DP1 2020-2021

Documento de Diseño del Sistema

Mineral House Spa

https://github.com/gii-is-DP1/dp1-2020-g2-07

Miembros:

- JUAN MANUEL GARCIA CRIADO
- FERNANDO MIGUEL HIDALGO AGUILAR
- DIEGO MARQUEZ GONZALEZ
- MIGUEL MOLINA RUBIO
- JAVIER RAMOS ARRONDO
- FRANCISCO JAVIER RODRIGUEZ ELENA

Tutor: Irene Bedilia Estrada

GRUPO G2-07

Versión 1.0

08/01/2021

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint
08/01/2021	V1	● Creación del documento ● Se escribe la introducción y cada miembro del equipo rellena sus decisiones tomadas	3
	V2	● Añadida decisión de diseño 14 ● Añadidos ambos diagramas UML	4

Contents

Introducción

Con este software se desea no solo agilizar el proceso de administración de un Spa sino también mejorar la experiencia de los clientes a la hora de facilitar funcionalidades del mismo. Por parte de la administración, implementaremos funciones tales como registro de usuario, gestión de abonos, organización de clases, selección de monitor…

Como hemos comentado anteriormente, no solo los administradores tendrán acceso a nuestro software, sino que los clientes también se verán beneficiados de él. Algunos ejemplos de funcionamiento destinado a los clientes son: inscripción virtual a clases, domiciliación del pago, autogestión de tarifa contratada…

Diagrama(s) UML:

Diagrama de Dominio/Diseño

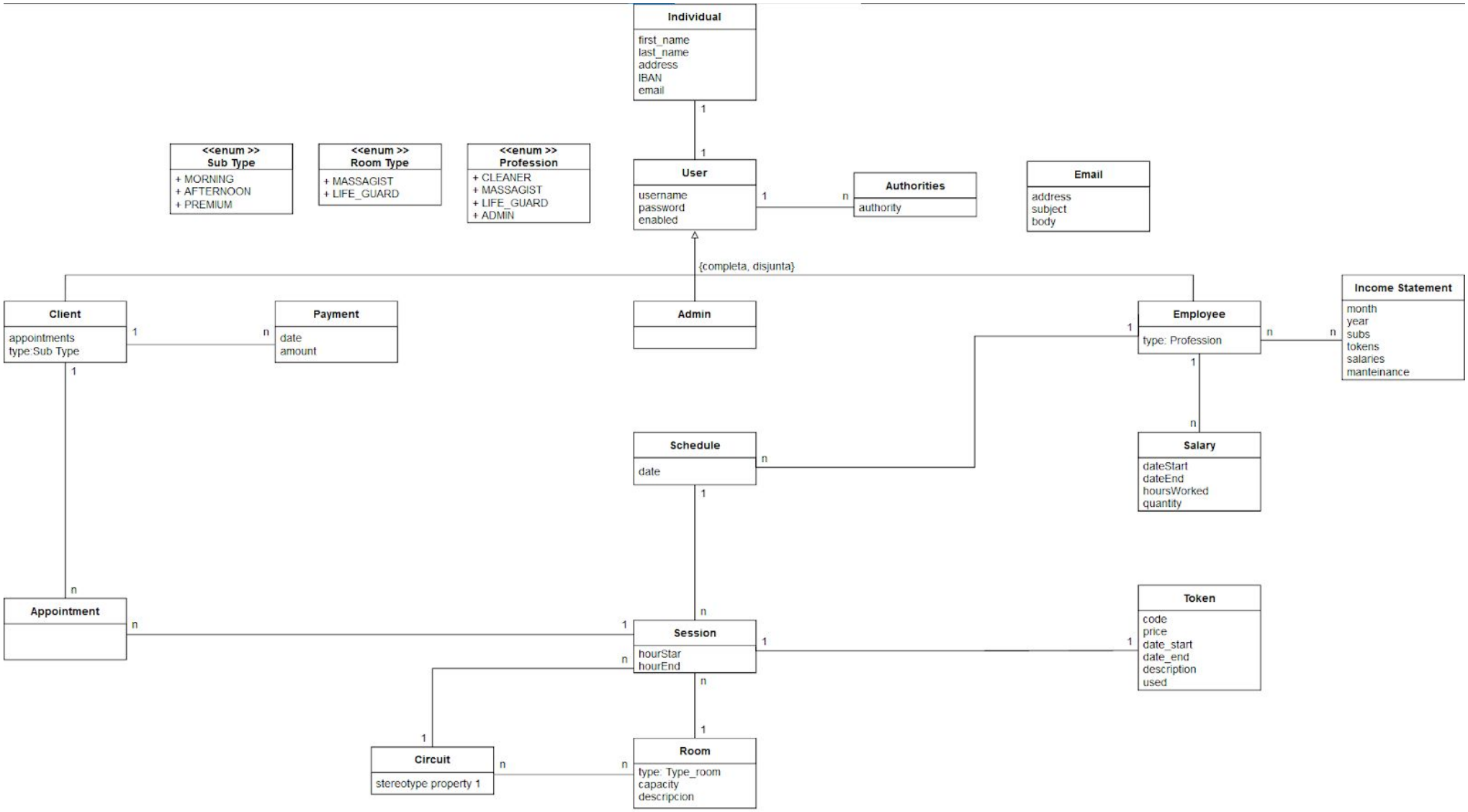
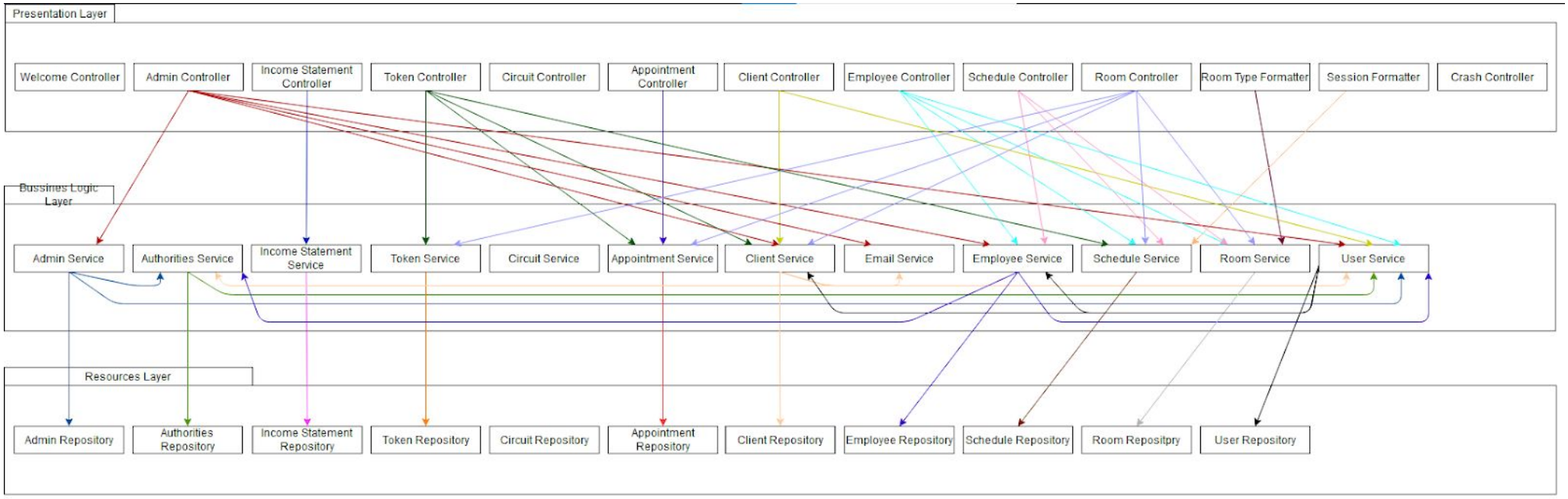


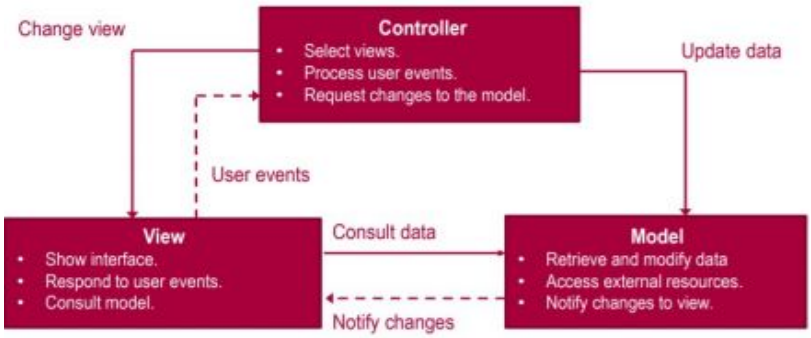
Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)



Patrones de diseño y arquitectónicos aplicados

Patrón: MVC

Tipo Arquitectonico



Contexto de Aplicación

Se utiliza en el total de la aplicación, separa los datos de la aplicación,la interfaz de usuario y la lógica de negocio en 3 componentes distintos:

Modelo: Es la representación específica de la información con la que se opera. Incluye los datos y la lógica para operar con ellos.Dentro del modelo se incluye tres elementos:entidades(objeto de dominio cuyos datos se conservan en la base de datos y tienen su propia identidad),repositorios(proporcionan métodos para recuperar y guardar objetos de dominio en la base de datos) y servicios(exponen la funcionalidad del sistema).

Controlador:Responde a eventos de la interfaz de usuario e invoca cambios en el modelo y probablemente en la vista(intermediario entre vista y modelo).

Vista:Es la presentación del modelo de forma adecuada para interactuar con ella y representan la información proporcionada por el controlador (en el ModelMap).

Clases o paquetes creados

Con respecto a los paquetes y clases creadas, se ha creado paquetes model,service,repository que hacen referencia al modelo de la aplicación.En este se han implementado las distintas entidades,servicios y repositorios con las que se han trabajado para la construcción de la aplicación.

- model:
  Admin,Authorities.java,Balance.java,BaseEntity.java,Bono.java,Categoría.java,Circuito.java,Cita.java,Cliente.java,Email.java,Employee.java,EmployeeRevenue.java,Horario.java,Individual.java,NamedEntity.java,Pago.java,Professesion.java,RoomType.java,Sala.java,Sesion.java,Subtype.java,TokenCode.java,User.java.
- service:
  AdminService.java,AuthoritiesService.java,BalanceService.java,BonoService.java,CircuitoService.java,CitaService.java,ClienteService.java,EmailService.java,EmployeeService.java,HorarioService.java,SalaService.java,UserService.java.
- repositoy:
  AdminRepository.java,AuthoritiesRepository.java,BalanceRepository.java,BonoRepository.java,CircuitoRepository.java,CitaRepository.java,ClienteRepository.java,EmployeeRepository.java,HorarioRepository.java,SalaRepository.java,UserRepository.java.

Más adelante se ha creado también un paquete web, que hace referencia al controlador de la aplicación.

- web:AdminController.java,BalanceController.java,BonoController.java,CircuitoController.java,ClienteController.java,CrashController.java,EmployeeController.java,HorarioController.java,SalaController.java,WelcomeController.java.

Por último con respecto a las vistas, las entidades del modelo admin,balances,bonos,circuitos,clientes,employees,pay,salary,timetable tienen varias clases jsp para diferentes finalidades.

- admin:adminHome,checkUsers,createOrUpdateAdminForms,newAnnouncement,newEmail
- balances:balanceDetails.jsp,BalanceListing.jsp.
- bonos:BonosListing.jsp,createToken.jsp,ReedemToken.jsp.
- circuitos:CircuitosListing.jsp,createOrUpdateCircuitosForm.jsp.
- clientes:clienteDetails.jsp,ClientsLisiting.jsp,createOrUpdateClientsForm.jsp
- employees:createOrUpdateEmployeeForm.jsp,employeeDetails.jsp,employeePastSessions.jsp,employeeListing.jsp,employeeTimeTable.jsp.
- pay:payForm.jsp.
- salary:salaryForm.jsp.
- salas:CreateOrUpdateForm.jsp,salaDetails.jsp,SalasListing.jsp.
- timetable:timetable:horarioForm.jsp,sesionForm.jsp
- exception.jsp,welcome.jsp.

Ventajas alcanzadas al aplicar el patrón

Soporte para múltiples vistas, favorece la alta cohesión,el bajo acoplamiento y la separación de responsabilidades. Facilita el desarrollo y las pruebas de cada tipo de componente en paralelo.

Patrón: Front Controller

Tipo Diseño

Contexto de Aplicación

Maneja todas las solicitudes de un sitio web y luego envía esas solicitudes al controlador apropiado.La clase @controller tiene los métodos adecuados para el manejo de solicitudes.Todos las solicitudes manejadas por los métodos de esta clase comienzas por /circuitos

```
@Controller
@RequestMapping("/circuitos")
public class CircuitoController {

    public static final String CIRCUITOS_FORM ="/circuitos/createOrUpdateCircuitosForm";
    public static final String CIRCUITOS_LISTING ="/circuitos/CircuitosListing";

    private final CircuitoService circuitosServices;
    private final SalaService salasServices;

    @Autowired
    public CircuitoController(CircuitoService circuitosServices, SalaService salasServices) {
        this.circuitosServices = circuitosServices;
        this.salasServices = salasServices;
    }

    @GetMapping
    public String cicuitosListing(ModelMap model) {
        Collection<Circuito> c = circuitosServices.findAll();
        for(Circuito i:c) {
            i.setAforo(circuitosServices.getAforo(i));
        }
        model.addAttribute("circuitos",c);
        return CIRCUITOS_LISTING;
    }
```

Este método maneja solicitudes GET que comienzan con URL similares a : “circuitos/{1}/edit”.

```
    @GetMapping("/{id}/edit")
    public String editCircuito(@PathVariable("id") int id,ModelMap model ) {
        Optional<Circuito> circuito = circuitosServices.findById(id);
        if(circuito.isPresent()) {
            Circuito c = circuito.get();
            c.setAforo(circuitosServices.getAforo(c));
            model.addAttribute("circuito",c);
            model.addAttribute("salas", salasServices.findAll());
            return CIRCUITOS_FORM;
        }else {
            model.addAttribute("message","We could not find the circuit you are trying to edit.");
            return CIRCUITOS_LISTING;
        }
    }
```

Clases o paquetes creados

El paquete creado para la adaptación de este patrón de diseño es el mismo que el paquete “web ” que forma parte del controlador de la aplicación.

Ventajas alcanzadas al aplicar el patrón

El control centralizado permite la aplicación de políticas en toda la aplicación, como el seguimiento de usuarios y seguridad.La decisión sobre quien es el administrador apropiado de una solicitud puede ser más flexible y el FrontController puede proporcionar funciones adicionales,como el procesamiento de solicitudes para la validación y transformación de parámetros.

Patrón:Arquitectura centrada en datos

Tipo Diseño

Contexto de Aplicación

Se ha implementado base de datos donde están almacenados los datos de la propia empresa y donde se irán almacenando los cambios y nuevos datos.

Clases o paquetes creados

Se ha llevado a cabo la creación de una clase data.sql

Ventajas alcanzadas al aplicar el patrón

La principal ventaja es que nos ha permitido crear y almacenar datos y sobre todo poder acceder a ellos para la posible manipulación de estos.

Patrón:Service Layer

Tipo Diseño

Contexto de Aplicación

Define la frontera de la aplicación con una capa de servicios que establece un conjunto de operaciones disponibles y coordina la respuesta de la aplicación en cada operación. La capa de presentación interactúa con la de dominio a través de la capa de servicio.

Clases o paquetes creados

El paquete creado para la adaptación de este patrón de diseño es el mismo que el paquete “service ” que forma parte del módulo de la aplicación.

Ventajas alcanzadas al aplicar el patrón

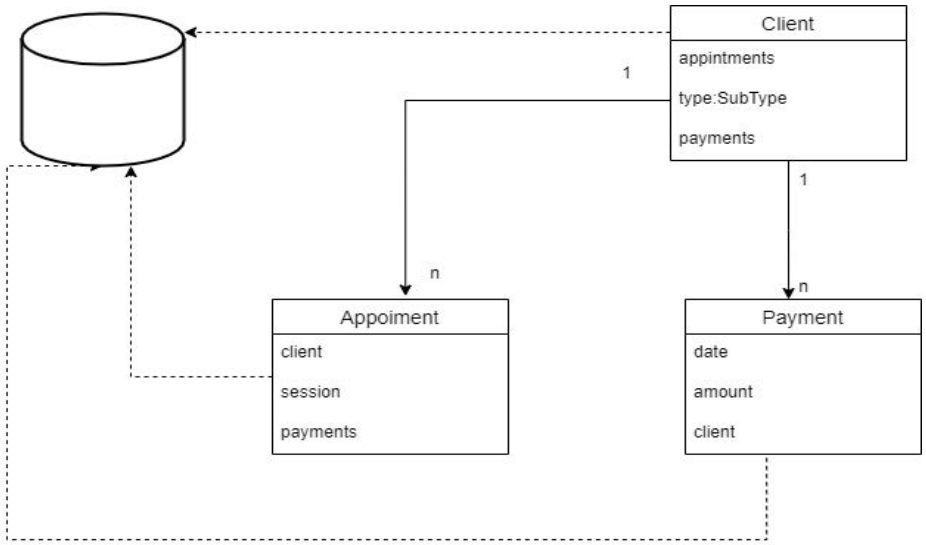
Nos permite tener muchos casos de uso que envuelven entidades de dominio y interactuar con servicios externos (tales como la base de datos).

Patrón:Domain Model

Tipo Diseño

Contexto de Aplicación

Este patrón nos ha ayudado a que el negocio pasa a ser sostenido en términos de objetos, estos objetos tienen comportamiento y su interacción depende de los mensajes que se comunican entre ellos, es el patrón orientado a objetos por excelencia y hace que los objetos modelados estén en concordancia con el negocio.



Clases o paquetes creados

El paquete creado para la adaptación de este patrón de diseño es el mismo que el paquete “model ” que forma parte del módulo de la aplicación.

Ventajas alcanzadas al aplicar el patrón

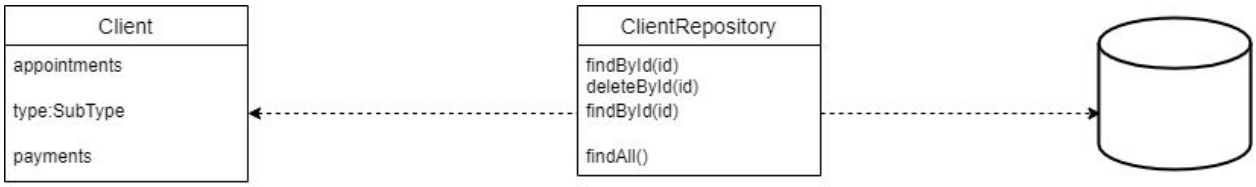
Facilita la concordancia de los objetos modelados del negocio.

Patrón:(Meta) Data Mapper

Tipo Diseño

Contexto de Aplicación

Este patrón nos permite realizar una transferencia bidireccional de datos entre la base de datos y el objeto del dominio.



Clases o paquetes creados

Las clases BonoRepository, ClientRepository y AdminRepository siguen a la perfección este patrón.

Ventajas alcanzadas al aplicar el patrón

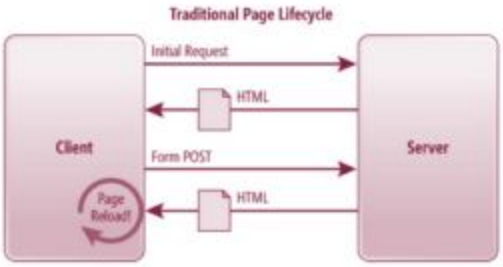
La principal ventaja es mover datos entre el objeto y la base de datos mientras los mantiene independiente de cada uno.

Patrón:Traditional page lifecylce

Tipo Diseño

Contexto de Aplicación

Conforme a la paginación de nuestra aplicación, está basada en el patrón Traditional page lifecyle. Lo que nos aporta a nuestra aplicación es una nueva página cada vez que el cliente interactúa con la aplicación, es decir se recarga la página cada vez que se produzca una request html por parte del cliente.La estructura que sigue es la siguiente:



Clases o paquetes creados

Dado que el servidor nos devuelve datos HTML, las clases implementadas en el controlador están diseñadas y codificadas para satisfacer esos datos devueltos.

Ventajas alcanzadas al aplicar el patrón

La principal ventaja que no has supuesto trabajar con este patrón es que la conversión de HTML a objetos Java de nuestro modelo de dominio es fácil ya que la hemos estado trabajando durante todo el desarrollo del proyecto.Por nuestra experiencia, podemos afirmar que el tiempo de carga es rápido.

Patrón:Repositories

Tipo Diseño

Contexto de Aplicación

Para las funcionalidades necesarias acorde a nuestra aplicaciones hemos implementado repositorios de tipo CRUD y Repository por cada entidad para así poder interactuar con la base de datos y con los objetos Java del modelo de dominio.

Clases o paquetes creados

Todos los implementados dentro del paquete “repository” mencionado anteriormente.

Ventajas alcanzadas al aplicar el patrón

La ventaja al utilizar este patrón es que no ha permitido un acceso más funcional a la base de datos. Hemos podido definir nuestros propios métodos, y algunos de ellos con querys.

Patrón:Layer super type

Tipo Diseño

Contexto de Aplicación

Nos ha sido esencial la utilidad de este patrón para la codificación de la aplicación ya que varios objetos java de nuestro modelo de dominio son extendidas por estas.

Clases o paquetes creados

Se han creado las siguientes clases para la adaptación de este patrón:BaseEntity.java,NamedEntity.java e Individual.java

Ventajas alcanzadas al aplicar el patrón

-Identifica entidades

-Auto incrementa el nuevo id en la base de datos

Decisiones de diseño

Decisión 1: Expresión de los horarios del trabajador (Tomada por Javier Ramos)

Descripción del problema:

El horario del empleado está formado por todas las fechas en las que trabaja en el spa. A su vez, dentro de todas esas fechas, se posee una serie de divisiones que constituyen las diferentes sesiones en las que desempeña su labor.

Alternativas de solución evaluadas:

Alternativa 1.a: Unificar la jornada laboral en una misma entidad.

Ventajas:

- Simple,nos permite expresar de manera sencilla el horario del trabajador.

Inconvenientes:

- No es realista, ya que la idea del spa no implica que el empleado tenga que desempeñar la jornada completa en una misma sala.
- Complica la futura creación de citas, en las que no se especificará la división de las sesiones.
- Puede darse la situación en la que el trabajador tenga turno de mañana y de tarde, habría que añadir variables extra a la entidad.

Alternativa 1.b: Hacer que el empleado posea una lista con las fechas laborables, cada una enlazada con una lista de sesiones diarias.

Ventajas:

- Facilidad a la hora de crear las citas de cara al futuro: una sesión ->citas para esa sesión.
- Mayor flexibilidad para añadir sesiones a cada día.

Inconvenientes:

- Mayor complicación para la creación de la jornada laboral y la extracción de datos.

Alternativa 1.c: Hacer que el empleado posea un único calendario, dentro de él una lista con las fechas laborables, cada una enlazada con una lista de sesiones diarias.

Ventajas:

- Toda la lógica del horario del trabajador quedaría recogida dentro de una entidad calendario.

Inconvenientes:

- Habría una entidad más respecto a la alternativa anterior y a la hora de extraer datos la sintaxis sería mayor (cliente.getCalendario().getHorario(x).getSession(x))

Justificación de la solución adoptada

Se decidió tomar como solución la alternativa b. A pesar de que supone una mayor carga de trabajo inicial y una mayor dificultad para extraer los distintos elementos en el controlador, resulta clave ir pensando en las citas.

Decisión 2: Limitación de acceso de usuario (Tomada por Miguel Molina)

Descripción del problema:

Al tener las vistas de usuario hechas nos dimos cuenta de que aún así el usuario podía acceder a páginas que no debería poder acceder mediante la url.

Alternativas de solución evaluadas:

Pensamos en limitar el acceso a esas páginas usando la Security Configuration, pero nos encontramos con el problema de que estas restricciones se aplican globalmente a una url, por ejemplo, podemos añadir una restricción a /clientes/* porque no queremos que el usuario acceda a otros perfiles, pero entonces tampoco podría acceder al suyo propio.

Justificación de la solución adoptada

La solución por la que hemos optado es una mezcla entre restricciones en el Security Configuration y en los controladores, comprobando si el usuario tiene permiso para ver una vista y en el caso contrario llevarle a una página de error.

Decisión 3: Implementación de un carrusel de 3 fotos sin botones (Tomada por Diego Márquez)

Descripción del problema:

Al realizar el carrusel de imágenes del centro e implementar correctamente su CSS, nos percatamos que su código presenta incompatibilidades con elementos del navbar.

Alternativas de solución evaluadas:

Algunas de las alternativas que nos planteamos fue modificar nuestro carrusel a otro diferente, pero visualmente quedó peor implementado.

Justificación de la solución adoptada

Al final optamos por eliminar los botones y hacer el carrusel automático, de esta manera, desaparecen las incompatibilidades con el navbar y quedaba bien integrado visualmente en el proyecto.

Decisión 4: Registro de usuarios (Tomada por Juan Manuel García y Miguel Molina)

Descripción del problema:

El registro de usuarios debía estar limitado a clientes y empleados, no queremos que cualquier persona pueda registrarse que sea ajena al establecimiento, por lo que necesitamos que el administrador verifique a los nuevos usuarios registrados.

Alternativas de solución evaluadas:

Que el administrador pueda enviar un código a un correo de un usuario que quiere registrarse y pueda usarlo en la página web para registrarse, pero esto resulta en que el administrador tiene que estar generando códigos individuales siempre que alguien quiera registrarse, no es una solución cómoda.

Justificación de la solución adoptada

Al final optamos por permitir a todo el mundo registrarse, pero sus cuentas están inicialmente desactivadas, por lo que el administrador desde su panel de control puede decidir qué usuarios permitir entrar fácilmente sin estar preocupado de enviar códigos y que éstos los usen.

Decisión 5: Separación entre días anteriores y futuros(Tomada por Javier Ramos)

Descripción del problema:

Dentro de la tabla de horarios del empleado, se visualizaba un registro completo de las fechas y sesiones asignadas a cada fecha para tener constancia del trabajo realizado.

Alternativas de solución evaluadas:

Se presentó la posibilidad de mostrar en el perfil del empleado únicamente las jornadas que tenía por delante. Para no eliminar el trabajo ya realizado, y que sirva a modo de consulta, se propuso mostrar en otra vista las sesiones que ya habían pasado.

Ventajas:

- Aumentamos la limpieza y claridad en la tabla del empleado.
- Código aprovechable para mostrar sólo las sesiones en las que se puede pedir cita.

Inconvenientes:

- Diseño de otra tabla para fechas ya pasadas.

Justificación de la solución adoptada

A pesar de suponer la inserción de una nueva tabla, son funciones que podrán ser útiles en otros aspectos.

Decisión 6:Implementación de una barra divisoria (Tomada por Diego Márquez)

Descripción del problema:

Al ir incluyendo distintos elementos en un mismo JSP, decidimos dividirlos entre sí.

Alternativas de solución evaluadas:

Una de las alternativas que se nos presentaba era incluir un elemento *hr* en nuestro JSP, pero visualmente no quedaba bien.

Justificación de la solución adoptada

Al final decidimos incrustar una imagen entre los elementos. De esta forma, le hemos dado el diseño que hemos creído conveniente y encaja a la perfección.

Decisión 7:Implementación de la API de Google Maps a través de un IFrame(Tomada por Diego Márquez)

Descripción del problema:

Al plantearnos cómo debería ser visualmente nuestro proyecto, llegamos a la conclusión de que necesitábamos mostrar a la gente dónde nos ubicamos, ya que nuestro servicio vive de la clientela.

Alternativas de solución evaluadas:

La alternativa que encontramos para implementarlo fue implementar la API de Google Maps en el proyecto, pero presenta muchas desventajas. Algunas de estas son:

- La API de Google Maps es de pago.
- Difícil integración en el proyecto
- Mucho gasto de recurso para muy poco beneficio (debíamos aprender cómo funcionaba la API y cómo trabajar con ella)

Justificación de la solución adoptada

Al final optamos por utilizar un iframe, ya que solucionaba nuestro problema de una manera rápida y sencilla.

Decisión 8: Relación N:N entre salas y circuitos (Tomada por Fco Javier y Javier Ramos)

Descripción del problema:

Debido a la inmensa cantidad de problemas en la implementación de la relación entre ambas entidades a causa de no poder en una misma columna una lista de salas,optamos por la siguiente decisión:varias salas pueden formar parte de varios circuitos a través de una dependencia N:N.

Alternativas de solución evaluadas:

Alternativa 1.a: Separar dichas entidades en dos y que dependan ambas con un MantyToMany .

Ventajas:

- Aumentamos el acoplamiento en el sistema.
- Separación de responsabilidades.
- Aumentamos la modularidad.

Inconvenientes:

- Diseño de dos entidades:Salas y circuitos.

Justificación de la solución adoptada

La justificación que tiene esta decisión era simple, dejar libertad de crear varios circuitos y asignarles todas las salas deseadas, para así evitar fallos en la dependencia.

Decisión 9:Aviso por email (Tomada por Juan Manuel García Criado)

Descripción del problema:

Se requería de una forma externa para avisar del registro de nuevos clientes, pues el aviso en el panel de control de administrador pero era fácil de obviar dejando usuarios sin dar de alta durante días.

Alternativas de solución evaluadas:

La alternativa pensada fue, crear popups de aviso para admin

Ventajas:

- Fácil de implementar

Inconvenientes:

- En principio engorroso y ensuciando la vista

Justificación de la solución adoptada:

Se decidió implementar el aviso por email pues aparte de suponer una opción más elegante y limpia externalizando funciones, nos permite aprovechar y añadir funcionalidades extra como el envío de circulares.

Decisión 10: (Tomada por Fernando Miguel Hidalgo Aguilar)

Descripción del problema:

Se decidió incorporar estadísticas en la sección de los balances, mediante el uso de la librería CanvasJS y GSon. Según la documentación y ejemplos de la librería, el objeto GSon se define desde el JSP, sin embargo, no funcionaba

Alternativas de solución evaluadas:

Crear el objeto GSon en el Controller

Ventajas:

- Permite usar librería en el proyecto

Inconvenientes:

- Ninguno

Justificación de la solución adoptada:

Se decidió tomar esta solución para garantizar el correcto funcionamiento de la librería

Decisión 11: (Tomada por Fernando Miguel Hidalgo Aguilar)

Descripción del problema:

Para crear los bonos, es necesario conocer la sala para la que será efectivo, por tanto, se necesita su id.

Alternativas de solución evaluadas:

Colocar el botón de creación de bonos en el Listing de salas

Ventajas:

- Fácil de implementar, además de unificar ambos apartados en una misma vista

Inconvenientes:

- Ninguno

Justificación de la solución adoptada:

Se decidió tomar esta decisión pues es Listing es el lugar más sencillo y con más sentido colocar el acceso a la creación de bonos. Se podría haber colocado dentro del perfil de la sala, pero el admin (quien es responsable de crear bonos) no tiene acceso a esta vista

Decisión 12: (Tomada por Fernando Miguel Hidalgo Aguilar)

Descripción del problema:

Para canjear un bono, es necesario comprobar su código, de tipo String. Esto se hace desde un formulario de canje donde se introduce dicho código. Sin embargo, si se le pasa al modelAtribute del JSP el tipo String, devuelve error

Alternativas de solución evaluadas:

Crear una clase Token Code, cuyo único atributo es String

Ventajas:

- Corrige el error y permite el correcto funcionamiento

Inconvenientes:

- Se crea una clase que funciona como paso intermedio

Justificación de la solución adoptada:

Se decidió tomar esta decisión pues es la más sencilla encontrada para corregir el problema

Decisión 13: Salario de los empleados (Tomada por Miguel Molina)

Descripción del problema:

Necesitamos que los empleados tengan un salario asignado para poder calcular lo que han ganado en base a las horas trabajadas.

Alternativas de solución evaluadas:

Asignar a los empleados un salario fijo individual al mes.

Justificación de la solución adoptada:

En vez de asignar a cada trabajador un salario individual, hemos optado por asignar a las distintas categorías de empleado (limpiador, masajista, etc) un salario/hora fijo porque así podemos evitar problemas de salarios asignados erróneamente, y como es por hora y no total, podemos fácilmente multiplicar este salario/hora por las horas trabajadas y obtener el salario total.

Decisión 14: Salario de los empleados (Tomada por Fernando Miguel Hidalgo Aguilar)

Descripción del problema:

Se busca conseguir una segunda relación N:N.

Alternativas de solución evaluadas:

Alternativa 1.a: Recuperar la entidad toalla y utilizarla de alguna forma

Ventajas:

- Simple, cuando existía era la clase más sencilla de todo el proyecto.

Inconvenientes:

- Debido a su sencillez, sería una relación que no aporta ningún fundamento al proyecto, tan solo cubriría el problema

Alternativa 1.b: Hacer que solo los empleados que el admin decida, los de su mayor confianza, puedan consultar los balances

Ventajas:

- Idea bien fundamentada, que aporta complejidad al proyecto
- Nos otorga una regla de negocio

Inconvenientes:

- Al principio puede parecer difícil de implementar

Justificación de la solución adoptada:

Se ha optado por la alternativa b, pues es la más interesante de trabajar, además de aportar un mayor empaque al apartado de Balances