

Arquitectura

Construyendo aplicaciones robustas



La arquitectura

Hace que el arranque de un desarrollo sea más lento

Hace más complicado que un desarrollador se ponga en productivo

Impone restricciones, añade complejidad

Necesita tener perfiles con seniority

Hace que un desarrollo sea robusto

Hace que un desarrollo sea mantenible

Permite tener a varios miembros de un equipo trabajar en paralelo

Permite reaprovechar elementos de un proyecto a otro

¡ Ojo ! No confundir Arquitectura con **SobreArquitectura**

“If you think good architecture is expensive, try bad architecture!”

— Brian Foote & Joseph Yoder

Trabajando mal y rápido

Puedes hacer un churro de desarrollo bien rápido

Lo puedes estabilizar

Puedes salir a producción y que funcione

Puede ser de entrada un éxito

Añadir modificaciones cada vez te va a costar más

Vas a tocar en un sitio y romper en otro

Vas a tener que escalar a base de hierro (en vertical) y eso tiene un límite

Sólo desarrolladores originales del equipo van a ser capaces de mantener ese código

No vas a poder atraer talento a tu equipo

Arquitectura Hexagonal

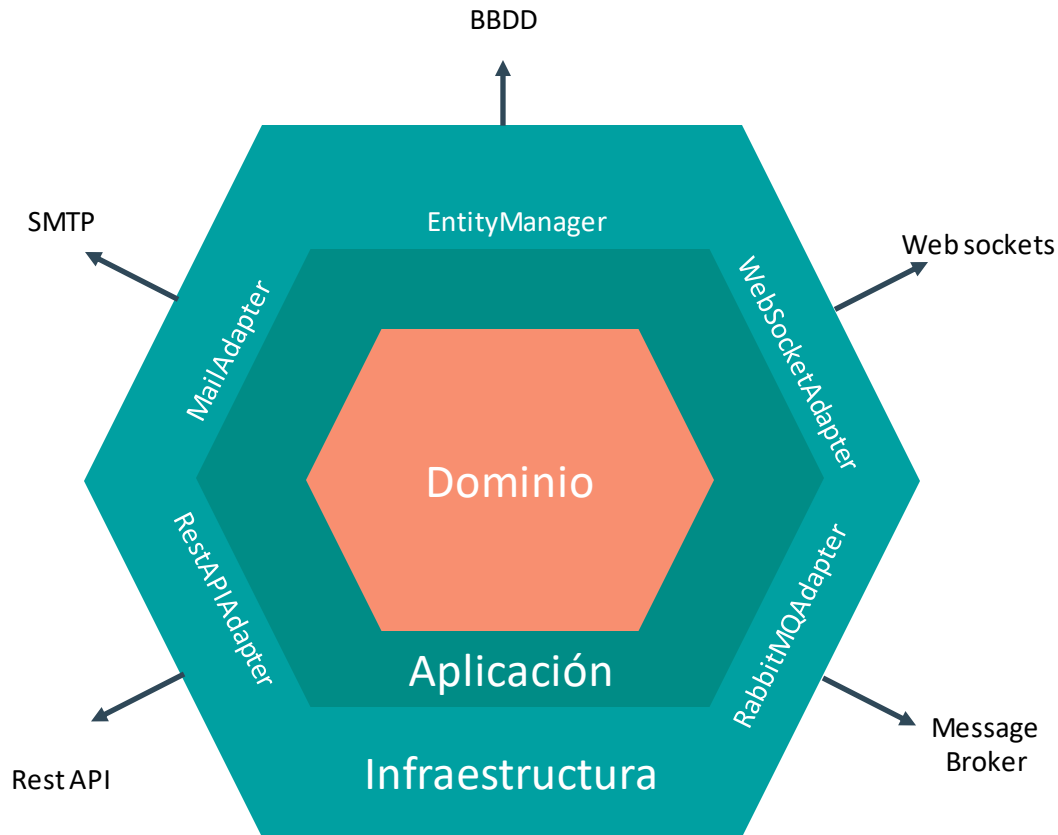
Muy usada en backend.

También se conoce como puertos y adaptadores

El dominio es el nucleo de las capas y no se debe acoplar a nada externo

No se hace uso explicito de los servicios sino siguiendo el principio de IOC nos acoplamos a contratos (interfaces o puertos) y no a implementaciones concretas

Propone que nuestro nucleo sea visto como una API con unos contratos bien especificados. Definiendo puertos o puntos de entrada e interfaces (adaptadores) para que otros modulos (UI, BBDD) puedan implementarlas y comunicarse con la capa de negocio sin que esta deba saber detalles de implementación de su origen



Ejemplo – Envío EMail

Contrato

```
interface EMailService {  
    sendEMail(from : string, to : string, title : string, body : string)  
}
```

Implementación SMTP

```
class SMTPEmailService implements EMailService {  
    public sendEMail(from : string, to : string,  
        title : string, body : string) {  
        MailMessage message = new MailMessage();  
        SmtpClient smtp = new SmtpClient();  
        message.From = new MailAddress(from);  
        message.To.Add(new MailAddress(to));  
        message.Subject = title;  
        message.Body = htmlString;  
        // TODO: read from ENV variable  
        smtp.Port = 587;  
        smtp.Host = "smtp.gmail.com";  
        // (...)  
        smtp.DeliveryMethod = SmtpDeliveryMethod.Network;  
        smtp.Send(message);  
    }  
}
```

Implementación SendGrid

```
class SendGridEmailService implements EMailService {  
    public sendEMail(from : string, to : string,  
        title : string, body : string) {  
        var apiKey = Environment.GetEnvironmentVariable("...");  
        var client = new SendGridClient(apiKey);  
        var from = new EmailAddress(from);  
        var subject = title;  
        var to = new EmailAddress(to);  
        var msg = MailHelper.CreateSingleEmail(from,  
            to,  
            subject,  
            body);  
        var response = await client.SendEmailAsync(msg);  
    }  
}
```

Referencias para saber más

Hexagonal Architecture demystified

<https://madewithlove.com/hexagonal-architecture-demystified/>

Hexagonal Architecture (Alistair Cockburn)

<https://alistair.cockburn.us/hexagonal-architecture/>

Hexagonal Architecture three principles and an implementation example

<https://blog.octo.com/en/hexagonal-architecture-three-principles-and-an-implementation-example/>

Arquitectura hexagonal

<https://medium.com/@edusalguero/arquitectura-hexagonal-59834bb44b7f>

Hexagonal y JavaScript

<https://softwarecrafters.io/react/arquitectura-hexagonal-frontend>

<https://github.com/adrian-afergon/hexagonal-arch-example>

Takeaways para Front

Que una capa interior no conozca detalles de implementación es buena idea

Nos permite retrasar decisiones (puedo arrancar con un fichero como almacenamiento y más adelante pasar a BBDD)

Hace que nuestras piezas sean más fáciles de testear

JavaScript es un lenguaje dinámico, no tiene porque hacernos falta IOC

Pods

Estructurar los proyectos en PODS viene del Framework Web **Ember**.

Cuando estructuramos un proyecto la primera organización que se nos puede venir a la cabeza es por tipo de fichero

Y si... los agrupamos por característica o funcionalidad

De esta manera todos los ficheros relacionados los tenemos agrupados

Podemos ir un paso más allá y usar fronteras para aislarlos (ViewModels + Mappers)

Separando por tipo

La primera opción que se nos viene a la cabeza es la de separar por tipo de elemento



common



components



actions



reducers



rest-api

Pros & Cons

Esta suele ser la primera organización que se nos viene a la cabeza

Puede estar bien para montar un “hola mundo” o un ejemplo pequeño

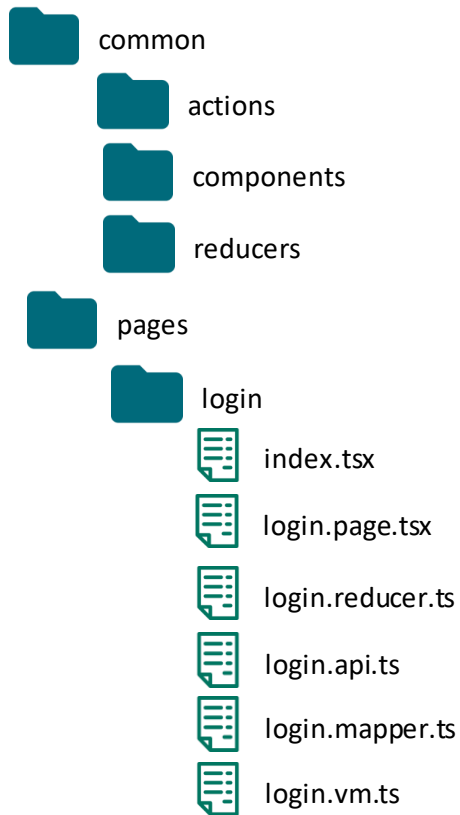
A poco que el proyecto crece la estructura es poco manejable, ficheros relacionados entre si están lejos

Si tienes que llevar funcionalidad a otro sitio no la tienes encapsulada

Hay una mezcla de código local y global

Separando por página

¿ Y si encapsulamos funcionalidad por página?



Pros & Cons

Agrupamos por fichero de funcionalidad común, es más fácil trabajar

Cada página representa una isla en la que el desarrollador puede trabajar de forma aislada

Una página rica se puede dividir en componentes reusables, esto no lo potenciamos aquí

Más que por página debemos pensar en dividir por dominio, un grupo de componentes puede pertenecer a un dominio

Separando por pods (I)

Rompemos por layout, escena, y pod.



common

Aquí van componentes comunes que no están vinculados al dominio (promocionables a librería)



common-app

Aquí van componentes comunes que si están vinculados al dominio (se reusan a nivel de aplicación)



core

Transversales: Rutas navegación, Store, Contexto, Entidades comunes...



layout

Máster Pages, es decir armazones (por ejemplo login, centrar contenido, app, header y footer)



pods

Componentes ricos agrupados por dominio



scenes

Páginas, una escena esta compuesta por un layout y uno o más pods

Separando por pods (II)

Detalle de un pod



pods



login



components



index.ts



login.container.ts



login.container.ts



login.component.ts



login.mapper.ts



login.vm.ts

Pros & Cons

Al agrupar por dominio, es más fácil poder centrarnos a la hora de desarrollar

Un componente rico o varios comparten un pod

Las páginas (escenas) se quedan tontas, elijen layout y pods a utilizar

El proyecto esta preparado para escalar y es más mantenible

Es más fácil extraer y reusar funcionalidad

Requiere hacer un buen trabajo de división en dominios

Para proyectos pequeños puede ser un overkill

Referencias para saber más

Ember pods structure

<https://www.surekhatech.com/blog/pod-structure-in-ember>

How to setup your Project with Ember Pods

<https://www.programwitherik.com/ember-pods/>

Organizing Large React Applications

<https://engineering.kapost.com/2016/01/organizing-large-react-applications/>

How to better organize your React applications?

<https://medium.com/@alexmngn/how-to-better-organize-your-react-applications-2fd3ea1920f1>

Propuesta final

Seguimos la aproximación de esconder implementación, JS es un lenguaje dinámico no tengo porque usar interfaces

Establecemos fronteras con los tipos de modelo (api, vm, state)


Cada POD se puede implementar de forma independiente

Tenemos sitio para tratar temas transversales y comunes

Se aplica el principio de promoción de funcionalidad común POD >> Common >> Lib

¡ Muchas gracias !



 @lemoncoders



<https://github.com/lemoncode>



 @basefactorteam