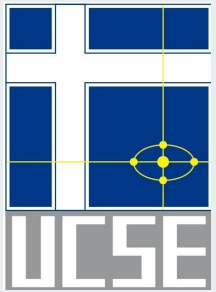
Modularidad de software

UCSE - SEIA



A nivel de funcionalidad

- “DB” de usuarios
- “DB” de contenidos
- Relaciones entre usuarios-contenido
- Relaciones entre usuarios-usuarios

- **Los límites son re borrosos!**

Usuarios

- Identificación (“responsabilidad”, mail, otras redes)
- Privacidad
- Metadata (origen, uso, etc)

Contenido

- Una tabla o muchas? (más de eso en otra unidad)
- Versionado? (más de eso en otra unidad)
- Estado
- Moderación
- Propiedad y visibilidad
- Clasificaciones, categorías, zonas
- Metadata, resúmenes

Usuarios-contenido

- Algunas cosas de lo anterior
- Gusta o no gusta, filtros, preferencias
- Autoría
- Usuarios con interacciones (comentarios, respuestas, votos)
- Quién leyó?
- Interés en ser notificado?

Usuarios-usuarios

- Agrupaciones (grupos, geografía, intereses, amistad, etc)
- Filtros y relaciones individuales (seguir, bloquear, mutear)
- Apertura a interacciones (DMs abiertos? etc)
- Super importantes para que el usuario se sienta cómodo (filtrar info)

A nivel de software

Módulos de código, cada uno con:

- tablas, modelos, db
- código que arma páginas, y sus templates
- código/funciones compartidas (en código, o en la db como SPs)
- documentación

A nivel de software

Cuántos módulos??

No hay una sola respuesta/criterio.

Pero criterio útil: que sea bastante obvio dónde están las cosas.

(“dónde está el html que se ve en tal url”, “si hay un error en la registración, dónde está el código de eso?”, “dónde está la doc del código de foros?”, etc)

Docs de módulos

- Dónde está el código? (+dependencias)
- Big picture: objetivo? alternativas que había? cosas buenas y malas que tiene? qué decisiones importantes se tomaron y por qué?
- Cómo configurarlo
- Cómo usarlo e info que pueda ser necesaria a la hora de hacerle mantenimiento

- **No es suficiente con docstrings. Docstrings necesarios, pero hace falta más.**

Interacciones entre módulos

- Común en otros tiempos: stored procedures. Hoy no tan buena idea.
- Hoy: Apis! (web o libs, funciones “públicas”)
  - No repetimos lógica en todos lados, unificada
  - Fácil saber cuando un cambio rompe
  - Fácil reutilizar, no hace falta conocer detalles de implementación
  - Fácil de cambiar la implementación
  - Definida, documentada, versionada!
- Microservicios?

Configs de módulos

- No hardcodear, el módulo se puede usar en distintos “entornos”
- En la DB? Problema: cantidad de consultas en las requests.
- Los frameworks tienen mejores opciones.

Separando programadores y diseñadores

Problemas:

- El diseño va a cambiar mucho más que la lógica, mejor que estén poco atados
- Los diseñadores y programadores tienen skills diferentes

Separando programadores y diseñadores

Opciones viejas, malas:

- una url -> un archivo con todo lo que se hace y ve en la url (ej: PHP) (super frágil! y difícil cambiar globalmente)
- HTML en un dir, código en otro dir, relación 1-1 entre código y HTML (sigue siendo difícil cambiar globalmente)
- Lo mismo, pero guardando el HTML en la db (mala idea, falta versionado, fácil romper!)

Separando programadores y diseñadores

La solución moderna: un sistema de templates!

- Separación lógica vs presentación
- Robustez (presentación no rompe backend)
- Mecanismos de herencia, reutilización, etc, entre templates
- Se hace fácil cambiar globalmente!
- Usualmente no en la DB. Templates versionados, como todo código.