DP1 2020-2021
Documento de Requisitos y Análisis del Sistema

Foorder

https://github.com/gii-is-DP1/dp1-2020-g3-20.git

Miembros:

- Angulo Moruno, Fernando
- Chellik, Abdelkader
- García Lergo, Horacio
- Monteseirín Puig, Víctor
- Sánchez Hossdorf, Alexander
- Tabares Rodríguez, José Manuel

Tutor: García, Jose María

GRUPO G3-20

Versión 1

10/01/2021

Historial de versiones

Estos son ejemplo del contenido que debería tener el historial de cambios del documento a entregar a lo largo de los sprints del proyecto

Fecha	Versión	Descripción de los cambios	Sprint
13/12/2020	V1	● Creación del documento	2
13/12/2020	V2	● Añadido diagrama de dominio/diseño ● Explicación de la aplicación del patrón caché	3

Contents

Introducción

Foorder es un sistema informático pensado para restaurantes que buscan llevar una mejor gestión de su negocio e inventario. Las funciones principales del software son: control y gestión del inventario, facilitar el contacto con los proveedores, realización de comandas de comida y estadísticas del negocio. Todos los datos del menú, empleados e inventario serán almacenados en una base de datos.

Diagrama(s) UML:

Diagrama de Dominio/Diseño

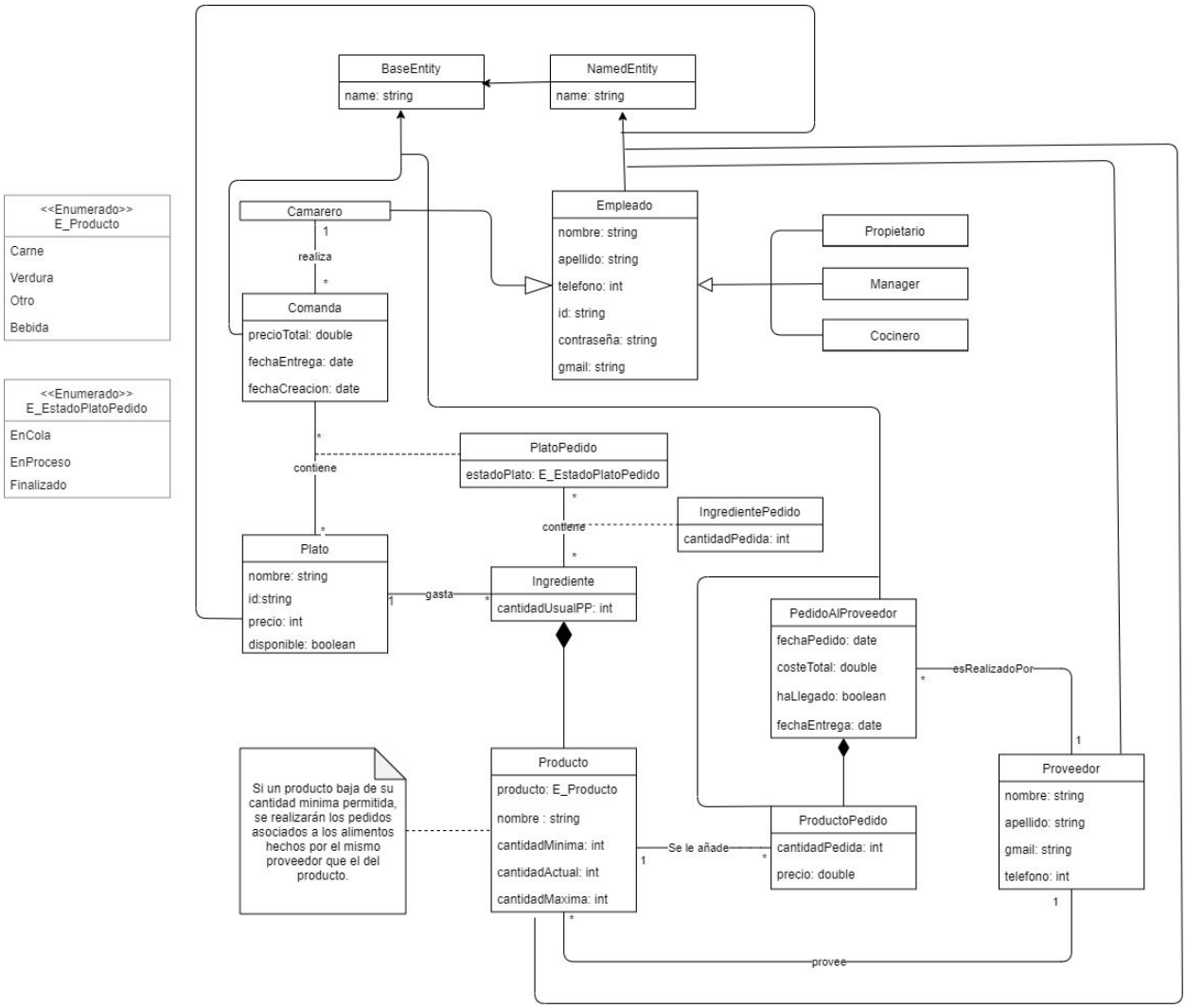
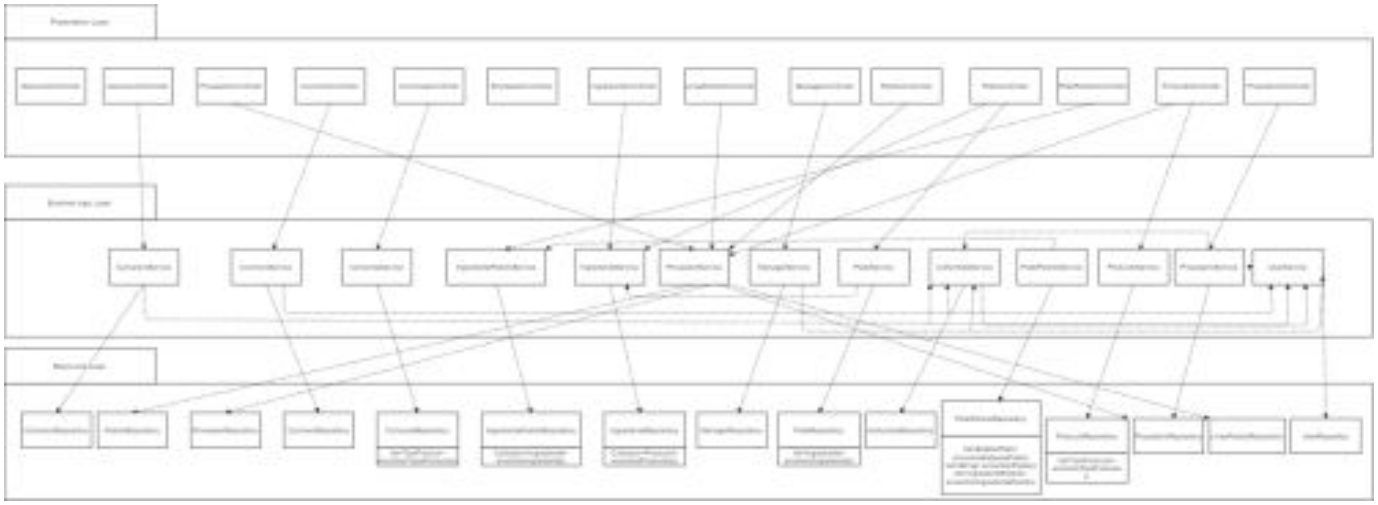


Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)

En esta sección debe proporcionar un diagrama UML de clases que describa el conjunto de controladores, servicios, y repositorios implementados, incluya la división en capas del sistema como paquetes horizontales tal y como se muestra en el siguiente ejemplo:



El diagrama debe especificar además las relaciones de uso entre controladores y servicios, entre servicios y servicios, y entre servicios y repositorios.

Tal y como se muestra en el diagrama de ejemplo, para el caso de los repositorios se deben especificar las consultas personalizadas creadas (usando la signatura de su método asociado).

Patrones de diseño y arquitectónicos aplicados

Patrón: MVC

Tipo: MVC

Contexto de Aplicación

El patron se usa en toda la aplicación.

Clases o paquetes creados

No se han creado nuevos paquetes , se mantuvieron los originales de petclinic.

Ventajas alcanzadas al aplicar el patrón

Tiene separaciones claras de dónde tiene que ir cada tipo de lógica,facilita el mantenimiento escalabilidad y pruebas unitarias así como de aplicar desarrollo guiado por pruebas.

Decisión 1: Implementar un DTO para Producto (Tomada por Horacio)

Descripción del problema:

Como grupo queríamos que el sistema nos permitiese guardar datos recibidos a través de la web para productos nuevos o cuando editamos los existentes. El problema es que al recibir los datos, por la forma en la que lo hacíamos, recibía el String que mostraba la información y esperaba el número de Id de esos datos, lo cual es un Integer.

Alternativas de solución evaluadas:

Alternativa 1.a: Poner un conversor dentro del propio controller.

Ventajas:

- Simple, rápido.

Inconvenientes:

- Llegaría como resultado del BindingResult un error que haría saltar el if y habría que quitar ese error.

Alternativa 1.b: Crear una entidad ProductoDTO que reciba los parámetros que va a mandar la página web y traducirlos en un tipo Producto en el save.

Ventajas:

- Sencillo, intuitivo y no saltarían problemas con el BindingResult de base.

Inconvenientes:

- Habría que crear una entidad nueva con sus respectivas clases, es un proceso lento.
- Habría que recordar cuándo solicitamos un Producto y cuándo un ProductoDTO.

Justificación de la solución adoptada

Se consideró que evitar recibir errores es lo más inteligente cuando estás trabajando en un sistema funcional, pues puede evitar futuros problemas, además es algo sencillo y fácil de aplicar, por lo que se escogió la alternativa 1.b.

Decisión 2: Implementar Tipo Producto como una tabla (Tomada por Horacio)

Descripción del problema:

Como grupo queríamos que el sistema contara con enumerados para clasificar algunas entidades. El problema es que los enumerados dan bastantes problemas con el SQL.

Alternativas de solución evaluadas:

Alternativa 1.a: Hacerlo con la función de la librería javax.persistence @Enumerated.

Ventajas:

- Es una función ya implementada.

Inconvenientes:

- Requiere tiempo para entender bien su funcionamiento y las condiciones que hay que cumplir para que funcione.

Alternativa 1.b: Hacer una tabla con nombre e id, pues son los atributos básicos de un enumerado y trabajar con ella como otra relación más.

Ventajas:

- Podemos modificar en el SQL rápido los datos del enumerado.
- Si en un futuro quisiéramos modificar los tipos de productos podríamos hacerlo sin tener que parar el servidor.

Inconvenientes:

- No usamos una función ya implementada.

Justificación de la solución adoptada

Puesto que carecemos de tiempo, acabamos valorando más nuestro tiempo añadiendo unas líneas más de código que sí comprendemos al 100%, por lo que se escogió la alternativa 1.b.

Decisión 3: Implementar todas las vistas iniciales de los empleados a través del mismo navbar(Tomada por Jose)

Descripción del problema:

Una de los historias de usuario que teníamos pendiente era la H4- Control de cuentas, en la que el propietario pedía gestionar las cuentas de acceso al sistema de mis empleados para así poder restringir las tareas que puedan realizar, entonces decidí que cada empleado tuviera una vista con sus respectivas partes en el proyecto, entonces decidí que dependiendo de qué persona hiciera login, tuviera un acceso a su función directamente en el navbar y tuviera una vista diferente .

Alternativas de solución evaluadas:

Alternativa 1: crear una pantalla de login por cada empleado.

Ventajas:

- Simple, son solo más botones en la pantalla de inicio.

Inconvenientes:

- Más tedioso para el usuario de la aplicación.
- Puede dar lugar a confusión para el usuario al iniciar sesión desde la página de otro tipo de empleado y ver que no funcionaba.

Alternativa 2: Usar el mismo login para todos los empleados y que te lleve a la misma pantalla en la que se vea distinta información dependiendo de quién es el empleado.

Ventajas:

- Más intuitivo para los usuarios de la aplicación.

Inconvenientes:

- Menos intuitivo para el programador al haber mucha información condensada en un único lugar.

Justificación de la solución adoptada

Decidimos que la aplicación fuera más intuitiva para los usuarios por lo que seleccionamos la alternativa de diseño 2.

Decisión 4: Crear una clase Empleado que extienda a todas las demás de personas(Tomada por Jose)

Descripción del problema:

Decidí utilizar una clase empleado que se extendiera a todas las clases de personas que utilizaremos durante el proyecto(camarero, cocinero, propietario y manager).

Alternativas de solución evaluadas:

Alternativa 1.a: Crear cada clase con todos los respectivos atributos de empleado.

Inconvenientes:

- Ralentiza todo el trabajo con el sistema para el desarrollo.
- Sería menos eficiente, puesto que habría más código y además serían iguales unos a otros.

Decisión 5: No se pueda editar el proveedor de un producto(Tomada por Víctor)

Descripción del problema:

Los pedidos tienen asociado un producto y el proveedor que lo entrega, si se modifica el proveedor del producto el pedido daría error,

Alternativas de solución evaluadas:

Alternativa 1.a: Que se modifica el pedido cuando se editase el proveedor , pero la idea es que si un pedido se ha realizado no se modifique automáticamente nunca ya que si se pidió a un proveedor , y luego se le va a pedir a otro si se cambiase el producto se cambiaría el pedido a pesar de que se le pidió a otro.

Ventajas:

- Simple, no requiere nada más

Inconvenientes:

- Para editar un proveedor hay que crear un nuevo producto

Justificación de la solución adoptada

Consideramos que era más conveniente y muy sencillo de aplicar.

Decisión 6: Meter el service de LineaPedido en Proveedor Service (Tomada por Alexander y Víctor)

Descripción del problema:

Proveedor, pedido y linea pedido están muy relacionados y no tiene sentido que linea pedido este en un service diferente que pedido.

Alternativas de solución evaluadas:

Alternativa 1.a: Quitar línea pedido service de producto service y meterlo en proveedor service.

Ventajas:

- Más comodidad a la hora de gestionar los service.

Inconvenientes:

- Hay tres service en una única clase (el de pedido, proveedor y línea pedido) y puede ser complicado a la hora de buscar funciones.

Justificación de la solución adoptada

Elegida por ser la única alternativa.

Decisión 7: Quitar la vista de listado de todas las lineas pedido. (Tomada por Alexander y Víctor)

Descripción del problema:

En la vista de listado de pedidos había un botón que nos llevaba a un listado con todos las líneas de pedido sin filtrar por ninguna ID, algo que es inutil porque vemos una lista de todas las líneas pedidos que hay.

Alternativas de solución evaluadas:

Alternativa 1.a: Quitar dicho botón y añadir un botón en cada pedido en el listado para que nos lleve a las lineas pedido de ese respectivo pedido, filtrandolo por ID.

Ventajas:

- Poder ver las líneas de pedido de un pedido y saber qué productos hemos pedido.

Inconvenientes:

- Ninguno

Justificación de la solución adoptada

Elegida por ser la única alternativa.

Decisión 8: Quitar la barra de busqueda de linea pedido por pedido ID (Tomada por Alexander y Víctor)

Descripción del problema:

Debido al anterior decisión, al entrar en la vista de listado de línea pedido, la barra de búsqueda por pedido ID no tenía sentido ya que ya tenemos la lista filtrada por ID.

Alternativas de solución evaluadas:

Alternativa 1.a: Quitar la barra de búsqueda del jsp ya que esa función la realiza el botón en pedido.

Ventajas:

- Al querer obtener las lineas pedido filtradas, se obtendrá de forma más fácil que teniendo que poner la ID en la barra del buscador, siendo algo más ineficiente.

Inconvenientes:

- Si por algún casual se llega a la vista con el listado de todas las líneas pedidos existentes, no podríamos buscarlo por ID al no haber barra de búsqueda.

Justificación de la solución adoptada

Elegida por ser la única alternativa.

Decisión 9: Al finalizar el pedido, se actualiza la fecha de finalización automáticamente. (Tomada por Alexander y Víctor)

Descripción del problema:

En pedido, la fecha de finalización debía editarse tras finalizar el pedido y meterla a mano.

Alternativas de solución evaluadas:

Alternativa 1.a: Al pulsar el botón de finalizar pedido, se actualiza la fecha automáticamente a la fecha del día que se ha pulsado.

Ventajas:

- El propietario o el manager no tiene que entrar en editar pedido para poner a mano el día en el que se ha finalizado el pedido.

Inconvenientes:

- Ninguno.

Justificación de la solución adoptada

Elegida por ser la única alternativa.

Decisión 10: Eliminar en cascada las lineasPedido de un pedido asociadas a un producto al eliminar dicho producto sin eliminar(Tomada por Fernando y Victor)

Descripción del problema:

Se deben eliminar las líneas de pedido en cascada sin eliminar los pedido asociados.

Alternativas de solución evaluadas:

Alternativa 1.a: Eliminación en cascada de las líneas de pedido al eliminar el producto manteniendo intacto el pedido.

Inconvenientes:

- Puede que haya líneas de pedido asociadas a un pedido que no queramos eliminar.
  .

Justificación de la solución adoptada

Más conveniente para ajustarse a los requisitos del proyecto.

Decisión 11: Añadir información sobre el producto asociado a un ingrediente(Tomada por Fernando)

Descripción del problema:

Se debe obtener información sobre el producto que referencia un ingrediente asociado a cada plato.

Alternativas de solución evaluadas:

Alternativa 1.a: Representar la información directamente como una foreign key de producto en la clase ingrediente .

Alternativa 1.b: Representar la información a través de una clase a parte donde tiene la información sobre los platos y los productos.

Inconvenientes:

- Hay que buscar la forma que sea menos complicada a la hora de comenzar a programar las entidades

Justificación de la solución adoptada

Más conveniente para la correcta codificación de los requisitos.