podría tener *sublayouts*.

Aconsejamos no entrar en este nivel de complejidad salvo que sea estrictamente necesario.

Además, la elección de *layout* también se puede realizar a nivel de página (escena), o se puede intentar hacer una agrupación de páginas por *layout* a nivel de *router*, pero esto depende mucho del *router* que estemos usando y de la versión del mismo (por ejemplo React Router en unas versiones lo permite en y otras no).

pods

En un *pod* encapsulamos una funcionalidad rica e independiente. Lo asemejamos a una isla de código, es estanca y sólo se comunica con el exterior mediante funcionalidad transversal que podemos encontrar en *core* o usando componentes comunes (*common*, *common-app*).

Un ejemplo de carpetas de pod:

```
my-app/
├─ pods/
│  ├─ patient/
│  │  ├─ components/
│  │  ├─ patient.api.ts
│  │  ├─ patient.mapper.ts
│  │  ├─ patient.business.ts
│  │  ├─ patient.container.tsx
│  │  ├─ patient.component.tsx
│  │  ├─ patient.vm.ts
│  │  ├─ index.ts
```

El concepto de *pod* lo cubriremos con más extensión cuando cubramos la sección *estructura de un pod*.

¿Por qué no encapsulamos esto en páginas / escenas?

El objetivo de una escena (página) es:

- Elegir qué *layout* va a usar.
- Manejar posibles parámetros a nivel de *query string* que puedan llegar.
- Elegir que *pod* o *pods* va visualizar.

Queda fuera del alcance mezclar otros detalles de implementación.

¿Por qué una isla de código?

El objetivo del pod es que sea lo más independiente posible, de esta manera:

- Cuando llega un desarrollador se puede centrar en un *pod* concreto y conocer la funcionalidad común y *cross* que le afecte, no tiene que conocer el proyecto completo.
- Es más fácil de mantener ya que vamos tocando por islas estancas y un cambio en un *pod* no tiene por qué afectar a otro.
- Es más fácil orientarlo al lenguaje del dominio para ese *pod*, creando *ViewModels* específicos.

- Todo lo que impacta al *pod* se encuentra cerca (mismo nivel de carpeta o inferiores) y el desarrollador no tiene que ir navegando por el proyecto para encontrar lo que necesita.

scenes

Una forma de definir las páginas de la aplicación es usando la nomenclatura de escena, es decir, cada página de la aplicación es una escena, ¿Por qué este término? Bueno, es como si a fin de cuentas sólo usáramos este elemento para hacer una composición de *pods* y *layouts*, intentando romper con el término de *página* que solemos asociar al concepto de *página web* en la que metíamos un mazacote de funcionalidad.

Si no estás cómodo con el término, siempre puedes usar "*pages*" o "*views*". Lo importante es ser consistente y que todo el equipo esté de acuerdo con el nombre que se usa.

Un ejemplo con más niveles

Cuando el proyecto crece, éste se va haciendo más complejo, por lo que se hace necesario añadir más niveles de carpetas. Salvo que tengamos claro desde un principio la dirección en la que va a crecer el mismo, es conveniente añadir niveles de carpetas conforme se vayan necesitando.

Un ejemplo de cómo puede quedar *common*:

```
common/
├─ components/
│  ├─ forms/
├─ hooks/
├─ validations/
```

Sobre este ejemplo:

- Conforme el proyecto crece, nos solemos encontrar que creamos varios componentes comunes reusables. Si esto pasa es buena idea crear una carpeta de *components* para tenerlo localizado.
- Es muy normal que en un proyecto se acabe con una capa que haga de *wrapper* de aquellos componentes comunes que incluyan por ejemplo la fontanería para trabajar con un motor de gestión de formularios o un tematizado dado. De ahí que se cree la subcarpeta *forms* dentro de *components*. Ésto sería candidato a promocionar a librería.
- Lo mismo pasa con los *hooks* y las validaciones; temas genéricos merecen la pena tenerlos localizados (una validación de NIF, o de código HL7..., un *hook* que realiza algo genérico como detectar si hemos hecho clic fuera de un elemento.)

Si esta ruta no encaja en tu proyecto, también te puedes plantear organizarlo por característica funcional. Todo depende de cómo crezca el proyecto.

Un ejemplo para la carpeta *core*:

```
core/
├─ api-configuration/
├─ auth/
```

```
├─ constants/
├─ router/
├─ theme/
```

En este caso organizamos temas transversales que se pueden usar en toda la aplicación:

- Configuración para nuestra *api rest*.
- Autenticación (preguntar por usuario, roles...).
- Constantes a nivel global.
- Configuración de rutas de navegación de páginas.
- El tema de la aplicación (colores, fuentes, tamaños...).

Igual que para la carpeta *common* este nivel de subcarpeta se puede ajustar a tus necesidades o no, algo importante a evitar es terminar con un montón de niveles de carpetas que tengan uno o dos ficheros únicamente.

Aquí lo importante es tener claro cuál es el punto de entrada e identificar los ficheros y carpeta por su tipo, de cara a saber dónde tocar.

Un ejemplo de carpeta *pod* (en este caso con carpetas y ficheros):

```
pod/
├─ patient/
│  ├─ api/
│  ├─ components/
│  ├─ patient.business.ts
│  ├─ patient.component.tsx
│  ├─ patient.container.tsx
│  ├─ patient.hooks.ts
│  ├─ patient.mapper.ts
│  ├─ patient.vm.ts
│  ├─ index.ts
```

En este caso podemos plantear crear una carpeta *api* si vamos a tener varios ficheros que monten la api para este *pod*.

De la misma manera, si con el *root container* y *component* no tenemos suficiente, es buena idea crear una carpeta *components* para agrupar los subcomponentes que se usen en el pod (y si hubiera muchos agruparlos a su vez por funcionalidad).

A nivel de ficheros, dependiendo de la complejidad del pod, iremos creando ficheros con funcionalidades independientes o rompiendo en carpetas si estas funcionalidades pueden ser agrupadas. Lo importante es no crear ficheros porque sí, sino ir agrupando funcionalidades en la medida de lo posible.

El objetivo principal es que los pods estén organizados y que tengan una firma determinada y común a todos ellos. De esta manera los haremos intuitivos para poder así distinguir rápidamente el punto de entrada e identificar donde tenemos que acudir para tocar algo.

Normalmente un pod no debería crecer de forma desmesurada. Si esto pasase tendríamos que pensar si romper en más de un *pod*.

Importación de rutas relativas y aliases

Un tema importante a la hora de importar módulos es tener en cuenta las rutas relativas y los aliases.

Por ejemplo, en un *pod* complejo nos podríamos encontrar con algo como:

```
import { Calendar } from
"../../../../common/components/calendar.component";
```

Esto presenta varios problemas:

- Legibilidad: es difícil de leer y entender.
- Mantenibilidad: si cambiamos la estructura de carpetas, tenemos que ir a todos los ficheros que importan este componente y cambiar la ruta (con suerte *VS Code* nos podría ayudar a hacerlo, pero no siempre es así).
- Productividad: si las herramientas de *VS Code* no automatizan los *imports*, tengo que ir probando subiendo carpeta por carpeta.

Para evitar esto aplicamos varias soluciones:

- Por un lado con el uso de *PODs* los *path* relativos (código específico del *POD*) se quedan controlados dentro de la isla de *PODS*.
- Por otro lado para acceso a carpetas globales (*common*, *core*,...) usamos aliases que nos permitan importar esos módulos sin usar rutas relativas largas y completas.

Es decir, se nos puede quedar con un alias de este tipo:

```
import { Calendar } from "@/common/components/calendar.component";
```

ó

```
import { Calendar } from "common/components/calendar.component";
```

Si trabajas con *webpack* y *typescript* una forma cómoda de configurar esto es:

- Le indicas en *Typescript* que el prefijo @ es un alias a la carpeta *src*. De esta manera creas un alias por cada carpeta que cuelgue justo de src (*common*, *core*, *scenes*, *pods*, *layout*...).

tsconfig.json

```
    "esModuleInterop": true,
+    "baseUrl": "src",
+    "paths": {
+      "@/*": ["*"]
+    }
  },
```

Para no repetir configuración e incurrir en posibles errores en *webpack*, nos podemos bajar el plugin *tsconfig-paths-webpack-plugin* que nos permite usar los mismos aliases que tenemos en typescript.

```
npm install tsconfig-paths-webpack-plugin --save-dev
```

Y consumirlo en *webpack*:

```
const HtmlWebpackPlugin = require("html-webpack-plugin");
+ const TsconfigPathsPlugin = require('tsconfig-paths-webpack-plugin');
const path = require("path");
const basePath = __dirname;

module.exports = {
  context: path.join(basePath, "src"),
  resolve: {
    extensions: [".js", ".ts", ".tsx"],
+   plugins: [new TsconfigPathsPlugin()]
  },
```

De esta manera los *imports* a carpetas raíz quedan de esta manera:

```
import { LoginPage } from "@/scenes/login";
```

Nombrando y creando ficheros

Otro tema importante es qué convenciones seguimos para nombrar ficheros.

En cuanto a *casing* siempre usamos *lowercase* (minúsculas) para nombrar los ficheros. La razón es la misma que con las carpetas; mientras que en Windows los nombres de los ficheros no son *case sensitive*, en *linux* sí, por lo que si tu entorno de desarrollo es Windows y tu entorno de CI/CD o producción es *linux* te puedes encontrar con problemas difíciles de detectar ya que en tu local la aplicación va a funcionar.

¿Y si un fichero tiene varias palabras? Aquí cubrimos dos casos:

- Lo que define al fichero a nivel de dominio / funcionalidad, lo separamos con guiones "-" (*kebab-case*).
- Lo que define técnicamente al fichero, lo separamos con puntos "." (*dot-case*).

¿A qué se debe esta complejidad?

- De esta manera podemos identificar de manera clara la funcionalidad del fichero (ésta se encuentra descrita al principio del fichero).
- De la misma forma ocurrirá con la definición técnica o tipado del fichero. Ésta se encontrará al final. Además, al reutilizar este sistema de nomenclatura de manera sistemática, los ficheros podrán ser filtrados de manera mucho más eficiente según distintos criterios de búsqueda.

Por ejemplo:

```
my-calendar.component.tsx;
my-calendar.business.tsx;
my-calendar.hooks.tsx;
my-calendar.styles.css;
```

En cuanto a los sufijos a usar en la parte técnica del fichero, dependerá de los que el equipo vea oportuno. Algunos de los que solemos usar:

- **.container.tsx**: Para componentes contenedores, es decir, que tienen lógica y presentación.
- **.component.tsx**: Para componentes tontos, es decir, que solo tienen presentación (o al menos no contienen el peso fuerte de lógica)
- **.business.ts**: Para ficheros con funciones puras que resuelvan problemas (también se pueden plantear con estados, pero todo lo que no tenga que ver con React, código plano JS)
- **.hook.ts**: Para almacenar *hooks* (funciones que usan *hooks* de React).
- **.api.ts**: Para almacenar código para gestionar llamadas a *API's rest*.
- **.mapper.ts**: Para convertir de *ApiModel* a *ViewModel*.
- **.model.ts**: Para definir modelos de datos (de API o global). Dependiendo del equipo se podría pensar en romper en más niveles de modelo (por ejemplo *api-model*, *domain-model*, etc. Es recomendable no meter más complejidad si no es necesario).
- **.validation.ts**: Para extraer lógica de validación de un formulario. Esto también es muy útil ya que es fácil de probar y lo tenemos localizado y no esparcido por el árbol de componentes de UI.

- **.vm.ts**: Para definir el modelo de la vista. Mientras habrá ocasiones en que ese módulo sea un mapeo directo de lo que nos traemos de la API (sobre todo si los que desarrollan la API son los mismos que desarrollan la aplicación), en otros casos puede ser que tengamos que mapear valores y realizar transformaciones para que se ajusten a la vista (esa complejidad la sacamos de la UI y la pasamos a una función pura JS, fácil de probar).

A pesar de que en las carpetas podíamos definir si usar singulares y plurales, en los ficheros consideramos recomendable usar el singular únicamente (aunque esto depende de lo que decida el equipo). Así evitamos la siguiente fuente de fallo (muy común):

```
client.component.tsx
clients.component.tsx
```

Es muy fácil qué a la hora de importarlo o usar el componente / clase / función, se nos baile una *s* y nos quedemos tontos pensando por qué no funciona ese código (los fallos tontos son los que más duelen). Para evitarlo se pueden usar las siguientes alternativas:

```
client.component.tsx
client-collection.component.tsx
```

ó

```
client.component.tsx
client-list.component.tsx
```

También podemos comentar qué hacer por contenido / tamaño del fichero:

- Si un componente / función, está muy cohesionado a por ejemplo un componente de un fichero, y éste es pequeño, podemos plantear dejar la funcionalidad en el mismo fichero, y más tarde extraerla si vemos que el fichero crece mucho o si la funcionalidad se puede reutilizar (aquí lo mismo comentarlo con el equipo, hay desarrolladores que prefieren tener un sólo fichero para cada cosa).
- Si uno de los ficheros empieza a crecer demasiado, o si tiene sentido agrupar cierta funcionalidad o romperla, nos podemos plantear crear una subcarpeta y realizar la división del fichero por contenido. Esto puede pasar si por ejemplo una *api* crece mucho, o si decidimos romper un componente en subcomponentes.

Estructura de un pod.

La solución de *pods* que usamos, que está inspirada en *Ember*, parte de que cuando desarrollamos en un proyecto medio/grande es complicado:

- Encontrar el fichero que queremos editar.
- Saber si un cambio en un fichero puede afectar a otra página / funcionalidad que lo use.

- Hacernos con el conocimiento de la funcionalidad que queremos editar.
- No impactar en el trabajo de otros compañeros (generar conflictos).
- Detectar si algo es funcionalidad común o específica.

Para ello planteamos utilizar el concepto de *pod*:

Un pod es un módulo de código que tiene todo lo necesario para funcionar.

Es decir, salvo funcionalidades *cross/común* (que las movemos a carpetas raíz, como vimos en la sección de carpetas), cada *pod* tiene encapsulado todo lo necesario para funcionar y no depende de nada más que de lo que está dentro de su carpeta (salvo carpetas comunes / transversales).

Así pues (salvo funcionalidad común), un pod tiene:

- Definidas a qué API's va a acceder y qué modelo de datos de API va a consumir.
- Definidas qué *ViewModels* va a usar.
- Definido su contenedor, componentes y subcomponentes.
- Definido su modelo y validaciones.

Esto permite que un desarrollador que no conozca el proyecto pueda, en un tiempo razonable, estudiar cómo funciona un *pod* en concreto y saber dónde tocar para introducir una modificación.

De esta manera:

- El tiempo de entrada en un proyecto es menor.
- Las colisiones con otros compañeros son menores (cada uno toca su *pod*).
- Con los *viewModels* enfocamos los datos a la vista.
- En caso de hacer una migración es más fácil ir migrando por *pods*.
- A la hora de añadir pruebas unitarias o plantear seguir TDD para ciertas partes del código, es más fácil ya que rompemos el código en piezas simples que hacen una y sólo una cosa.

Sobre la separación de ficheros dentro de un mismo pod, al igual que que comentamos en la sección de "*Nombrando y creando ficheros*", la idea es separar el *pod* en piezas que hagan una sola cosa.

Además, dentro de los pods, un tema muy importante a la hora de definir los *ViewModels* es que no debo importar en un pod un *ViewModel* de otro *pod*. Por muy parecido que sea tengo que volver a crearlo ¿Por qué? Porque no quiero tener acoplamientos entre *pods*. Una situación distinta es tener un *ViewModel* que se use a lo largo de toda la aplicación (por ejemplo un *Lookup Id/Valor*), en cuyo caso deberá promocionarse a *core/model* o *core/vm*, lo que mejor encaje con el equipo.

Para aprender más sobre este tipo de solución, tenemos repositorios y formaciones específicas para esto.

Desventajas de los pods:

- Si el proyecto es pequeño esta solución puede suponer un *overkill*.
- Se fragmentan en muchos ficheros y hay que entender qué hace cada uno.
- Hay que saber medir qué funcionalidad es común / transversal y cual no.
- Otro tema es que muchas veces terminamos con un *pod* por *escena*, o un modelo de api muy parecido al de *vm*, con el trabajo adicional de fontanería que implica (*mappers*, *vm*, ...)

¿Qué pasa si empiezo a tener muchos *pods*? Aquí me puedo plantear crear una carpeta superior de "modules" y agrupar por funcionalidad (ver sección carpeta)

Nombrando funciones, eventos, callbacks...

Nombrar elementos de código, es una tarea complicada y la base para que un desarrollador (o el propio programador una semana después) pueda entender ese código o seguirlo. Para ello es importante que el equipo esté de acuerdo en seguir una serie de reglas.

Nombrando componentes, hooks, funciones

Sobre los nombres de componentes:

- Como estamos con React siempre tienen que empezar por mayúsculas (los que empiezan por minúsculas están reservados para elementos básicos de HTML).
- Se debe estudiar si añadir sufijos, si añadir al nombre "*Container*" o "*Component*", etc.dependiendo de lo que estemos nombrando. Esta decisión no es clara y tiene sus pros y cons:
    - Pros: Cuando usamos un componente sabemos que lo es "*patientComponent*" porque lo tiene en el nombre (por ejemplo no se confunda con la entidad *Patient*).
    - Cons: Es muy pesado arrastrar "*Component*" para cada componente.

Sobre los nombres en entidades (*model* y *vm*), aquí el equipo debe plantear si añadir sufijo entity o vm para distinguir: es buena idea añadir *vm* a una entidad de *Viewmodel*, ya que así evitamos confusiones cuando importamos de forma automática entidades (nos puede traer un nombre con entidad de *api model* y se nos puede complicar darnos cuenta de ese fallo tonto).

Sobre los nombre de *hooks*, aquí seguid las recomendaciones del equipo de Facebook, es decir que empiecen por "*use*" y que sean verbos.

Sobre los nombres de funciones, aquí hay que tener en cuenta que las funciones puras son muy fáciles de testear, por lo que es muy recomendable que sean verbos descriptivos.

Eventos y callbacks

Un tema importante cuando gestionamos eventos es saber de forma fácil qué función maneja un evento en nuestro componente, y cual se burbujea con una *prop*.

Aquí es bueno que el equipo esté de acuerdo en seguir una aproximación consistente.

En nuestro caso proponemos nombrar los manejadores de eventos locales con el prefijo *handle* y los que se burbujean con *props*, con el prefijo *on*, un ejemplo:

```
export const App = () => {
  const [value, setValue] = React.useState("");
  const handleValueChange = (newValue: string) => {
    setValue(newValue);
  };

  return (
```

```
    <div>
      <NameEdit value={value} onValueChange={handleValueChange} />
    </div>
  );
};

interface Props {
  value: string;
  onValueChange: (value: string) => void;
}

export const NameEdit: React.FC<Props> = (props) => {
  const { value, onValueChange } = props;
  const handleInputChange = (e) => {
    onValueChange(e.target.value);
  };

  return <input value={value} onChange={handleInputChange} />;
};
```

Principio de promoción

Hay ocasiones en las que está claro que un componente / función / hook... será reutilizable, en cuyo caso podremos colocarlo directamente en la carpeta que toque.

En otros, dicha funcionalidad seguirá el siguiente camino:

- Empieza siendo implementada, dentro del mismo componente o en el mismo fichero.
- Si vemos que gana peso se puede extraer a un fichero aparte.
- Si vemos que es reutilizable se puede promocionar a la carpeta que toque (*common*, *core*, *common-app*).
- Si este componente demuestra valía y se puede usar en otros proyectos, lo promocionamos a librería.

De esta manera nos ahorramos crear falsos reusables, y los componentes/funciones/hooks se quedan en el nivel que toque.

Política de pruebas unitarias

El tema del *testing* es muy controvertido, aquí el equipo debe llegar a un acuerdo en el que estén cómodos y se pueda llevar a cabo.

En nuestro caso, la decisión que tomamos es:

- Trabajar con componentes UI a nivel de aplicación e incluir pruebas unitarias es complicado, ya que se suelen realizar muchos cambios, y se pierde parte de agilidad (*refactors*, etc...). Si detectamos un componente que sea crítico a nivel de aplicación sí que le añadimos pruebas unitarias, sino, delegamos en e2e.
- Lo bueno es que esos componentes los solemos dejar lo más vacíos posible (*vaciar el cangrejo*), es decir, si aplicamos *Unit Testing* en los siguientes elementos a nivel de aplicación (de ahí la importancia de vaciar los componentes):
    - *Hooks*.
    - Funciones puras y negocio (aquí podemos aplicar TDD)
    - *Mappers* (aquí aplicamos TDD)
    - API.
- También aconsejamos realizar pruebas unitarias de *common* y *core*, ya que es base de código que se va a usar de forma pesada en la aplicación.
- El resto lo cubrimos con e2e *testing* (*cypress*).

Librerías de terceros y frameworks

Librerías

Proceso de elección de librería

Elegir qué librería usar para un proyecto es una decisión muy complicada, normalmente tenemos varias opciones que elegir, y no es fácil saber cuál es la opción más adecuada ya que influyen muchos factores.

Lo primero que hacemos cuando tenemos varias librerías que nos pueden servir para resolver un problema es realizar una **evaluación rápida**:

- Lo primero que evaluamos es la licencia del proyecto ¿Es licencia MIT? Aquí nos podemos encontrar con un problema si esa librería tiene una licencia restrictiva (podría ser que no permitiera el uso comercial o que tuviéramos que publicar nuestro código fuente como open source)
- Uno obvio es el número de estrellas que tiene el proyecto, un proyecto con un buen número de estrellas suele querer decir que es un proyecto que lo ha usado bastante gente y que seguramente encontramos bastante ayuda en *StackOverflow*.
- Otro factor importante ¿Cómo de actualizado está el proyecto? Vamos a ver cuándo se hizo la última *release* y cómo de actualizado está el proyecto, ya que igual en su día fue muy popular, pero ahora se encuentra abandonado (igual no es compatible con la última versión de React, o tiene fallos de seguridad que no se han corregido...).
- Otro tema interesante es comparar el flujo de descargas de *npm trends*, aquí sacamos una gráfica de descargas (por cierto muy divertido ver el bajón de descargas en el periodo navideño), aquí podemos descargar cual es la que más descargas tienes y cómo evoluciona a lo largo del tiempo (que no quiere decir que sea la mejor...)
- Es buena idea fijarnos en el autor y grupo que ha desarrollado la librería:
    - Si es un autor de otras librerías de renombre podemos esperar que la librería tenga cierta calidad.
    - Si pertenece a un grupo de autores y además cuenta con un buen *sponsorship*, podemos esperar que la librería tenga su mantenimiento.
    - Si es de una empresa (*Facebook*, *Microsoft*, *Google*, *Air Bnb*) podemos esperar un código supuestamente de calidad, y en cuanto a mantenimiento depende, puede que mientras le haga falta tenga una evolución perfecta, en cuanto no puede entrar en vía muerta.
- También es importante evaluar el readme y ver que ofrece, igual nos encontramos de primeras que el proyecto ya no tiene mantenimiento oficial.
- Seguimos con temas técnicos, evaluar qué funcionalidad ofrece, y que queremos implementar.
- Es importante ver qué material de aprendizaje tiene:
    - Documentación.
    - Video tutoriales.
    - Posts de terceros.
    - ...
- Otro tema a tener en cuenta es hacer un *audit* de la librería y ver si tiene dependencias anticuadas, etc...

Todo esto nos puede servir para descartar alguna opción (licencia no válida, librería discontinuada), pero son pruebas menores, la prueba real está en probar la librería, en caso de que sea necesidad de un proyecto, pedir tiempo de *spike* y requerimientos que debe cumplir la misma y montar una prueba de concepto para ver hasta dónde llega y que tal se porta.

Librerías que usamos en Lemoncode

A fecha de hoy usamos una serie de librerías en mayor o menor grado, ya que este es un ecosistema volátil, esta lista puede cambiar a corto plazo.

Librería de componentes

Aquí tenemos una apuesta clara: **Material UI** razones por que la usamos:

- Es una librería veterana que se encuentra en un estado maduro.
- Tiene una gran comunidad detrás.
- Es popular.
- Tiene una gran documentación (incluye ejemplos en vivo *codesandboxes*).
- Da buena solución a temas avanzados como la accesibilidad.
- Tiene una gran variedad de componentes.
- Tiene una solución de tematizado muy versátil.
- Tiene un buen grupo de *backers*.
- Tiene una evolución estable (librería core de material ui, y lab para nuevos componentes que cuando están estables de pasan al core)
- Su solución comercial se basa en vender tematizado, soporte, y una parte premium.

En nuestro caso llevamos varios años usando esta librería, y nos ha dado muy buenos resultados, tanto en proyectos internos, como usándola como wrapper para montar librerías de componentes tematizadas (imagen corporativa) para nuestros clientes.

Gestión de formularios

Aquí hemos trabajado con diferentes opciones:

- Tenemos compañeros que prefieren no utilizar librerías de este tipo ya que al final se encuentran con *casos arista* que les hacen tener que implementar *hacks* para que puedan cubrir su caso.
- Hemos trabajado con **React Final Form**: esta librería es muy potente, y cubre muchos casos estándares, pero te puedes encontrar algunos escenarios complicados como hacer que una validación se dispare sólo al cambiar de campo (esto es importante en las asíncronas), y también se puede volver complejo el manejo de validaciones con arrays.
- Otra que hemos cubierto es **Formik**, esta librería es muy parecida a **Final Form**, y le hemos encontrado problemas similares.
- Para finalizar una tercera que estamos probando en proyectos internos a **React hooks forms** la estamos probando en casos avanzados de validaciones con diferentes niveles y arrays, en estos casos hemos tenido problemas de rendimiento y hemos tenido que realizar algunos hacks.

La conclusión actual que tenemos:

- Para formularios estándares, las tres opciones son válidas.
- En el momento en que nos metemos en casos avanzados las tres necesitan de soluciones o *hacks* complejos.
- La que ahora parece que tiene más tracción en la comunidad es **React hook forms**

¿Quiere esto decir que no debo de usar ninguna de ellas? La respuesta es NO, estás librerías suelen resolver muchos problemas, y salvo que los formularios que vayas a utilizar sean muy complejos, es buena

idea utilizarlas, resuelven muchos problemas, y nos ahorran tiempo de desarrollo.

Validación de formularios

En este caso hay varias opciones en el mercado, una de ellas es ***yups***, en su momento evaluamos dicha librería y no nos convenció, decidimos armar la nuestra ***Fonk*** (basada en un desarrollo previo que realizamos **IcValidacionSummary**), razones por las que la usamos:

- Es una librería agnóstica de framework / librería, está desarrollada en JavaScript Plano.
- Pesa menos de 6Kb gzipeada.
- Incluye conectarnos a librerías de gestión de formularios como **Formik** o **React Final Form** (React Hook forms en progreso).
- Está basada en un ecosistema de micro validadores (librerías externas que hemos desarrollado), también permite crear validaciones *custom*, tanto síncronas como asíncronas.
- La hemos rodado en multitud de proyectos tanto internos como externos.

Frameworks

Proceso de elección de framework

Frameworks que usamos en Lemoncode

Herramientas

Aquí cada equipo decide qué herramientas usar y ser consistente con la elección.

El mínimo con el que trabajamos es *Prettier* y el plugin para evitar que se haga un *push* sin haber aplicado *prettier* a los ficheros.

Por otro lado utilizamos la herramienta para gestionar PR del proveedor de *Git* que estamos usando, normalmente *Github*.

Después se pueden añadir herramientas tanto en local y/o *CI/CD*:

- *Linting*: Esto es decisión personal del equipo, a veces puede ser fuente de discusión sobre qué reglas aplicar o no.
- Herramientas para detección de malos olores en el código (por ejemplo *SonarQube*)
- Herramientas para detección de vulnerabilidades y librerías que necesitan actualización (por ejemplo SonarQube).