DP1 2020-2021

Documento de Diseño del Sistema

Proyecto Taller Sevilla Customs

<https://github.com/gii-is-DP1/dp1-2020-g2-01>

Miembros:

* Arias Ruda, Sergio
* Borrego Caro, Francisco José
* Funes Mejías, Antonio
* Morales García, Alejandro
* Vargas Zambrana, Jesús

Tutor: Jose Antonio Parejo

# GRUPO G02-01

## Versión 1.1

28 de diciembre de 2020

Historial de versiones

|  |  |  |  |
| --- | --- | --- | --- |
| **Fecha** | **Versión** | **Descripción de los cambios** | **Sprint** |
| 28/12/2020 | V1 | * Creación del documento * Añadido diagrama de dominio/diseño | 3 |
| 30/12/2020 | V1.1 | * Añadido patrones * Añadido decisiones de diseño | 3 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

[Historial de versiones 2](#_Toc58783389)

[Introducción 4](#_Toc58783390)

[Diagrama(s) UML: 4](#_Toc58783391)

[Diagrama de Dominio/Diseño 4](#_Toc58783392)

[Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios) 5](#_Toc58783393)

[Patrones de diseño y arquitectónicos aplicados 5](#_Toc58783394)

[Decisiones de diseño 5](#_Toc58783395)

[Decisión X 6](#_Toc58783396)

[Descripción del problema: 6](#_Toc58783397)

[Alternativas de solución evaluadas: 6](#_Toc58783398)

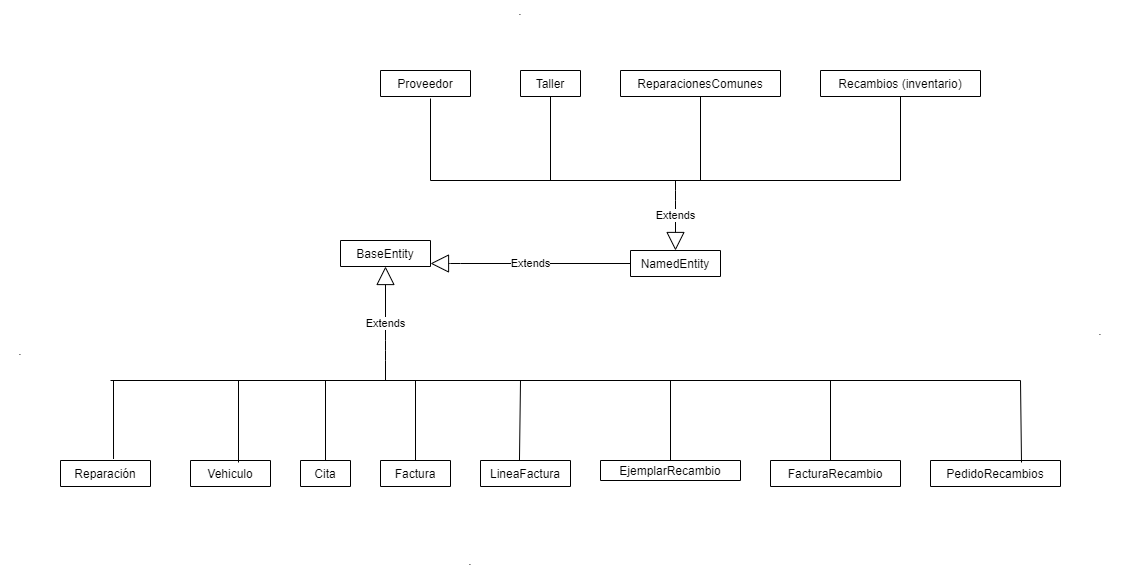
[Justificación de la solución adoptada 6](#_Toc58783399)

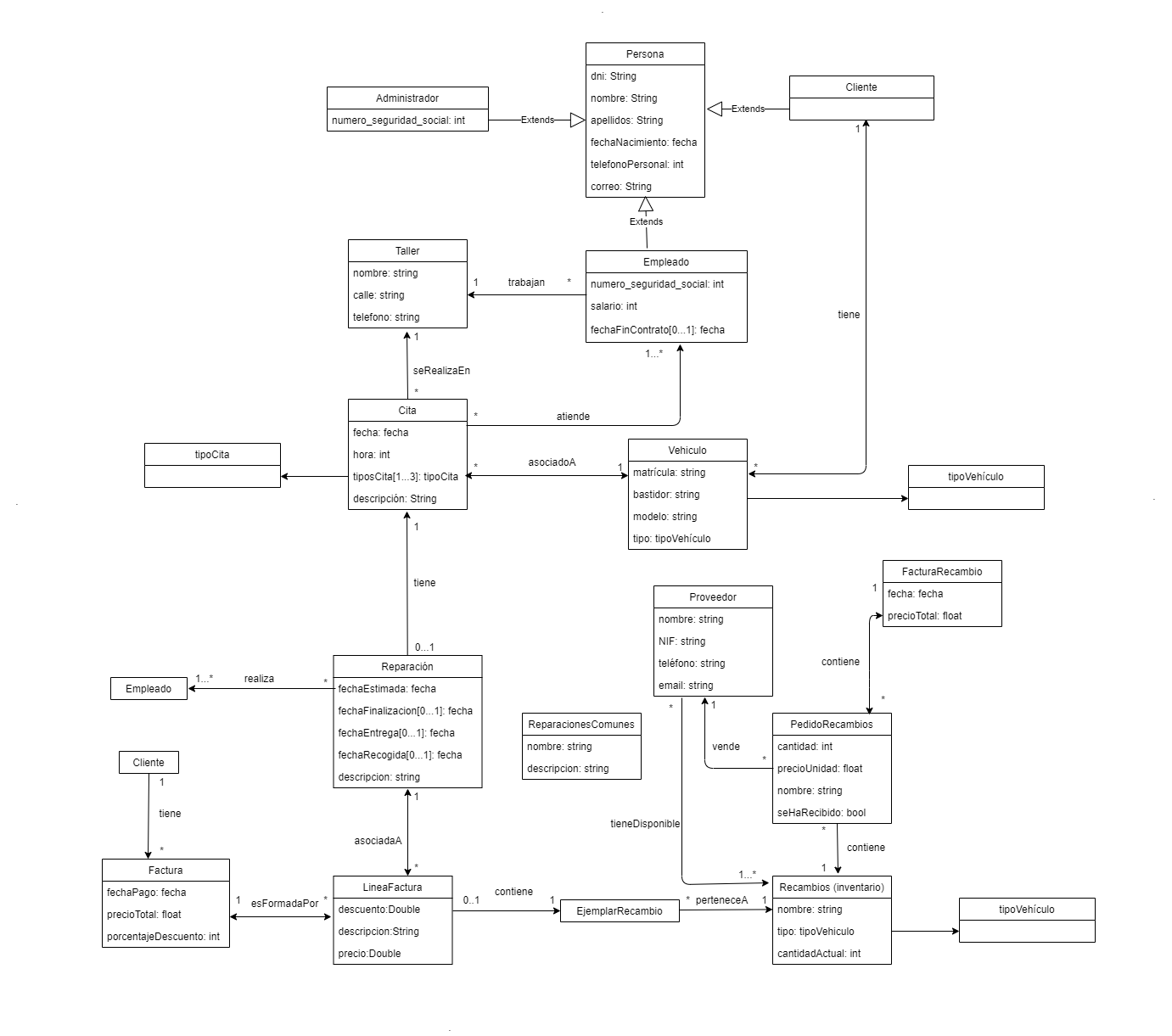
# Introducción

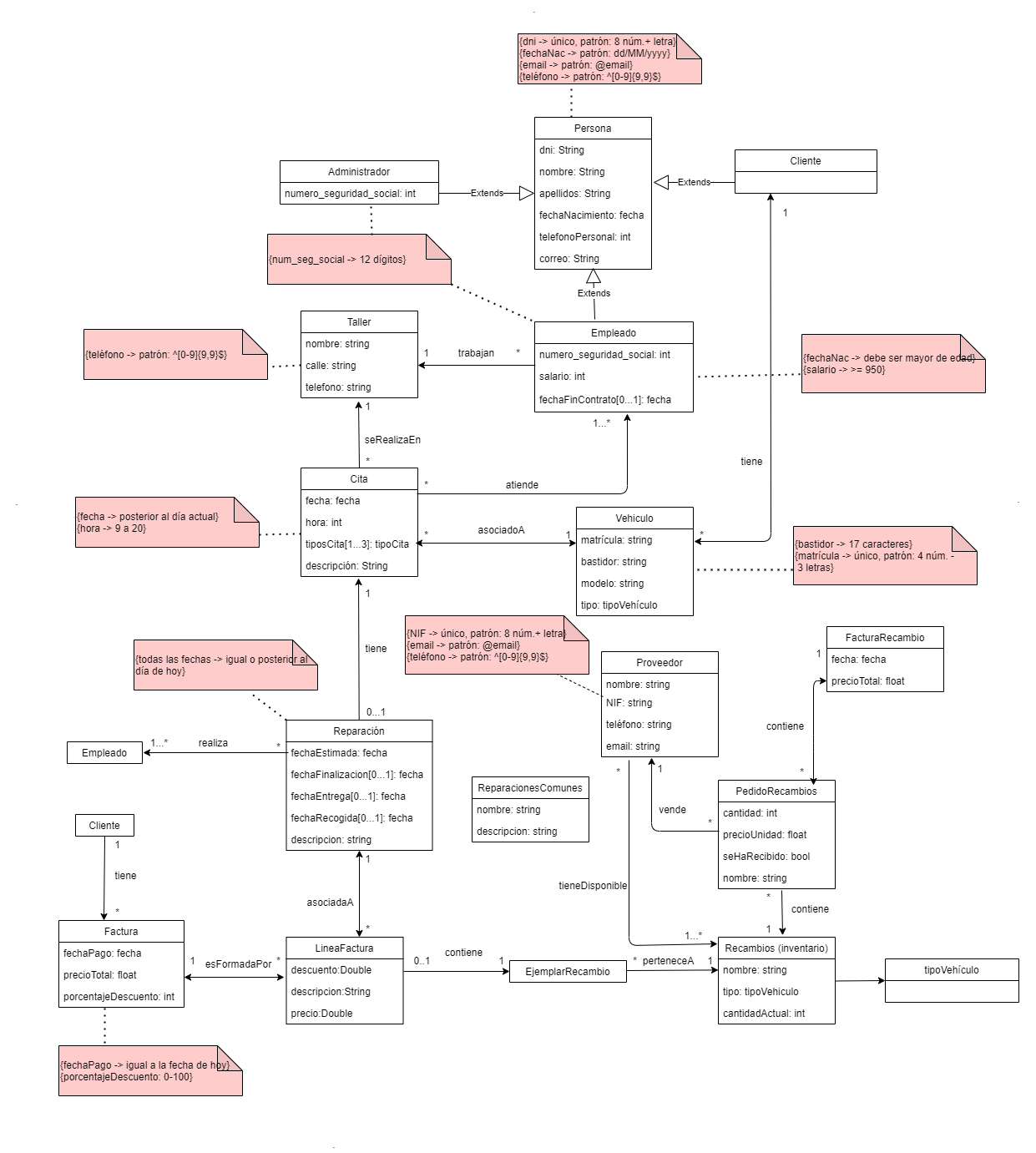
*El proyecto que vamos a desarrollar trata sobre un taller de reparación de vehículos único en Sevilla. Este taller cuenta con un administrador y varios empleados por cada local. Los empleados son los encargados de las reparaciones y de las supervisiones de los diferentes vehículos. El administrador se encarga de contratar a dichos empleados y de contactar con los distribuidores para la obtención de los repuestos necesarios. El sistema que vamos a desarrollar se encargará de la organización de las citas realizadas por los clientes, de la gestión económica del taller y el registro de todas las citas, de todos los clientes, de todos los empleados y de las piezas de repuesto obtenidas.*

# Diagrama(s) UML:

## Diagrama de Dominio/Diseño

**

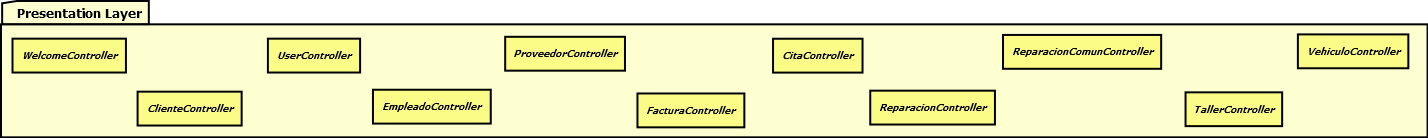
**

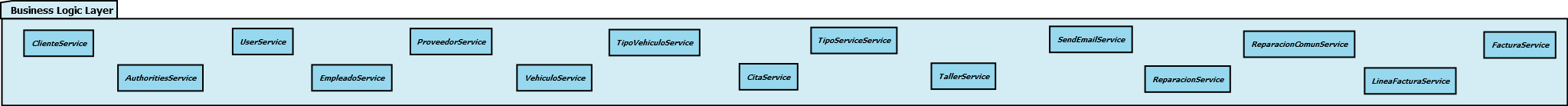
**

*Diagrama

Descripción generada automáticamente*

## Diagrama de Capas (incluyendo Controladores, Servicios y Repositorios)





***TipoCitaService***

Diagrama

Descripción generada automáticamente

# Patrones de diseño y arquitectónicos aplicados

## Patrón: MVC (Modelo-Vista-Controlador)

### Tipo: Arquitectónico.

### Contexto de Aplicación

### Clases o paquetes creados

Vista:

* src/main/webapp/WEB-INF

Controlador:

* src/main/java/org/springframework/samples/**petclinic**/web

Modelo:

* src/main/java/org/springframework/samples/**petclinic**/service
* src/main/java/org/springframework/samples/**petclinic**/model
* src/main/java/org/springframework/samples/**petclinic**/repository

### Ventajas alcanzadas al aplicar el patrón

* Soporte para múltiples vistas.
* Favorece la alta cohesión, el bajo acoplamiento y la separación de responsabilidades.

## Patrón: Inyección de dependencias.

### Tipo: de Diseño

### Contexto de Aplicación

A lo largo de toda la aplicación como consecuencia de integración con Spring Framework.

### Clases o paquetes creados

No se crean clases o paquetes como tal sino que se añaden anotaciones propias de Spring (@Component, @Repository, @Controller, @Service, @Configuration).

### Ventajas alcanzadas al aplicar el patrón

* Aseguramos que solo una instancia de clases indicadas como beans se crean y que todas las clases que las necesitan pueden acceder a estas.

## Patrón: Front Controller.

### Tipo: de Diseño

### Contexto de Aplicación

Se usa cuando se recibe una petición en la aplicación. El Front Controller busca el controlador apropiado para gestionar dicha petición.

### Clases o paquetes creados

Es generado por Spring por nosotros.

### Ventajas alcanzadas al aplicar el patrón

* Tenemos centralizado en un único punto la gestión de las peticiones.
* Aumentamos la reusabilidad de código.
* Mejoramos la gestión de la seguridad.

## Patrón: Modelo del dominio.

### Tipo: de Diseño

### Contexto de Aplicación

Se utiliza al traducir las entidades del modelo conceptual al código.

### Clases o paquetes creados

* src/main/java/org/springframework/samples/petclinic/model

### Ventajas alcanzadas al aplicar el patrón

* Permite implementar lógica de negocio compleja.
* Soportado por Spring Framework.

## Patrón: Capa de servicio.

### Tipo: de Diseño

### Contexto de Aplicación

Se utiliza para definir las interacciones entre el modelo y la capa de recursos con la capa de presentación. Funciona como intermediario que además añade lógica de negocio.

### Clases o paquetes creados

* src/main/java/org/springframework/samples/petclinic/service

### Ventajas alcanzadas al aplicar el patrón

* El acceso a datos está centralizado en un punto.
* Permite definir interfaces distintas según el tipo de usuario que accede.

## Patrón: Data Mapper.

### Tipo: de Diseño

### Contexto de Aplicación

Se utiliza al pasar datos entre las entidades del modelo y las filas de las tablas de la base de datos, son usados a través de los repositorios. También se usa invocando al EntityManager en ciertas pruebas de los servicios. Por ejemplo, en “ProveedorServiceTest”, cuando se intenta actualizar un proveedor con datos inválidos.

### Clases o paquetes creados

El Data Mapper (EntityManager) forma parte de JPA, luego solo hay que crearlo e invocarlo donde haga falta.

### Ventajas alcanzadas al aplicar el patrón

* Mantiene los objetos en memoria y la base de datos independientes entre sí y del propio Data Mapper.

## Patrón: Campo identidad.

### Tipo: de Diseño

### Contexto de Aplicación

Se utiliza en el modelo del dominio para asignar la clave primaria de las filas de la base de datos a un atributo del objeto en memoria con el que se corresponde.

### Clases o paquetes creados

No se crean nuevas clases o paquetes, sino que se incluye un nuevo atributo en cada entidad.

### Ventajas alcanzadas al aplicar el patrón

* Permite la asignación automática de identificadores.

## Patrón: Capa supertipo.

### Tipo: de Diseño

### Contexto de Aplicación

Se utiliza en el modelo del dominio. Una clase contiene el conjunto de atributos comunes de varias clases que la extienden. Todas las clases extienden BaseEntity pues necesitan el atributo id.

### Clases o paquetes creados

* BaseEntity.java
* NamedEntity.java
* Persona.java

### Ventajas alcanzadas al aplicar el patrón

* Agiliza la creación de entidades.

## Patrón: Repositorio

### Tipo: de Diseño

### Contexto de Aplicación

Se utiliza en la capa de recursos en conjunto con el patrón Data Mapper.

### Clases o paquetes creados

* src/main/java/org/springframework/samples/**petclinic**/repository

### Ventajas alcanzadas al aplicar el patrón

* Aparenta que se accede a un conjunto de objetos en memoria en lugar de filas de tablas de la base de datos.

## Patrón: Eager Loading.

### Tipo: de Diseño

### Contexto de Aplicación

Se usa en las asociaciones entre entidades del tipo Many/OneToOne y en la asociación entre “Cita” y “TipoCita”.

### Clases o paquetes creados

No se crean nuevas clases o paquetes.

### Ventajas alcanzadas al aplicar el patrón

* El rendimiento no se ve afectado posteriormente pues se inicializan todos los datos consultados al principio.

## Patrón: Lazy Loading.

### Tipo: de Diseño

### Contexto de Aplicación

Se usa en las asociaciones entre entidades del tipo Many/OneToMany.

### Clases o paquetes creados

No se crean nuevas clases o paquetes.

### Ventajas alcanzadas al aplicar el patrón

* La carga inicial de datos es menor pues no se cargan todos desde el primer momento.
* El uso de memoria es menor.

# Decisiones de diseño

## Decisión 1: Relación entre citas, reparaciones y facturas.

### Descripción del problema:

Como grupo nos dimos cuenta de que había una incongruencia en el diagrama UML: se podían escoger varios tipos de citas, pero no había varios tipos de reparaciones. Por ejemplo, se podía escoger una cita para los neumáticos y frenos, pero la reparación asociada solo incluiría a uno de los dos tipos en lugar de ambos.

### Alternativas de solución evaluadas:

Alternativa 1.a: Asociar varias reparaciones a una misma cita.

**Ventajas**:

* Hay que modificar poco el diagrama UML.

**Inconvenientes**:

* Muchos datos duplicados.
* Dificulta al empleado manipular o editar las reparaciones (se pueden generar incongruencias entre los datos duplicados).
* Todas las reparaciones asociadas a una cita tienen una misma factura asociada, por lo que no se pueden saber los gastos asociados a cada una de las reparaciones. Habría que anotarlo en la descripción de la reparación, lo que es más tedioso para el empleado.

Alternativa 1.b: Asociar una única reparación a cada cita y asociar una o varias líneas de factura a dicha reparación, que contendrán datos sobre los costes. Por ejemplo, para una cita para neumáticos y frenos, se asociaría una única reparación, pero tendría dos líneas de factura asociadas, una recoge los gastos de los neumáticos y otra los gastos de los frenos.

**Ventajas:**

* Toda la información queda asociada a una misma reparación, lo que facilita la manipulación por parte del empleado.
* Se conocen los gastos asociados a cada una de las reparaciones realizadas (en el ejemplo anterior, el cambio de neumáticos cuesta X y los frenos cuesta Y; con la alternativa anterior solo se podía saber el coste total en la factura).
* Permite asociar recambios a cada una de las reparaciones (en el ejemplo anterior, los neumáticos con el cambio de neumáticos).

**Inconvenientes:**

* Hay que realizar múltiples cambios en el diagrama UML.

### Justificación de la solución adoptada

Como se puede observar, la segunda alternativa es mejor a pesar de que implica mayor carga de trabajo. Por ello escogimos la segunda opción.

## Decisión 2: Enviar emails desde la aplicación.

### Descripción del problema:

Como grupo nos gustaría poder enviar emails a los clientes de la aplicación para avisarles de distintos sucesos. Por ejemplo: avisar de cuando la reparación del vehículo del cliente ha finalizado o avisar de que su cita se ha cancelado por motivo del COVID, etc.

### Alternativas de solución evaluadas:

Alternativa 2.a: Utilizar la API de Gmail.

**Inconvenientes**:

* Había que trabajar y analizar una API con la que no se había trabajado antes, lo que conlleva más carga de trabajo.

Alternativa 2.b: Utilizar la librería JavaMail integrada en Spring Boot.

**Ventajas:**

* Es soportado por Spring Framework de manera nativa.
* No requiere instalar nada.
* No requiere gestionar la autenticación con Oauth2.0, sino que incluye métodos para la identificación simples de utilizar.

**Inconvenientes:**

* No se había trabajado antes con esta librería.

### Justificación de la solución adoptada

Como la segunda opción es una alternativa para el envío de emails, elegimos esta debido a que ya se encuentra integrada en Spring Framework.

## Decisión 3: Acceso a datos por parte de usuarios.

### Descripción del problema:

Como grupo queremos que cada cliente de la aplicación solo pueda acceder a sus datos personales (vehículos, citas, reparaciones, etc) y no pueda acceder a la información de otros para mantener la privacidad y seguridad de los usuarios.

### Alternativas de solución evaluadas:

Alternativa 3.a: Utilizar el nombre de usuario en la URL para acceder a información concreta. Si no se incluye nombre de usuario, entonces se accede a la vista de administrador que muestra la información de todos los clientes (por ejemplo, los vehículos de todos los clientes).

Por ejemplo:

* Si un cliente logueado con su usuario hace click en la pestaña “Vehículos”, este le llevará a “vehículos/listadoVehiculos/{suUsername}” y le mostrará solo sus vehículos.
* Si un administrador logueado hace click en la pestaña “Vehículos”, este le llevará a “vehículos/listadoVehiculos” y le mostrará todos los vehículos.
* Si un cliente sin haberse logueado hace click en la pestaña “Vehículos”, se le pedirá antes que haga login.

**Ventajas**:

* Modificando la configuración de seguridad, se puede establecer que solo se pueda acceder a la URL sin username solo si se está logueado con un usuario con permisos de administrador.

**Inconvenientes**:

* Requiere duplicar método en controladores (usando el username y sin usar el username en el URL).
* Modificando la configuración de seguridad solo se puede establecer que el cliente debe estar logueado para acceder a la URL con un username, pero puede acceder a la información de otros usuarios si conoce su username (por ejemplo, tenemos 2 usuarios que son clientes, “fraborcar” y “jesfunrud”, el usuario “fraborcar” puede acceder a los vehículos de “jesfunrud” si pone en el buscador “vehículos/listadoVehiculos/jesfunrud”).

Alternativa 3.b: Obtener el nombre del usuario logueado desde código y mostrarle una información u otra en función del nombre de usuario y sus permisos.

Por ejemplo:

* Si un cliente logueado con su usuario hace click en la pestaña “Vehículos”, este le llevará a “vehículos/listadoVehiculos” y le mostrará solo sus vehículos.
* Si un administrador logueado hace click en la pestaña “Vehículos”, este le llevará a “vehículos/listadoVehiculos” y le mostrará todos los vehículos.
* Si un cliente sin haberse logueado hace click en la pestaña “Vehículos”, se le pedirá antes que haga login.

**Ventajas:**

* Reduce el código en los controladores.
* Un cliente no puede acceder a los datos de otros clientes.

**Inconvenientes:**

* Hay que hacer comprobaciones adicionales en los controladores.

### Justificación de la solución adoptada

Escogemos la segunda alternativa pues soluciona el problema fundamental que teníamos sobre la seguridad.