Al Filo del Éxito

Proyecto para el desarrollo de la Pagina Web para la Gestión de Acciones Estándares

Versión 1.0

Historial de Revisiones

Fecha	Versión	Descripción	Autor
20/11/2024	1.0	Definición de los Estándares Utilizados	Grupo Al filo del Éxito: Defaz Joel Haro Rodrigo Mendosa Miguel Trocellier Michael

Tabla de Contenidos

1 Introducción

Este documento define los estándares y lineamientos a seguir en el desarrollo del proyecto de una plataforma web destinada a la gestión y compra de acciones en el mercado bursátil. El objetivo principal es asegurar la calidad, consistencia y eficiencia en todas las etapas del desarrollo, así como la interoperabilidad entre los equipos involucrados.

1.1 Propósito del documento

Establecer un conjunto claro de estándares para guiar el diseño, desarrollo, implementación y mantenimiento de la plataforma.
Facilitar la colaboración entre los equipos de desarrollo, diseño y negocios.
Asegurar que el producto final cumpla con las expectativas del cliente, las normativas regulatorias aplicables y los requisitos de seguridad específicos del mercado bursátil.

1.2 Alcance

Este documento aplica a todas las fases del desarrollo de la plataforma web, incluyendo:

- Diseño de interfaces de usuario (UI) y experiencia de usuario (UX).
- Desarrollo del back-end para la gestión de datos y operaciones financieras.
- Implementación del front-end para interacción con los usuarios.
- Integración con servicios externos, como APIs de mercados bursátiles.
- Aseguramiento de la calidad mediante pruebas unitarias, de integración y de rendimiento.
- Mantenimiento y futuras actualizaciones del sistema.

No se cubrirán detalles específicos sobre estrategias de marketing o aspectos comerciales del negocio, salvo cuando estén directamente relacionados con las funcionalidades de la plataforma.

1.3 Audiencia objetivo

El siguiente documento va dirigido a:

- **Equipo de desarrollo:** desarrolladores back-end, front-end, y DevOps que trabajarán en la implementación de la plataforma.
- **Clientes y/o Stakeholders:** quienes necesitan una comprensión general de los estándares que se seguirán en el desarrollo del sistema.

1.4 Referencias

[1] Django, “Django documentation | Django documentation,” Django Project, 2024. https://docs.djangoproject.com/en/5.1/

[2] A. Sharma, “Docstrings in Python Tutorial,” Datacamp.com, Apr. 10, 2020. https://www.datacamp.com/tutorial/docstrings-python (accessed Nov. 23, 2024).

[3] “Redirecting,” Sharepoint.com, 2024. https://epnecuador-my.sharepoint.com/:w:/g/personal/jose_defaz_epn_edu_ec/EaPFOmwU1sFOl5Sjoxm0_fgBuMc2vpc9L7IdfNOVyP3nLw?e=tXCcuo (accessed Nov. 23, 2024).

[4] “Conventional Commits,” *Conventional Commits*. https://www.conventionalcommits.org/en/v1.0.0/

2 Estándares

2.1 Lenguajes de programación utilizados

El proyecto será desarrollado utilizando los siguientes lenguajes de programación:

- **Python**
  Utilizado como el lenguaje principal para el desarrollo del back-end a través del framework Django.
  Permite manejar la lógica del negocio, realizar integraciones con bases de datos, y gestionar servicios relacionados con las operaciones financieras de la plataforma.
- **JavaScript**
  Utilizado para el desarrollo del front-end, garantizando una experiencia interactiva y dinámica para los usuarios.
- **HTML5 y CSS3**
  Utilizados para la estructura y el diseño de las páginas web.
  Se complementarán con herramientas como Bootstrap o TailwindCSS para asegurar un diseño responsivo y accesible.
- **SQL**
  Utilizado para la gestión de la base de datos en el back-end.
  Django ORM proporcionará una capa de abstracción para manejar las consultas y operaciones sobre la base de datos.

2.1.1 *Formato de código:*

Aparte de los estándares que se conocen en la documentación de Django [1], se ha de tomar en cuenta los siguientes estándares para Python.

- Estilo de nombres:
  - snake_case: para nombres de variables y funciones en Python (cumpliendo con PEP 8).
    Ejemplo: calcular_ingresos().
  - PascalCase: para nombres de clases.
    Ejemplo: GestionDeAcciones.
  - UPPERCASE_SNAKE_CASE: para constantes globales.
    Ejemplo: API_KEY = "abc123".

- Límites de longitud de línea:
  Se limitará a 80 caracteres por línea, según las recomendaciones de PEP 8 para Python. En casos excepcionales, se permitirá hasta 120 caracteres, pero solo si mejora la legibilidad.

- Indentación:
  Se usarán 4 espacios para la indentación (no tabulaciones), conforme a PEP 8.
  Todos los archivos deben ser configurados para mantener este estándar de forma automática

2.1.2 *Comentarios*

En este proyecto, se prioriza la escritura de código auto-documentado, donde los nombres de clases, métodos, variables y funciones sean lo suficientemente descriptivos para que el propósito del código sea evidente sin necesidad de comentarios.

Directrices
Los comentarios se usarán únicamente cuando:

- Se realice una lógica compleja que no pueda ser entendida fácilmente leyendo el código.

- Sea necesario aclarar un comportamiento inesperado o un ajuste temporal.
- Se explique una decisión técnica o comercial que no sea evidente.

Nombres descriptivos:

Los nombres de métodos, variables y clases deben reflejar con precisión su propósito.
Ejemplo:
En lugar de calc(), usar calcular_balance_financiero().
En lugar de x, usar total_acciones.

Formato de comentarios (cuando sean necesarios):

- Los comentarios seguirán una estructura simple y concisa.
- Se usará el formato de docstrings [2] únicamente para métodos, funciones o clases cuya lógica no sea autoevidente.
  Ejemplo de docstring en un método complejo:

```python
class Transaccion:
    def calcular_dividendo(self, acciones, tasa):
        """
        Calcula el dividendo de un usuario en función de las acciones y la tasa.

        Este método usa una fórmula especial debido a una regla del mercado
        que aplica una tasa compuesta en lugar de simple.

        Args:
            acciones (int): Número de acciones compradas.
            tasa (float): Tasa de interés aplicable.

        Returns:
            float: Valor del dividendo calculado.
        """
        return acciones * ((1 + tasa / 100) ** 2 - 1)
```

- Evitar comentarios redundantes:
  No se deben escribir comentarios que repitan lo que ya se entiende del código.
  Ejemplo incorrecto:

```
# Calcula el total de acciones disponibles.
total = acciones_disponibles()
```

  Ejemplo correcto:

```
total_acciones = acciones_disponibles()
```

- Comentarios temporales:
  Los comentarios que expliquen soluciones temporales deben ir acompañados de un TODO para indicar su necesidad de revisión futura.
  Ejemplo:

```
# TODO: Reemplazar este algoritmo con uno más eficiente.
```

2.2 Gestión de Versiones

2.2.1 *Sistema de control de versiones:*

- Herramienta utilizada:
  - Git: Se utilizará Git como sistema de control de versiones distribuido debido a su robustez, popularidad y capacidad de manejo eficiente en proyectos colaborativos.
  - Repositorio remoto: GitHub para almacenamiento remoto, revisión de código y colaboración en equipo.

2.2.2 *Convención de ramas:*

Para la descripción del nombre de las ramas, aparte de las que ya se han definido en el flujo de trabajo [3] y son siempre las mismas (main, develop, docs) ni tampoco las ramas de cada módulo, se tomaran en cuenta las siguientes directrices

- Usar nombres descriptivos y consistentes con el prefijo adecuado:
  - feature/nombre:
    Para nuevas funcionalidades (equivalente a feat en los commits).
    Ejemplo de rama:
    feature/agregar-carrito-compra
  - bugfix/nombre:
    Para correcciones de errores (equivalente a fix en los commits).
    Ejemplo de rama:
    bugfix/solucionar-carga-inicial
  - style/nombre:
    Para cambios estéticos o relacionados con el formato que no afectan la lógica del código (equivalente a style en los commits).
    Ejemplo de rama:
    style/ajustar-identacion-css
  - refactor/nombre:
    Para modificaciones que mejoran la estructura del código sin cambiar su comportamiento (equivalente a refactor en los commits).
    Ejemplo de rama:
    refactor/reorganizar-modulo-usuarios
  - test/nombre:
    Para agregar o modificar pruebas automatizadas (equivalente a test en los commits).
    Ejemplo de rama:
    test/agregar-pruebas-login
  - chore/nombre:
    Para tareas menores o mantenimiento del proyecto, como actualizaciones de dependencias (equivalente a chore en los commits).
    Ejemplo de rama:
    chore/actualizar-dependencias

2.2.3 *Commits*

- Estilo de mensajes:
  Los mensajes de commit deben ser claros, concisos y seguir un formato estándar basado en el tipo de cambio. Para ello se ha de ocupar la “Conventional Commits” [4] :

[Tipo]: Descripción corta en tiempo presente.

- Tipos principales:

    - [Feature]: Para funcionalidades nuevas.
      Ejemplo**: [Feature]: Implementar registro de usuario.**
    - [Fix]: Para correcciones de errores.
      Ejemplo: **[Fix]: Resolver bug en autenticación.**
    - [Refactor]: Para cambios en el código sin alterar la funcionalidad.
      Ejemplo**: [Refactor]: Simplificar validación en formulario.**
    - [Docs]: Para cambios en los documentos de configuración.
      Ejemplo**: [Docs]: Actualizar Readme**

NOTA: *Todos estos nombres de los commits deben ir asociados al nombre de la rama como se indico en el punto anterior.*

Consideraciones:s

- Realizar commits pequeños y frecuentes: cada cambio debe abarcar una sola responsabilidad para facilitar su revisión.
- Usar descripciones claras en inglés o español.
- Evitar mensajes genéricos como "arreglado" o "cambios menores".

2.2.4 *Convención para la creación de carpetas*

Cuando se necesite agregar una nueva carpeta en el proyecto, orientado más a la rama docs, sigue estas reglas para garantizar una organización consistente:

- Usa minúsculas para los nombres de las carpetas.
- Evita caracteres especiales, como tildes o acentos, en los nombres de las carpetas.
- Si el nombre de la carpeta tiene más de una palabra, usa guiones (-) para separarlas, en lugar de espacios o guiones bajos (_).
  Ejemplo: **docs/requerimientos-del-sistema, docs/guias-de-instalacion.**
- Mantén una estructura lógica y jerárquica.

2.3 Revisión de Código

En nuestro proyecto, los cambios serán revisados a través de pull requests (PR) en nuestra plataforma de control de versiones GitHub.
Pasos para la revisión de código:

- **Creación del pull request:**
  Después de realizar cambios en una rama el desarrollador crea un pull request para integrar esos cambios a la rama develop.
- **Revisión por parte de los compañeros de equipo:**
  El pull request debe ser revisado por al menos un miembro del equipo, preferiblemente junto al que ha realizado los cambios.
- **Discusión y correcciones:**
  Se verificará el código, realizará comentarios y se planteará posibles correcciones. Corregido todas las observaciones el pull request se actualizará.
- **Aprobación y merge:**
  Una vez que todos los comentarios han sido resueltos el pull request es aprobado y fusionado con la rama develop.

Luego de esto se pasará a realizar las respectivas pruebas y se pasará a realizar el mismo proceso

anterior, pero para la fusión entre la rama develop con la main.

2.4 Pruebas

2.4.1 *Tipos de pruebas*

En el desarrollo del proyecto se implementarán diferentes tipos de pruebas para asegurar la calidad y el correcto funcionamiento del sistema. Los tipos de pruebas que se utilizarán son los siguientes:

- Pruebas Unitarias: Las pruebas unitarias son fundamentales para asegurar que cada componente del sistema funcione correctamente de manera aislada. En este caso, se probarán unidades específicas del código, como funciones, métodos de modelos y vistas individuales.
- Pruebas de Integración: Las pruebas de integración se enfocan en verificar que las diferentes unidades del sistema funcionen bien cuando se combinan entre sí.
- Pruebas de Sistema: Las pruebas de sistema se realizarán para validar que el sistema completo funciona correctamente en su conjunto.
- Pruebas de Aceptación: Las pruebas de aceptación se llevarán a cabo para verificar que las funcionalidades del sistema cumplen con los requisitos del usuario final.

2.4.2 *Herramientas de prueba*

En el caso de nuestro proyecto para verificar todo lo tiene que realizar el tester o persona encargada de esta parte, para lo cual se ha de apoyar de su experiencia, y como principal software SonarQube, si se añaden más tecnologías para las pruebas se ha de actualizar este documento con todo lo nuevo y correspondiente.

2.4.3 *Cobertura de pruebas*

El objetivo es garantizar que el código sea probado de manera exhaustiva, asegurando que las funcionalidades clave estén cubiertas.

- Nivel mínimo aceptable de cobertura
  Se buscará alcanzar un nivel mínimo de 80% de cobertura de código, aunque se dará mayor prioridad a la cobertura de los casos críticos del sistema, como las vistas y los modelos principales de la aplicación de Django.
- Estrategias de diseño de casos de prueba
  La estrategia de diseño de pruebas estará basada en los siguientes principios:
  - Pruebas de Caja Negra: Se crearán pruebas basadas en los requisitos y casos de uso, verificando que el sistema haga lo que el usuario espera sin tener en cuenta la implementación interna.
  - Pruebas de Caja Blanca: Se realizarán pruebas unitarias que se enfoquen en el código interno y la lógica de las funciones o clases individuales.
  - Cobertura de Funcionalidades Críticas: Las funciones que interactúan directamente con la base de datos deben tener pruebas unitarias e integradas.
  - Casos de borde y errores: Se realizarán pruebas específicas para verificar que el sistema maneja correctamente las entradas no válidas, los errores y los límites de datos.

2.5 Gestión de Configuración

2.5.1 *Archivos de configuración*

- Nombrado
  Para todos los documentos se utilizará la siguiente nomenclatura, evitando el uso de caracteres especiales:

[NombreDelEquipo]_[NombreDelDocumento]v[x.y].[extensión]

Ejemplo: ***AlFiloDelExito_FlujoDeTrabajov1.0.pdf***

- Ubicación
  Para todos los archivos de configuración se encontrarán en el repositorio del proyecto, específicamente en la rama “docs”, distribuido por carpetas para su identificación.