Gestión del Código Fuente

Desde el grupo de piripi-hub se han acordado los siguientes aspectos en cuanto a la gestión del código fuente del proyecto para así trabajar de manera homogénea y ordenada y garantizar la máxima calidad:

- **Idioma:** tanto para el código desarrollado, los commits, las ramas, las issues y cualquier descripción se utilizará el inglés para su elaboración.

- **Política de versionado:** se divide en los siguientes bloques:
  - ➢ **MAJOR:** número de versión que se incrementa cuando se rompe la compatibilidad de versiones anteriores.
  - ➢ **MINOR:** número de versión que se incrementa cuando se añade funcionalidad y esta es compatible en la versión MAJOR actual.
  - ➢ **PATCH:** número de versión que se incrementa cuando se arreglan errores en la versión MAJOR.MINOR actual.

  De esta forma, se crean las tags en el proyecto y a partir de estas, las releases.
  Se espera que al menos se creen dos releases a lo largo del proyecto.

  Ej de versionado:

```
// MAJOR.MINOR.PATCH

2.12.7
// 2 -> MAJOR
// 12 -> MINOR
// 7 -> PATCH
```

- **Política de commits:** se utilizará Conventional Commits, una convención en el formato de los mensajes de los commits para hacerlos más legibles. De esta forma, conseguimos un acuerdo en el formato de los commits con todo el equipo de desarrollo y armonía en el historial del repositorio.
  El mensaje de commit debe ser estructurado de la siguiente manera:

```
<tipo>(ámbito opcional): <descripción>
<LINEA_EN_BLANCO>
[cuerpo opcional]
<LINEA_EN_BLANCO>
[nota(s) al pie opcional(es)]
```

  Tipo

  El primer elemento es el tipo de commit refiriéndose al contenido del commit.
  Basados en la convención establecida por Angular, estos son:

- **feat:** cuando se añade una nueva funcionalidad.
- **fix:** cuando se arregla un error.
- **chore:** tareas rutinarias que no sean específicas de una feature o un error como por ejemplo añadir contenido al fichero `.gitignore` o instalar una dependencia.
- **test:** si añadimos o modificamos tests.
- **docs:** cuando solo se modifica documentación.
- **build:** cuando el cambio afecta al compilado del proyecto.
- **ci:** el cambio afecta a ficheros de configuración y scripts relacionados con la integración continua.
- **style:** cambios de legibilidad o formateo de código que no afecta a funcionalidad.
- **refactor:** cambio de código que no corrige errores ni añade funcionalidad, pero mejora el código.
- **perf:** usado para mejoras de rendimiento.
- **revert:** si el commit revierte un commit anterior. Debería indicarse el hash del commit que se revierte.

Ámbito

El campo ámbito es opcional y sirve para dar información contextual como por ejemplo indicar el nombre de la feature a la que afecta el commit.

Descripción

Breve descripción del cambio cumpliendo lo siguiente:

- Uso de imperativos, en tiempo presente: “add” mejor que “added”.
- Primera letra siempre en minúscula.
- No escribir un punto al final.

Cuerpo

Es opcional y debería aportar más información que la descripción, además de usar un tono imperativo.

Nota al pie

Es opcional. Siempre se utiliza para indicar cambios que rompan la compatibilidad de la versión actual (Breaking Changes).

Si queremos poner en el pie del commit un Breaking Change, el formato deberá ser el siguiente:

```
BREAKING CHANGE: <description>
```

Aclaración

En cuanto a lo anterior mencionado de política de versionado:

- ➢ Si el tipo del commit es **fix** indica que el commit es un parche de un error y está relacionado con la versión *PATCH* del proyecto.
- ➢ Si el tipo es **feat** indica que el commit añade una nueva funcionalidad y se relaciona con la versión *MINOR* del proyecto.
- ➢ Añadir el texto **BREAKING CHANGE** en el pie de un commit, o el carácter **!** después del tipo, indica que se rompe la compatibilidad de la versión actual y está relacionado con la versión *MAJOR* del proyecto.

Ej de commit:
feat: add download all service

- **Política de ramas:** en nuestro caso usaremos un modelo de ramas por features, quedando la estructura de la siguiente manera:

  - ➢ **Rama main** (producción): esta rama debe contener siempre el código que esté listo para desplegar en producción. Solo se fusionan a main los cambios que hayan pasado por todas las pruebas y revisiones necesarias.
  - ➢ **Rama develop** (preproducción): esta rama representa el entorno más cercano a producción. Los cambios deben pasar primero por aquí para hacer pruebas de integración y pruebas finales antes de desplegar a producción.
  - ➢ **Ramas feature** (funcionalidades): cada nueva funcionalidad que esté en desarrollo debe tener su propia rama, normalmente creada a partir de develop. Cuando el desarrollo de la funcionalidad está completo y pasa las pruebas locales, se podrá fusionar en develop.
  - ➢ **Rama incidents**: en esta rama se desarrollarán las correcciones a los problemas encontrados, especialmente orientado a aquellos problemas que se detecten y que en un primer momento no se sepan asociar a una rama feature concreta.

    Formato de nombrado (camelCase): WIx-NombreWorkItem
    donde x es el número de work item definido en el grupo.
    Ej: WI1-SearchQueries

- **Política de pull requests:** una vez terminados los cambios en una rama feature, para poder integrarlos en la rama develop habrá que crear una pull request que deberá ser aprobada y aceptada añadiendo un comentario positivo por al menos un miembro del equipo diferente al que desarrolló la funcionalidad. En caso de no aprobar los cambios deberá ser rechazada la pull request añadiendo un comentario explicando el motivo.

  Para poder integrar los cambios desde develop a main, se procederá de la misma forma, pero al menos dos personas tendrán que comentar positivamente para que se apruebe y acepten los cambios.

  - ❖ Leer detenidamente el documento “Plantilla pr”.