

Python No Muerde

Capítulo: Las Capas de una Aplicación



Este libro está disponible bajo una licencia CC-by-nc-sa-2.5.

Es decir que usted es libre de:



Copiar, distribuir, exhibir, y ejecutar la obra



Hacer obras derivadas

Bajo las siguientes condiciones:



Atribución — Usted debe atribuir la obra en la forma especificada por el autor o el licenciante.



No Comercial — Usted no puede usar esta obra con fines comerciales.



Compartir Obras Derivadas Igual — Si usted altera, transforma, o crea sobre esta obra, sólo podrá distribuir la obra derivada resultante bajo una licencia idéntica a ésta.

El texto completo de la licencia está en el sitio de [creative commons](https://creativecommons.org/licenses/by-nc-sa/2.5/).

“Que tu mano izquierda no sepa lo que hace tu mano derecha”

Anónimo

En el capítulo anterior cuando estaba mostrando el uso del ORM puse

Si tenemos cuidado y aislamos el ORM del resto de la aplicación, es posible reemplazarlo con otro más adelante (o eliminarlo y “bajar” a SQL o a NoSQL).

¿Qué significa, en ese contexto, “tener cuidado”? Bueno, estoy hablando básicamente de lo que en inglés se llama [multi-tier architecture](#).

Sin entrar en detalles formales, la idea general es decidir un esquema de separación en capas dentro de tu aplicación.

Siguiendo con el ejemplo del ORM: si todo el acceso al ORM está concentrado en una sola clase, entonces para migrar el sistema a NoSQL alcanza con reimplementar esa clase y mantener la misma semántica.

Algunos de los “puntos” clásicos en los que partir la aplicación son: Interfaz/Lógica/Datos y Frontend/Backend.

Por supuesto que esto es un formalismo: Por ejemplo, para una aplicación puede ser que todo twitter.com sea el backend, pero para los que lo crean, twitter.com a su vez está dividido en capas.

Yo no creo en definiciones estrictas, y no me voy a poner a decidir si un método específico pertenece a una capa u otra, normalmente uno puede ser flexible siempre que siga al pie de la letra tres reglas:

Una vez definida que tu arquitectura es en capas “A”/“B”/“C”/“D” (exagerando, normalmente dos o tres capas son suficiente):

- Las capas son una lista ordenada, se usa hacia abajo.

Si estás en la capa “B” usás “C”, no “A”.

- Nunca dejes que un componente se saltee una capa.

Si estás en la capa “A” entonces podés usar las cosas de la capa “B”. “B” usa “C”. “C” usa “D”. Y así. Nunca “A” usa “C”. Eso es joda.

- Tenés que saber en qué capa estás en todo momento.

Apenas dudes “¿estoy en B o en C?” la respuesta correcta es “estás en el horno.”

¿Cómo sabemos en qué capa estamos? Con las siguientes reglas:

1. Si usamos el ORM estamos en la capa datos.
2. Si el método en el que estamos es accesible por el usuario, estamos en la capa de interfaz.
3. Si $\text{not } 1 \text{ and not } 2$ estamos en la capa de lógica.

No es exactamente un ejemplo de formalismo, pero este libro tampoco lo es.

Proyecto

Proyecto

Vamos a hacer un programa dividido en tres capas, interfaz/lógica/datos. Vamos a implementar dos veces cada capa, para demostrar que una separación clara independiza las implementaciones y mejora la claridad conceptual del código.

El Problema

Pensemos en una aplicación de tareas pendientes (el clásico TODO list). ¿Cómo la podríamos describir de forma súper genérica?

- Hay una lista de tareas almacenada en alguna parte (por ejemplo, una base de datos).
- Cada tarea tiene una serie de atributos, por ejemplo, un texto describiéndola, un título, un estado (hecho/pendiente), una fecha límite, etc.

Podríamos asignarle a cada tarea una serie de atributos adicionales como categorías (tags), colores, etc. Por ese motivo es probablemente una buena idea poder asignar datos de forma arbitraria, mas allá de un conjunto predefinido.

- Hay distintas maneras de ver la lista de tareas:
 - Por fecha límite
 - Por categoría
 - Por fecha de último update
 - Por cualquier dato arbitrario que le podamos asignar según mencionamos antes.
- Hay que poder editar esos atributos de alguna forma.

Ahora pensemos en un tablero de [Kanban](#). O pensemos en un sistema de reporte de bugs.

¿Cuál es exactamente la diferencia en la descripción al nivel que usé antes? Bueno, la diferencia principal es cuales datos se asignan por default a cada "tarea". Si tenemos una descripción razonable de cómo debiera ser una tarea, entonces debería ser posible implementar estas cosas compartiendo mucho código.

Entonces dividamos esta teórica aplicación en capas:

Interfaz:

Muestra las tareas/bugs/tarjetas/loquesea y permite editarlas.

Lógica:

Procesa los cambios recibidos via la interfaz, los valida y procesa.

El Problema

Datos:

Luego de que un cambio es validado por la capa de lógica, almacena el estado en alguna parte, de alguna manera. Es responsable de definir exactamente qué datos se esperan y/o aceptan.

Vamos a implementar esta aplicación de una manera... peculiar. Cada capa va a ser implementada dos veces, de maneras lo más distintas posible.

La manera más práctica de implementar estas cosas es de atrás para adelante:

FIXME hacer diagrama

Datos -> Lógica -> Interfaz

Capa de Datos: Diseño e Implementación

Necesitamos describir completamente y de forma genérica todas estas aplicaciones.

Qué tenemos en común:

Elementos

Son objetos que tienen un conjunto de datos. Deben incluir una especificación de cuales campos son requeridos y cuales no, y qué tipo de datos es cada uno.

Ejemplo: una tarea, un bug, una tarjeta.

Campos

Cada uno de los datos que “pertenecen” a un elemento. Tiene un tipo (fecha, texto, color, email, etc). Puede tener una función de validación.

Creo que con esos elementos puedo representar cualquiera de estas aplicaciones.¹

1 La ventaja que tengo al ser el autor del libro es que si no es así vengo, edito la lista, y parece que tengo todo clarísimo desde el principio. No es ese el caso.

Elementos

Estamos hablando de crear objetos y guardarlos en una base de datos. Hablamos de que esos objetos tienen campos de distintos tipos. Si eso no te hace pensar en un **ORM** por favor contáme en que estabas pensando.

Hay montones de ORM disponibles para python. No quiero que este capítulo degenera en una discusión de cuál es mejor, por lo que voy a admitir de entrada que el que vamos a usar no lo es, pero que tengo mis motivos para usarlo:

- Funciona
- Es relativamente simple de usar
- No tiene grandes complejidades escondidas
- Por todo lo anterior: te lo puedo explicar a la pasada

El ORM que vamos a usar se llama **Storm** y ya usamos en el capítulo anterior.

Campos

De hecho, uno podría decir “mi capa de datos es el ORM”, y que toda la definición de campos, etc. es lógica de aplicación, y no sería muy loco. En este ejemplo no voy a hacer eso principalmente para poder presentar una interfaz uniforme en la capa de datos entre dos implementaciones.

Campos

Storm provee [algunos tipos de datos](#) incluyendo fechas, horas, strings, números, y... Pickle. Pickle es interesante porque permite en principio almacenar casi cualquier cosa, mientras no te interese indexar en base a ese campo.

Con un poco de imaginación uno puede guardar cualquier cosa usando Storm y ofrecer una interfaz razonable para su uso. Al intentar tener un diseño *tan* genérico necesitamos algo adicional: necesitamos poder saber qué campos proveemos y de qué tipo es cada uno. Eso se llama introspección.

Diseño

Nuestro plan es crear una aplicación que pueda ser cosas distintas reemplazando pedazos de la misma. Para ello es **fundamental** ser claro al definir la interfaz entre las capas. Si no es completamente explícita, si tiene suposiciones que ignoramos, si no es clara en lo que hace, entonces no vamos a tener capas separadas, vamos a tener un encastramiento en el que se filtran datos de una capa a otra a través de esos huecos en nuestras definiciones.

Por lo tanto, sería útil tener algún mecanismo de especificación de interfaces. Por suerte, lo hay: [Zope.Interface](#)

Primero, no dejes que te asuste el nombre. No vamos a implementar una aplicación Zope. Zope.Interface es una biblioteca para definir interfaces, nomás.

No vamos a incluir acá un tutorial de Zope.Interface, pero creo que el código es bastante claro.

Veamos primero la interfaz que queremos proveer para los elementos.

datos1.py

```
5 # Definiciones de interfaces
6
7
8 class IFieldType(zope.interface.Interface):
```

Campos

```
9
10     """La definición de un tipo de campo."""
11
12     name = zope.interface.Attribute("Nombre del tipo de campo")
13
14     def set_value(v):
15         """Almacenar valor "v" en la instancia del campo."""
16
17     def get_value(v):
18         """Obtener valor de la instancia del campo."""
19
20
21 class IElement(zope.interface.Interface):
22
23     """Un elemento a almacenar, una tarea, etc."""
24
25     def fields():
26         """Una lista de los campos de este elemento."""
27
28     def save():
29         """Guarda este elemento en storage persistente."""
30
31     def remove():
32         """Elimina este elemento del storage."""
33
34 # Fin de definicion de interfaces
```

Algunas aclaraciones con respecto a estas interfaces. Hay un elemento que *no* vamos a implementar de manera abstracta en la capa de datos que debería, en cualquier implementación seria, estar allí: búsquedas.

Normalmente, la interfaz de datos debería proveer algún mecanismo para obtener un subconjunto de los elementos, tal vez ordenados por algún criterio. Lamentablemente, es *muy* difícil implementar eso sin quedar pegados a la implementación del backend.

Vamos a proveer algunos mecanismos con este fin, pero desde ya sepan que son limitados, y hacen que el código sea ineficiente y complicado, comparado con lo que debería ser ².

2

¡Lero lero, es un ejemplo con fines educativos! ¡Esa excusa da para casi todo, che!

Capa de Lógica: Diseño

Capa de Lógica: Diseño

Capa de Interfaz: Diseño

Capa de Interfaz: Diseño