Lemoncode - Estructura Front

versión: 0.1

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 si 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 (*kebak 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 estos describan que es lo que hacen.

En caso de que hagan 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 este 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 se necesario.

Por ejemplo:

- Tenemos una carpeta `*components*`, de primeras vamos soltando componentes aquí, conforme esta carpeta crece en contenido va haciendo más grande va a hacer falta re factorizarla y agrupar componentes comunes en subcarpetas.
- Tenemos debajo de *components*, un componente de *layout*, otro de que *wrapea* un *input*, un *combobox*, una lista... esos componentes serán candidatos de 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 para las carpetas, ya que suelen agrupar elementos (*components*, *scenes*, *layouts*).

Siguiendo esta aproximación es más natural identificar carpetas, en los ficheros evitaremos el uso de plurales.

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 que funcionalidad queremos publicar como "*pública*", si bien el desarrollador puede ir a por la 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 puede elegir dos aproximaciones:

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

Aquí depende como se enfoque cada submódulo podrían compartir *concerns* comunes.

Otra opción es la siguiente:

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

En este caso cada módulo es una isla independiente, si necesitamos 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 cualquier.
- 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 y 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 (SI tienen relación con el dominio de la aplicación), por ejemplo:

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

Es decir son componentes / funcionalidades reusable 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 que debe incluir.

core

En esta carpeta incluimos la funcionalidad que es transversal 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 puedes costar distinguir que debe entrar en la carpeta *common* y la *core* como regla para distinguirlo:

- La carpeta *common* está más orientadas a componentes y funcionalidades independientes que podemos incrustar en la aplicación, estas funcionalidades actúan como cajas negras independientes.
- La carpeta *core* almacena funcionalidad que se usa y está cohesionada a nivel de aplicación, por ejemplo información de la aplicación que necesitamos consumir a diferentes niveles de la aplicación.

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 que *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.

También la elección de *layout* 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 la versión del mismo (por ejemplo React Router en unas versiones lo permite en 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 capetas 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 / escena?

El objetivo de una escena (página) es:

- Elegir que *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 encontrarlo.

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*, intenta 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 termino, 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

Conforme el proyecto crece y se va haciendo más complejo, se va haciendo necesario añadir más niveles de carpetas, salvo que tengamos claro desde un principio la dirección en la que va a crecer el mismo, es mejor ir añadiendo niveles de carpetas conforme se vaya 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 acaba con una capa que haga de *wrapper* de los componentes comunes que se usen y 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*, esto sería candidato a promocionar a librería.
- Lo mismo pasa con los *hooks* y las validaciones, temas genéricos merece la pena tenerlos localizados (por *ejemplo* una validación de NIF, o de código HL7..., o un *hook* que realiza algo genérico como por ejemplo un hook que nos permite detectar si hemos hecho clic fuera de un elemento.

Esta ruta puede que 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 en 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 a lo sumo.

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

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.

```
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 en *api* crear una carpeta si vamos a tener varios ficheros que monten la api para este *pod*.

En *components* tenemos un caso parecido, 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 hubieran muchos agruparlos a su vez por funcionalidad).

A nivel de ficheros, dependiendo de la complejidad del pod, vamos creando ficheros o rompiendo en carpetas (sobre los ficheros lo cubriremos más adelante), no debemos crear ficheros porque si, sino porque realmente necesitamos agrupar funcionalidad.

El objetivo aquí es que cuando abramos el pod podamos ir rápidamente al punto de entrada e identificar donde tenemos que acudir para tocar algo.

Normalmente un pod no debería de crecer de forma desmesurada, en ese caso 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 *VSCode* nos puede ayudar a hacerlo, pero no siempre es así).
- Productividad: si las herramientas de *VS Code* no me sacan el importa de manera automática, 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 controlado 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": {
+      "@/*": ["*"]
+    }
  },
```

Y para no repetir configuracíon y 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 se nos quedan de esta manera:

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

Nombrando y creando ficheros

Otro tema importante es que convenciones seguimos para nombrar ficheros.

En cuanto a *casing* nos quedamos con siempre usar *lowercase* (minúsculas) para nombrar los ficheros, la razón es la misma que con las carpetas, en Windows los nombres de los ficheros no son *case sensitive*, en *linux* sí, si tu equipo 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 "-" (*kebap-case*).
- Lo que define técnicamente al fichero, lo separamos con puntos "." (*dot-case*).

¿Y está complejidad por qué?

- Por un lado podemos identificar y leer fácilmente lo que define la parte funcional del fichero.
- Por otro pasa lo mismo con la técnica, siempre nos la encontramos en el final del fichero, y además un fichero que tenga varias partes relacionadas va a salir junto (ordenado por nombre).

Por ejemplo:

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

Sobre los sufijos a usar en la parte técnica del fichero, aquí depende de los que el equipo vea oportuno, 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 el peso fuerte de lógica)
- **.business.ts**: para fichero de con funciones puras que resuelvan problemas (también se pueden plantear con estado, 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**: aquí podemos almacenar código para gestionar llamadas a *API's rest*.
- **.mapper.ts**: para convertir de modelo de API a *ViewModel*.
- **.model.ts**: para definir modelos de datos (de API o global), depende del equipo se podría pensar en romper en más niveles de modelo (por ejemplo *api-model*, *domain-model*... un consejo aquí es no meter más complejidad si no es necesario).
- **.validation.ts**: cuando extraemos la lógica de validación de un formulario a un fichero aparte, 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**: Aquí definimos el modelo de la vista, habrá ocasiones en que ese módulo sea un mapeo de uno a uno con lo que nos traemos con la API (sobre todo si los que desarrollan la API son los mismos que desarrollan la aplicación), pero en otros casos puede ser que tengamos que mapear valores y realizar transformaciones para que se ajusten a la vista (esa complejidad la sacamos del UI y la pasamos a una función pura JS, fácil de probar).

Un tema importante a definir es si debemos usar sólo singulares para definir los ficheros o admitimos plurales (en las carpetas lo hacíamos así), en los ficheros, el consejo es que normalmente no (aunque esto depende de lo que decida el equipo), una fuente de fallo común es tener los siguientes ficheros:

```
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 porque no funciona ese código (los fallos tontos son los que más duelen), en vez de esto se pueden usar las siguientes alternativas:

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

ó

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

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

- Si un componente / función, está muy cohesionada a por ejemplo un componente de un fichero, y el mismo tienen un tamaño pequeño, podemos plantear dejar la funcionalidad en el mismo fichero, y más tarde extraerla si vemos que el fichero crece mucho o 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 un componente lo rompemos en subcomponentes.

Estructura de un pod.

La solución de *pods* que usamos 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 funcionalidad *cross/común* (que la 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 que API's va a acceder y que modelo de datos de api va a consumir.
- Definidas que *ViewModels* va a usar.
- Definido su contenedor, components y subcomponentes.
- Definido su módelo y validaciones.

Esto permita 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 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 cosa y sólo una cosa.

Sobre la separación de ficheros, es como comentamos en la sección de "*ficheros*", la idea es separar el *pod* en piezas que hagan una cosa y una sola cosa:

- **.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 el peso fuerte de lógica)
- **.business.ts**: para fichero de con funciones puras que resuelvan problemas (también se pueden plantear con estado, 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**: aquí podemos almacenar código para gestionar llamadas a API's rest.
- **.mapper.ts**: para convertir de modelo de API a *ViewModel*.
- **.model.ts**: para definir modelos de datos (de API o global), depende del equipo se podría pensar en romper en más niveles de modelo (por ejemplo *api-model*, *domain-model*... un consejo aquí es no meter más complejidad si no es necesario).

- **.validation.ts**: cuando extraemos la lógica de validación de un formulario a un fichero aparte, 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**: Aquí definimos el modelo de la vista, habrá ocasiones en que ese módulo sea un mapeo de uno a uno con lo que nos traemos con la API (sobre todo si los que desarrollan la API son los mismos que desarrollan la aplicación), pero en otros casos puede ser que tengamos que mapear valores y realizar transformaciones para que se ajusten a la vista (esa complejidad la sacamos del UI y la pasamos a una función pura JS, fácil de probar).

Un tema muy importante a la hora de definir los *ViewModels*, es que no puedo importar un *ViewModel* de otro *pod*, por muy parecido que sea tengo que volver a crearlo, ¿Por qué? Por no quiero tener acoplamientos entre *pods*, otro tema es un *ViewModel* que se use en todas la aplicación (por ejemplo un *Lookup Id/Valor*), en ese caso lo promociono 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 puede ser un *overkill*.
- Se fragmentan en muchos ficheros y hay que entender que hace cada uno.
- Hay que saber medir que funcionalidad es común / transversal y cual no.
- Otro tema es 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 "module" 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).
- Hay que estudiar si añadir sufijos, si es un contenedor añadir al nombre del componente "*Container*", si es componente "*Componente*", esta decisión no es clara, tienes sus pros y cons:
    - Por un lado cuando usamos un componente sabemos que lo es "*patientComponent*" porque lo tiene en el nombre (por ejemplo no se confunda con la entidad *Patient*).
    - Por otro lado es muy pesado arrastrar "*Component*" para cada componente.

Sobre los nombres en entidades (*model* y *vm*), aquí el equipo debe de 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 (no 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 seguir 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 y que describan.

Eventos y callbacks

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

Aquí es bueno que el equipo este 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 onValueChange={handleValueChange}/>
    </div>
  );
  )
}

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

export const NameEdit : React.FC<Props> = (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... va a ser reutilizable, en ese caso directamente podemos colocarlo en la carpeta que toque.

En otros, dicha funcionalidad sigue el siguiente camino:

- Empieza a ser implementada, puede ser 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 complicad, 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 si le añadimos pruebas unitarias, si no delegamos en e2e.
- Lo bueno es que esos componentes los solemos dejar lo más vacío 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* el resto (*cypress*).

Herramientas

Aquí cada equipo decide que 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 que 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*).