

## Coding Best Practices (WIP!)

### General

- Aprender bien git, es una herramienta poderosísima que van a usar 100 veces por día. No hace falta entender la funcionalidad avanzada, pero sí entender claramente los conceptos básicos y los comandos que usan a diario. En caso de duda ("se rompió algo y quieren recuperar X versión"), consultar a alguien con más experiencia.
- Las herramientas de gestión de proyectos en especial las que se adaptan a metodologías ágiles (Trello, Jira, Ora, y muchísimos etc.) son casi vitales para hacer el seguimiento de las tareas a realizar y hechas, tener en claro los alcances de (y discutir mucho) cada card/task/tarea, y organizar al equipo. Bien usada, no se debería empezar ninguna tarea sin pasar por ahí para que esté todo el equipo alineado. El uso que se le da depende mucho del proyecto específico, así que encararlo temprano y hablarlo bien con el líder de proyecto.
- Divisiones y subdivisiones de tareas. Aun si el formato justo depende de lo preferido por el equipo, siempre dividir las tareas a realizar en lo más pequeñas y manejables que se puedan encarar sin miedo. Una buena regla mnemotécnica es la de INVEST para stories, que también se puede aplicar para otros tipos de tareas.
- Codear apurado por la ansiedad es peor que pensar bien y en grupo qué hacer antes de lanzarse al IDE. La urgencia injustificada termina en desarrollos que hacen algo distinto a lo que se buscaba en el peor de los casos, y en diseños inextensibles o inmantenibles en el mejor de ellos. Incluso en proyectos de unas pocas semanas, se recomienda destinar unos días al diseño y a planear bien el proyecto antes de escribir la primera línea.

### PRs y feature branches

Trabajamos con **feature branch + PR** hacia el branch principal.

El Pull Request (PR) es una herramienta simple pero muy útil para agrupar commits en un cambio incremental pero significativo al repositorio. Tiene que ser relativamente chico, idealmente concentrado en una sección, e idealmente enfocado en 1 cosa. ("*Do one thing and do it well*"). En la descripción: mencionar brevemente qué se hizo, se pueden incluir capturas de los cambios en el caso de UI (¡sugerimos

hacerlo!), se puede mencionar qué tickets cierra o avanza (o simplemente alguno relacionado), se listan cosas a tener en cuenta y limitaciones del PR.

Si quedan dudas sobre el workflow hay muy buenos artículos sobre "feature branches" online.

// TODO: incluir alguno específico?

## Commits y branches

Los commits en sí hacerlos chicos y específicos. Si tienen que tocar varias cosas, vean de atacar una cosa a la vez y en commits separados. Si tienen que tocar varias secciones, consideren hacerlo en distintos PRs. Una práctica que tomamos algunos es revisar bien y elegir explícitamente qué archivos estamos metiendo en un commit (menos `git add .`), no solo para rechequear lo hecho, sino para dividir en commits específicos, dejar el mensaje justo de lo que están modificando.

El commit tiene un título (el *commit message*). Recomendamos poner mensajes informativos, concisos y claros (esto es algo a lo que le van agarrando la mano, como cualquier otra cosa). Lo mejor es mantener un formato común en el grupo. En menos de 80 caracteres quieren decir qué área tocaron, si es algo nuevo o un fix o un test etc., y brevemente qué cambiaron. Es bastante popular usar una variación de este que usan en Angular. Algunos ejemplos posibles:

- `feat(api/players route): add endpoint GET /players/:id`
- `refactor(client/dashboard): date logic for product expirations`
- `fix(api/users controller): return 404 on inexistent user`
- `test(api/math): add missing tests for division by zero`

Como recomendación extra, **no alterar** el mensaje de commit de los merge, dejar el que se genera automáticamente.


Poner nombres informativos a los branches (ramas). Esto suele ser más al punto y general, pero todavía informar sobre qué sección se trabaja y qué tipo de branch es. El formato suele ser distinto, y usar solo `a-z0-9_-/.` . Algunos ejemplos:

`feat/user_creation`, `fix/currency_with_decimal`, `chore/lint_ci`, `refactor/auth_routing`.

## Code review


Recomendamos *fuertemente* hacer **code review** en cada PR. Avisar al reviewer cuando un PR está listo para revisar (taguear en Ora o en PR). Si el PR se crea antes de estar listo (porque es útil para conversar y ver los cambios hasta ahora) poner al

principio del título "WIP: ", y sacarlo cuando el PR se considere feature-complete, testeado y estable. (Ver "Definition of Done".) Particularmente a mí me gusta hacer refactors y clean-ups antes de considerar un PR como listo. Una vez que está aprobado, se recomienda que el autor del PR sea quien lo mergee.

De nuevo, los empujamos a tomarse en serio hacer CR; en el feedback hemos recibido comentarios de que fue el cambio de proceso más importante que han tomado, y el más valioso. A continuación el Henry  Tomás Mercado explyándose sobre las ventajas de hacer CR:

Tiempo ganado en no repetir código que ya hicieron otros, tiempo ganado a la hora de resolver conflictos al mergear por saber qué hicieron otros, la posibilidad de aprender viendo cómo resolvieron algún problema, y la chance que da de que todo el equipo se sienta más cómodo con el código, etc.

Una de las cosas que más me sirve del CR es que mientras más lo pongo en práctica, más me esfuerzo en pensar en mis compañeros a la hora de codear, para que sea lo más entendible posible. Me parece que ayuda enfocarlo desde ese lado: ¿cómo me gustaría que sea un PR que yo lea? ¿qué tan complicado sería de leer lo que estoy haciendo? ¿cuánto tiempo voy a sacarle a mis compañeros con este CR?

Una idea extra sobre esto: el code review no supone entender hasta el último detalle cada línea de código ajena pero, además de agregar una etapa de validación del trabajo hecho, sí incentiva a que tengamos una visión global del proyecto; queremos tener una idea clara de las partes que lo componen y cómo se relacionan. Lucas Verdiell, otro Henry , dice:

Creo que es importante no encerrarse 100% en lo que uno esta haciendo como si fuera lo único importante y, al menos, entender que es lo que hace el resto del equipo. Creo que tener una visión holística con cierto nivel de abstracción en los detalles es muy positivo.

**Main branch.** El branch principal (master o main, generalmente) **debe ser estable.**

Para mergear hacia master/main hay que estar suficientemente confiado de que va a andar todo y no va a haber regresiones (romperse cosas que ya andaban). En caso de tener cambios muy grandes por delante se sugiere hacer un branch aparte (una especie de "epic branch") y desde el cual se pueden hacer feature branches más chicos sobre los cuales trabajar, finalmente mergeando la épica hacia master/main una vez que se termina.

## Código

Código legible siempre. El código se lee muchas más veces de lo que se escribe.

- Usar una buena estructura de carpetas y ser ordenados. Para esto, la recomendación general es partir de una buena plantilla que ya organice los archivos como se quiera (y a esto se le puede hacer pequeños cambios). Por ejemplo, para el que probé `express-generator` habrá visto que ya crea carpetas para rutas y views. Create React App por otro lado crea aun menos estructura para el código, solo la carpeta `src/`. Recomendamos construir sobre estos organizando bien dónde va a ir cada cosa para que esté todo mucho más ordenado y consistente más adelante. Un buen template también ayuda a que se pueda arrancar más rápido a trabajar en un proyecto. Hay varias muy buenas dando vueltas, y herramientas para scaffoldear como por ejemplo yeoman.
- Hoy en día "todo está hecho" (bueno... no, pero mucho mucho sí!). El 99 % del tiempo no conviene intentar reinventar la rueda: usemos las herramientas / librerías / servicios ya existentes para hacer apps más poderosas en menos tiempo! En líneas generales nos vamos a dar cuenta de que si un problema es común entonces *ya fue resuelto* y **mucho mejor que nosotros lo haremos**. Esto no significa que confiemos en cualquier cosa que nos crucemos (guarda con código no actualizado hace mucho, sin mantenimiento, con muchos issues abiertos, versión muy alfa (0.0.5), mal documentado...), y a veces conviene simplemente sacarle ideas a cosas ya armadas pero no dependibles.
- Reutilizar funciones y componentes lo más que se pueda. Si algo es muy similar a lo que se usa en otra sección similar, pensar que se puede abstraer para que ambas secciones usen una versión generalista de lo mismo.
- En todos los lenguajes hay convenciones de sintaxis y de estilos que hacen que el código sea mucho más amigable para leer (por ejemplo para Javascript la de airbnb es una base de facto ya). Recordemos que **el código se lee muchas más veces de las que se lo escribe**. Prender el linter automático (eslint) en el IDE que usemos (e.g. VS Code, Atom) y darle bola. Se pueden usar herramientas como eslint o prettier para corregir automáticamente el estilo.

## Nombrando

La elección de nombres para conceptos básicos de la lógica de negocio parece tonto pero es algo que se arrastra por el resto del desarrollo. ¡Pensarlos bien! ¡Decidirlos entre todos!

**Usar el nombre justo.** Nombrar una variable, función o constante tiene que ser una tarea creativa. No usar lo primero que se les ocurre y no descarten cambiar su nombre si dejó de representar lo que verdaderamente es. Usar el nombre justo implica que el nombre de la variable permita entender específicamente qué contiene dicha variable a cualquiera que esté mínimamente familiarizado con el código del proyecto. Un ejemplo: `players` sería una lista de jugadores, pero si hay una versión filtrada entonces se puede considerar `filteredPlayers`. Tampoco son siempre las mismas reglas al nombrar una función o método, ya que esta suele implicar una acción: `filterPlayers`. Buscar el balance entre corto y claro. Priorizar claridad sobre minimizar la cantidad de caracteres. Tiene que ser tan obvio que se entienda de solo leerlo.

Seguir las convenciones de casing (minúsculas y mayúsculas): `playersStatistics`, `getVideoURL` (variables y funciones, *camelCase*), `MATCH_LENGTH` (constantes), `PlayerDashboard` (clases y componentes de React), `/players/:id/basic_info` (URLs), etc.

Un buen nombre evita la necesidad de comentarios

## Otros

Esto cambia de lugar a lugar pero recomendamos minimizar la cantidad de comentarios en el código: el código tiene que poder entenderse lo máximo posible sin comentarios. Muchas veces un comentario puede reemplazarse extrayendo algunas líneas a una función, otras con un mejor nombre de variable. Si todavía vemos beneficioso dejar un comentario, que sea claro y directo y esté bien cerca de lo que aclara.

También minimizar lo más posible los string literals, lo que se suele llamar "hardcodeo". Muchas veces eso debería ser una constante en el lugar apropiado e importarse desde todos los lugares donde se lo necesite.

// TODO: MUY ABSTRACTO ESTO QUE SIGUE? // Extraer lógica abstracta en funciones de ayuda (*helper functions*). E.g. para saludar al usuario queremos el día de hoy en un string humano "21 de diciembre de 2021", pues entonces que exista

`getHumanDateForToday()` y, como su lógica no tiene que ver con el componente actual, considerar ponerla donde mejor pertenezca, como por ejemplo un archivo o carpeta de `utils`.

Por último pero para nada menos importante, dejemos claro que el **inglés ya ganó la pelea por internet**, al menos en lo que respecta a codear. Lo mejor es seguir esta corriente para que la comunicación sea consistente y no tenga nada fuera de ASCII. Por esto, es buena práctica que **todo en el código, incluyendo clases, variables, constantes, ¡e incluso comentarios! sea en inglés**. Esto hace que la lógica de negocio sea consistente y haya menos ambigüedades, porque siempre es en el mismo idioma. Y también recomiendo fuertemente hacerlo para commits, PRs y branches. Por otro lado, READMEs, conversaciones en PRs, Wiki, etc está bien si es en otro idioma. En el código solo strings que termina leyendo el usuario deberían estar en otro lenguaje (y hasta esto lo reducimos con `i18n`).

## Testing

**Testear hace nuestras vidas más fáciles.** Todo código sin tests no da ninguna garantía, y por defecto se puede considerar inestable. Estas palabras pueden parecer generalizadoras al principio pero se vuelven más obvias a medida que ganan más experiencia.

Es muchísimo, muchísimo mejor tener tests que se corren automáticamente comprobando en todo momento que la funcionalidad desarrollada sigue cumpliendo las expectativas. Depender de pruebas manuales (con, por ejemplo, Postman) hace a un desarrollo lento, repetitivo, cansador y repetitivo. El tiempo ahorrado a la larga es inmenso.

Dicho esto, ¡cuidado! con escribir tests que *no* testeen específica y correctamente lo que *dicen* testear, porque ¡tests que dan confianza no merecida son **peores que ningún test!**


También mencionamos Test-Driven Development (TDD) como una forma de trabajo interesante que algunos les gusta mucho. Considerarla.

## Continuous integration

Sugerimos usar herramientas de CI (como GitHub Actions) para que lo hecho por cada integrante se valide constantemente.

Tal como algunas veces al principio se quejarán "El CI de @\$#% no me deja mergear!", muchas más veces van a agradecerle por sacar a luz un bug nuevo en el momento justo, así que tratarlo como amigo. ¡No ignorarlo!

## Yapa

- Es una muy buena práctica (¡y muy agradecida!) incluir un README en cada proyecto (y subproyecto) resumiendo muy brevemente: nombre del (sub)proyecto, las partes, los pasos a seguir para tener todo corriendo (e.g. ¿archivo .env?), los comandos típicos que se usan en el día a día para compilar/correr/etc., qué prácticas siguen en ese proyecto ("Usamos prettier, ..."), links a documentación y/o más info, reglas del grupo para contribuir, y otras cosas que sean útiles para el que llega al repo. Lo más importante son los primeros 4, y aunque van a ver READMEs muy diversos, ¡ciertas cosas se repiten mucho por una razón!
- Aprender a dividir los problemas ("divide y vencerás") y a abstraerse de las capas contiguas es una de las lecciones más poderosas en la programación y, también, algo en lo que se mejora lentamente.
- Sobre comunicación y foco, por Leandro Álvarez :

Comunicación: Es crucial saber expresarse y comunicarse de forma óptima.

Aprovechar el tiempo de las dailys y dejar constancia de lo importante en las vías que correspondan (Ora, Trello, GitHub). Saber utilizar cada uno de estos espacios de forma apropiada es algo que se aprende con la práctica.

Foco: Entender que tareas son prioritarias respecto de otras y simplificar las ideas a la hora de explicarlas permite que mis compañeros entiendan de verdad lo que quiero comunicar, antes de perder el hilo de la comunicación. Muchas veces creemos que nos estamos entendiendo y cada uno tiene un universo diferente en su cabeza.

- Entender bien el modelo MVC (modelo-vista-controlador). ¡Es muy importante en el backend de los proyectos que solemos encarar!
- Ciertas herramientas conviene consensuarlas en grupo antes de arrancar para que todos usen la misma. Por ejemplo, algunos prefieren `npm` y otros `yarn`. Mezclar herramientas o incluso versiones de la misma es problemático ya que cada persona tiene un ambiente distinto y puede tener problemas que los demás no, así que elegir en equipo y mantener uno. (En este caso específico si algún integrante quiere usar otro package manager, que no pushee los lockfiles.)

- Manejo de la frustración. Codear es muchas veces una actividad frustrante. Queremos que el código hace "esto" pero parece que se niega a entender lo que le decimos. Aprendemos frameworks y herramientas y paquetes y ¡lenguajes enteros! muy seguido, lo cual nos mantiene siempre como estudiantes, con toda la frustración que esto conlleva. Es muy importante tomarse la calma de entender que esto es normal como también de buscar las muchas maneras que hay de minimizar la frustración. Desde herramientas que nos facilitan la vida a... ¡buenas prácticas como estas! Aprovechar el equipo para compartir sus penas como también sus victorias.

Maintainer: @MartinCura