

# Documentación de la práctica 4

Sergio Quijano Rey

Fernando Valdés Navarro  
Ignacio Carvajal Herrera

Carlos Corts Valdivia

08/06/2022

## Resumen global del sistema

Antes de comenzar con los detalles de la aplicación, pensamos que es adecuado hacer una introducción sobre los objetivos globales de esta.

Nuestra aplicación sirve como punto de venta para una hipotética marca de coches, o taller de coches. En la parte del móvil (*Flutter*), tendremos una *app* de compra de automóviles. En dicha compra, podremos personalizar las distintas partes del coche (llantas, color, tapicería y extras), lo que supondrá cierto aumento en el precio.

En la aplicación web, permitiremos a los dueños del taller, marca de coches o parecidos, administrar modelos de coches o ciertas partes de la personalización.

Además, la aplicación web escrita en *Ruby on Rails* expondrá una **API REST** que será consumida por la aplicación móvil. De esta forma, la administración realizada desde la web tendrá un impacto directo, sin tener que modificar el código, en el catálogo que los usuarios tienen disponible desde el móvil.

Por tanto, estaremos manejando dos bases de códigos. En ambas, hemos usado *Github* como sistema de control de versiones. Al tener dos bases de códigos claramente separadas, hemos trabajado en dos repositorios:

1. Repositorio para la aplicación *Flutter* que se puede encontrar en <https://github.com/fervalnav/CarConfigurator>
2. Repositorio para la aplicación *web*, que se puede encontrar en <https://github.com/SergioQuijanoRey/CarConfiguratorWeb>

En ambos repositorios hemos trabajado de la misma forma. Hemos usado *issues* para marcar las tareas pendientes. Hemos trabajado con distintas ramas, que hemos *mergeado* contra **master** | **main** usando *pull requests*.

En esta práctica, se pedía que usáramos una herramienta como *Jira*. Sin embargo, optamos por los proyectos de *Github* por los siguientes motivos:

1. En las prácticas anteriores ya veníamos usando esta *feature* de *Github*, y por lo tanto, ya estábamos familiarizados con esta herramienta, como se puede comprobar en <https://github.com/fervalnav/CarConfigurator/projects/3>
  - Notar que las antiguas *cards* ya no se encuentran en el proyecto, porque al empezar con la práctica 4 las hemos borrado, para partir desde cero
2. El uso de *issues* como tarjetas en el *Kanban* es directo, y por lo tanto, mucho más cómodo a nuestro parecer (desconociendo si otras herramientas como *Jira* permiten trabajar cómodamente con *issues*)
3. Al tener una lista de *checkboxes* en una *issue*, podemos convertir cada *checkbox* en una *issue* propia. Con esto podemos:

- Dividir las tareas en subtareas, de forma que cuando estén en la columna *TO-DO* se puedan completar en menos de dos horas
- Hacer un seguimiento del porcentaje de completado de tareas más complejas

Como se pide en el guión de la práctica, a continuación mostramos una captura de pantalla en la que se muestra el uso de esta herramienta:

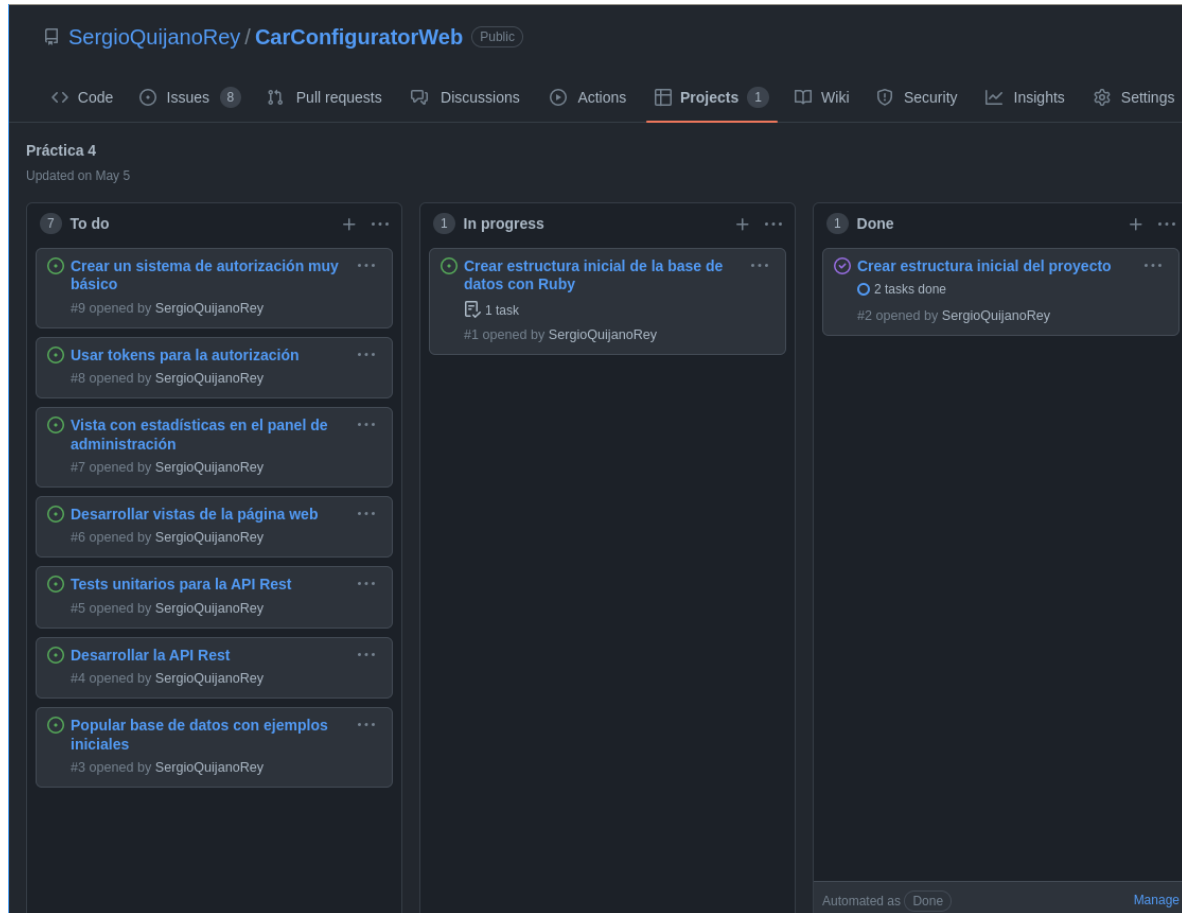


Figure 1: Ejemplo de uso de los proyectos de Github. Se puede ver claramente la ventaja de usar directamente esta herramienta, por su integración directa con las *issues* de Github, que ya veníamos usando para marcar tareas, definir las ramas que vamos a crear, ...

Como se puede comprobar en los dos repositorios, los cuatro integrantes hemos trabajado en las dos bases de código. Así hemos podido avanzar más rápido, pues es claro que ambos proyectos son inter-dependientes (ie. para poder conectar *Flutter* con la API REST, primero necesitamos tener desarrollada dicha API).

Y en último lugar, como estamos trabajando con dos bases de código, que aunque estén conectadas, se han diseñado en paralelo (basta con tener en cuenta que la aplicación en *Flutter* la hemos diseñado y codificado, hasta cierto punto, mucho antes de diseñar la *app web* y *API REST*), deberemos mostrar los dos diseños por separado. Esto lo haremos en dos secciones claramente separadas, una para cada base de código.

# Aplicación *Flutter*

## Notas iniciales

Como es lógico, hemos partido del código de la práctica anterior. Recordad que habíamos aplicado un patrón repositorio para los distintos elementos que necesitaban de persistencia, acompañado de una patrón *singleton* para hacer fácil su acceso sin tener que inyectar dependencias como parámetros en toda la cascada de métodos.

Es por esto que el cambio necesario para esta práctica es muy sencillo, basta con implementar los repositorios adecuados haciendo uso de la *API REST* desarrollada.

Además, los *tests* serán también sencillos, puesto que podemos usar los que ya teníamos en las clases que implementaban los repositorios en memoria. Podemos considerar estas clases como *mocks* de las clases que interactúan con la base de datos. Puesto que no nos interesa hacer las pruebas tocando directamente la base de datos que tenemos en producción (*clados.ugr.es* en nuestro caso).

Esto último se podría haber hecho usando una librería que *mockease* el acceso a la base de datos, pero como ya lo teníamos programado, consideramos que esto no es ni necesario ni interesante.

## Requisitos funcionales

A partir de los requisitos funcionales de la práctica pasada, desarrollamos estos en mayor detalle, en la siguiente tabla:

### 1. Gestión de configuraciones

Nombre   Id. RF	1.1. Comenzar una configuración desde cero
Entrada	Acción del usuario de crear una nueva configuración
Procesamiento	Se crea una configuración por defecto para mostrarla
Salida	Se muestra la vista con la nueva configuración por defecto. Se permite a partir de esta vista configurar el coche.
Nombre   Id. RF	1.2. Borrar una configuración creada
Entrada	Identificador de la configuración a borrar. Debe existir ya en el sistema
Procesamiento	Se borra la configuración del sistema de persistencia de datos (en nuestro caso, base de datos <i>mysql</i> )
Salida	Mensaje de éxito o error según la operación se realice o no correctamente
Nombre   Id. RF	1.3 Modificar configuración ya existente
Entrada	Identificador de la configuración existente y sus nuevos datos
Procesamiento	Se modifican los datos de la configuración en la persistencia de datos

Salida	Mensaje de error o éxito según como se haya desarrollado la operación
<hr/>	
Nombre   Id. RF	1.4 Almacenar una configuración que se esté modificando actualmente
Entrada	Identificador de la configuración que se está modificando, así como los datos seleccionados actualmente
Procesamiento	Se crea una nueva entrada en la base de datos si no hay una configuración con el identificador, o se modifican los datos de la configuración del mismo identificador
Salida	Mensaje de error o éxito según como se haya desarrollado la operación
<hr/>	
Nombre   Id. RF	1.5 Previsualizar una configuración ya existente
Entrada	Identificación de la configuración a visualizar
Procesamiento	Ninguno
Salida	Se muestran los datos, visualmente, de la configuración seleccionada, al usuario
<hr/>	

## Partes interesadas y sus preocupaciones

Las partes interesadas que identificamos son:

1. Arquitecto: (máster en ingeniería informática) es el director gerente de una pequeña empresa especializada en el desarrollo de aplicaciones web y móviles. Será el arquitecto encargado de elaborar la descripción arquitectónica y supervisar el proceso de desarrollo
  - En este caso, nosotros cuatro seremos los arquitectos de la aplicación
2. Cliente: Dueño de la empresa de coches que contrata nuestros servicios para el desarrollo de las aplicaciones.
3. Desarrolladores: equipo de desarrolladores (nuestros cuatro) que se encargan del desarrollo de las aplicación web y móvil. Se comunican entre ellos pues hay una parte en común de desarrollo para las dos aplicaciones a desarrollar
4. Técnico de pruebas: realiza las pruebas de los sistemas software desarrollados.
  - En este caso, también seremos nosotros los técnicos de pruebas
5. Usuarios: existirán tres tipos de usuarios
  - Clientes de la app móvil
  - Gestor de catálogo y ventas: usará la app web para introducir nuevos modelos. Aunque pertenezca más al otro sistema, dejamos esto aquí por claridad

La empresa de venta de coches tiene que mantener una gran reputación asociada a su marca. Por tanto, los siguientes objetivos son importantes:

- Los valores de la empresa deben reflejarse en las aplicaciones desarrolladas
- Las aplicaciones deben presentar una cara “humana” al cliente
- Las aplicaciones deben ser sencillas de usar, accesible para todas las personas
- Las aplicaciones deben ser rápidas y responsivas

Los usuarios expresan las siguientes inquietudes:

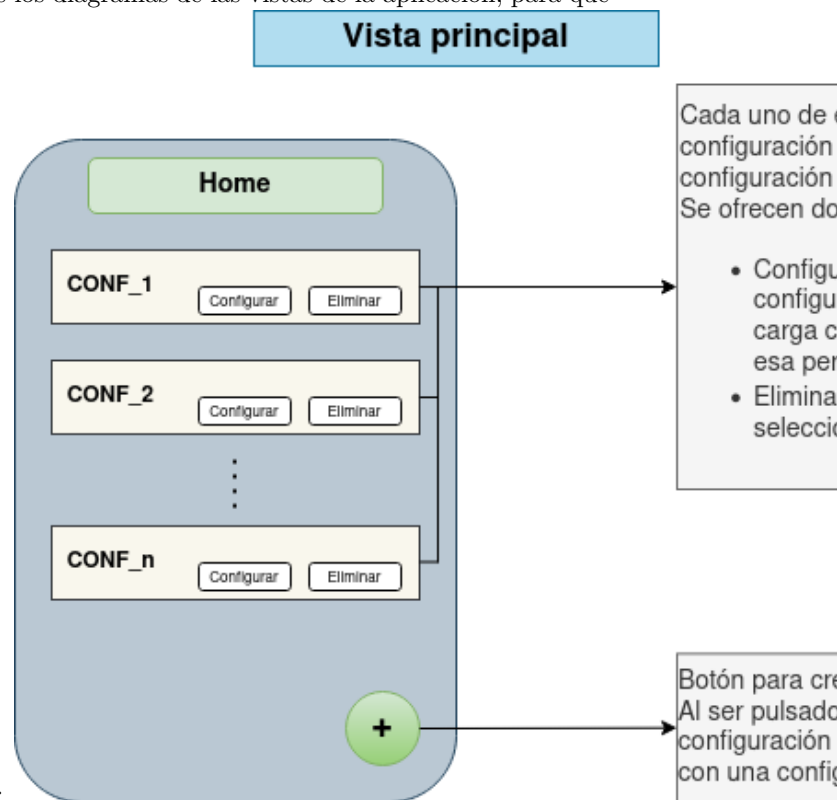
- Temo que tenga problemas de accesibilidad (paleta de colores no adecuada para la vista, letras muy pequeñas, botones pequeños. . .)
- Quiero que el sistema gestione distintos tipos de usuario y que analice los datos del usuario para facilitar la navegación / compra a través del sistema
- Temo que la selección de las distintas variantes de las componentes de una configuración sea compleja y poco usable

Todo esto tendrá un impacto directo en los requisitos funcionales y no funcionales del sistema

## Diagrama que muestra la arquitectura

### Diagramas de vistas

Como hacíamos en la práctica anterior, mostramos los diagramas de las vistas de la aplicación, para que



sea más fácil comprender el funcionamiento de esta:

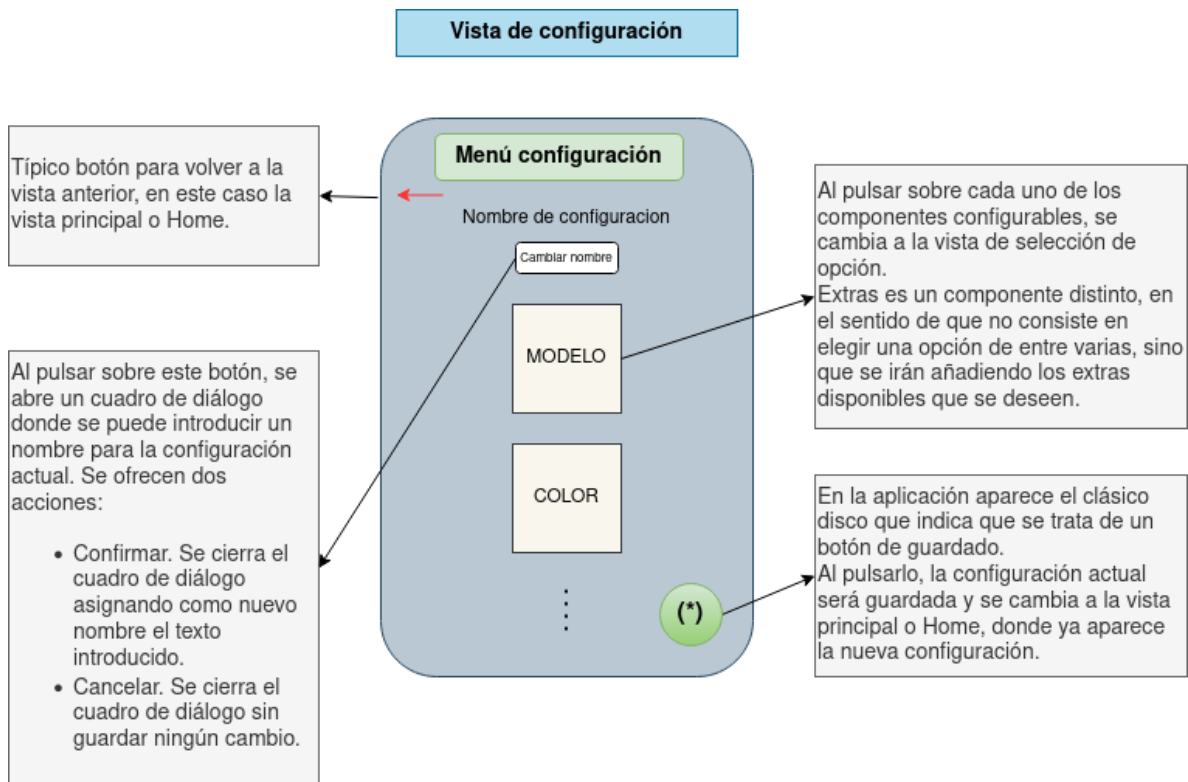


Figure 2: Vista de selección de configuración

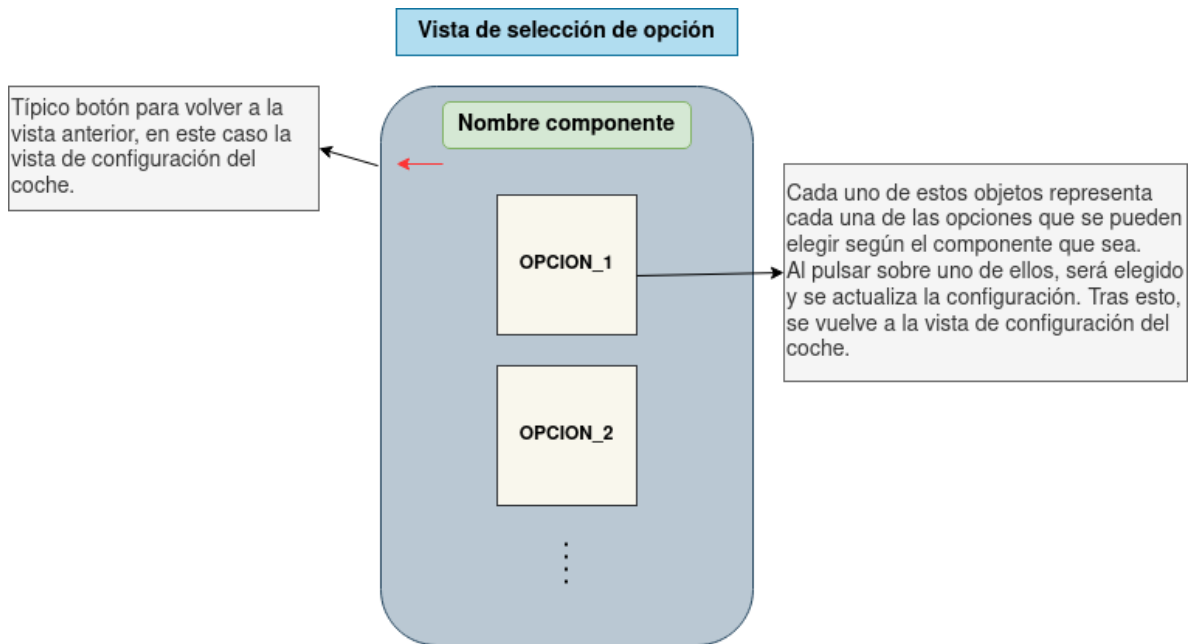


Figure 3: Vista de selección de una componente

## Diagramas de clases y de *Widgets*

Empezamos con el diagrama de clases:

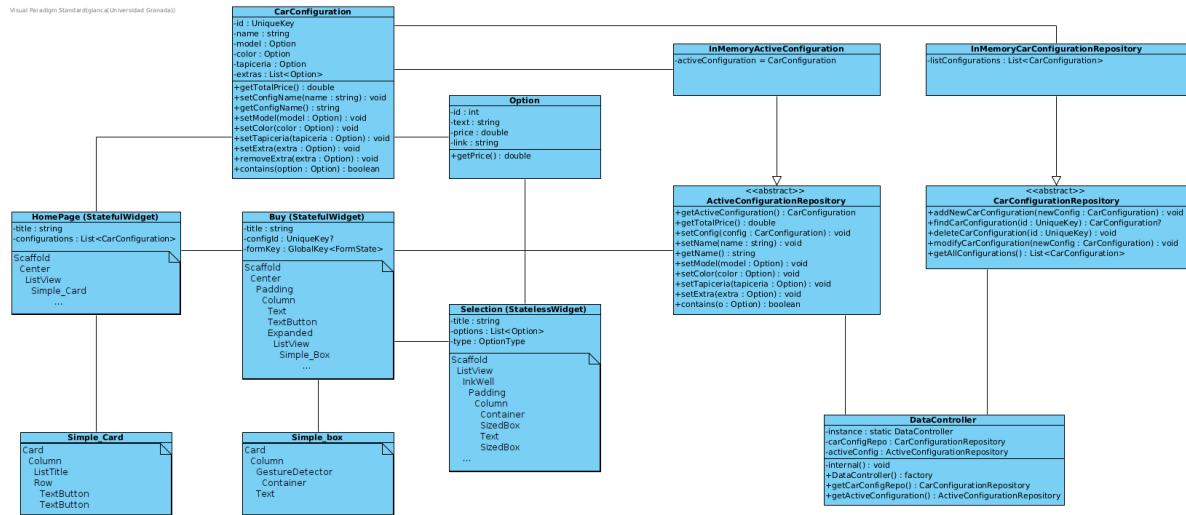


Figure 4: Diagrama de clases de la aplicación

Notar que es fácil la modificación, añadiendo las clases que implementan el patrón repositorio (que vienen dados por clases abstractas) usando la *API REST*.

Ahora mostramos el diagrama de *Widgets*:



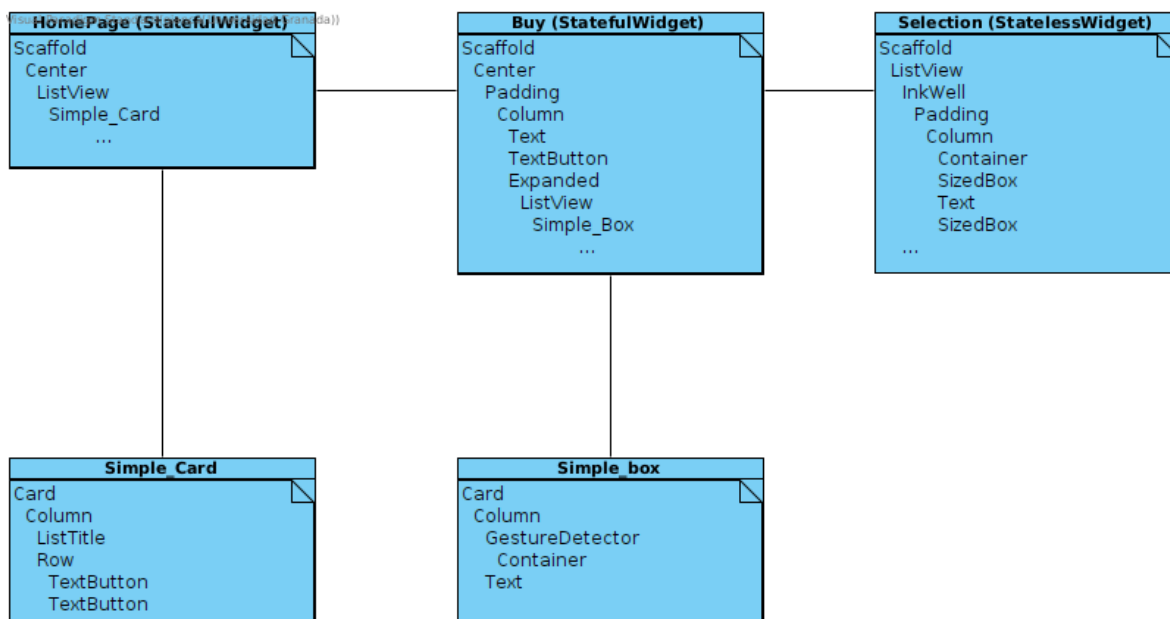


Figure 5: Diagrama de Widgets de la aplicación

## Criterios de calidad a partir de los requisitos no funcionales

Los criterios de calidad, que vamos a expresar en términos de requisitos funcionales, son:

1. Deben mantenerse protegidos los datos de los usuarios.
2. Configurable
  - Permitir añadir/eliminar/modificar partes configurables, opciones para una parte, métodos de pago guardados, etc.
3. Facilidad de uso
  - Buscamos que la aplicación sea fácil de navegar, intuitiva, destinada a un usuario sin conocimiento experto
4. Accesibilidad
  - El diseño de la *app* está pensada para que sea fácil de visualizar, con buenos contrastes
  - Con fuentes sencillas de visualizar y de buen tamaño
  - Pensando en facilitar el uso a personas con ciertos problemas visuales
5. Cumplimiento con las normativas legales vigentes
  - Por ejemplo, con algunas de las más importantes, como la *Normativa Europea de ley de Protección de Datos*
  - Por el alcance de nuestra aplicación, y por no estar pensada en lanzarse al mercado, no seguimos procedimientos usuales para garantizar estos requisitos (como podría ser el caso de auditorías externas para poder guardar datos de pago, uso de pasarelas de pago externas, ...)

La mayoría de estos requisitos no pueden ser comprobados automáticamente, y por ello debemos verificarlos manualmente antes de lanzar una nueva versión de la aplicación. Por ejemplo, usando la *app* para comprobar algunos de los requisitos, realizando comprobaciones de seguridad en bases de datos...

## Diseño de las pruebas

### Pruebas unitarias

Notar que en todos los tests o bien tenemos una función tipo **warmup** que inicializa ciertos datos con los que vamos a jugar, o bien definimos una lista con los datos que vamos a manipular y realizar comprobaciones.

#### 1. Tests sobre la configuración activa actual

Descripción de la prueba	Tomar la configuración activa
Datos requeridos	Ninguno, la configuración activa se toma sin parámetros
Condiciones a cumplir	Se toma una configuración activa sin problemas, y el nombre es el nombre por defecto

Descripción de la prueba	Hacemos <b>set</b> del modelo, en la configuración activa
Datos requeridos	La configuración activa, nuevos datos para el modelo
Condiciones a cumplir	El modelo de la configuración activa tiene los datos que hemos pasado

Descripción de la prueba	Hacemos <b>set</b> del color, en la configuración activa
Datos requeridos	La configuración activa, nuevos datos para el color
Condiciones a cumplir	El color de la configuración activa tiene los datos que hemos pasado

Descripción de la prueba	Hacemos <b>set</b> del tapicería, en la configuración activa
Datos requeridos	La configuración activa, nuevos datos para el tapicería
Condiciones a cumplir	El tapicería de la configuración activa tiene los datos que hemos pasado

Descripción de la prueba	Añadimos un extra en la configuración activa
Datos requeridos	El extra que queremos añadir
Condiciones a cumplir	El extra se debe añadir a la configuración que tenemos activa

Descripción de la prueba	El cálculo del precio debe ser correcto
Datos requeridos	Se toma la configuración activa, y a mitad del test se cambia el modelo

Condiciones a cumplir	El precio por defecto debe ser el correcto. Cuando cambiamos de modelo, el nuevo precio debe ser correcto
-----------------------	---

## 2. Tests del repositorio de configuraciones de un coche

- Disponemos de la función `create_basic_repo` como warmup

Descripción de la prueba	Los valores se almacenan correctamente
Datos requeridos	Repositorio creado por <code>create_basic_repo</code>
Condiciones a cumplir	Debe tener los elementos correctos

Descripción de la prueba	La funcionalidad de añadir funciona como se espera
Datos requeridos	Datos para añadir a la configuración
Condiciones a cumplir	Se debe almacenar bien la nueva configuración. Si la intentamos volver a añadir, como ya está almacenada, no debe almacenarse de nuevo

Descripción de la prueba	La funcionalidad de buscar configuraciones funciona como se espera
Datos requeridos	Identificador de la configuración que se busca
Condiciones a cumplir	Si el identificador es bueno, se debe encontrar bien la configuración. Si la configuración no es buena, se debe retornar un <code>null</code>

Descripción de la prueba	La funcionalidad de eliminar configuraciones funciona como se espera
Datos requeridos	Identificador de la configuración a eliminar
Condiciones a cumplir	La configuración se elimina con éxito

Descripción de la prueba	Borrar dos veces la misma configuración solo borra en la primera vez
Datos requeridos	Identificador de la configuración que se borra dos veces
Condiciones a cumplir	La primera vez, se borra con éxito. La segunda vez, al no encontrarse en el sistema, no se realiza ningún cambio en la persistencia de datos

## 3. Test de la clase que usamos para modelar las opciones

Descripción de la prueba	Crear una función actúa como debería
Datos requeridos	Datos de la opción que se crea
Condiciones a cumplir	Se crea una opción con los datos correctos

Descripción de la prueba	El precio de una opción es el esperado
Datos requeridos	Datos de la opción creada
Condiciones a cumplir	El precio debe coincidir con el esperado

### Tests de widgets

Descripción de la prueba	El botón de eliminar debe eliminar la configuración correcta
Datos requeridos	Identificador de la configuración que queremos eliminar
Condiciones a cumplir	Tras simular las acciones necesarias, esa configuración ya no se encuentra

Descripción de la prueba	El botón de configurar debe cambiar a la vista correcta
Datos requeridos	Identificador de la configuración que queremos modificar
Condiciones a cumplir	Se debe pasar a la vista de configuración, con los datos asociados a esa configuración

### Tests de integración

Descripción de la prueba	Se realiza una configuración desde cero correctamente
Datos requeridos	La secuencia de acciones que se realiza
Condiciones a cumplir	Se pulsa el botón de configurar de cero. Se guardan los datos. Se vuelve a la vista principal donde se encuentra la configuración, con datos por defecto, guardada en el sistema.

# Aplicación *Ruby on Rails*

## Notas iniciales

## Requisitos funcionales

```
\item \textbf{RF 3. Gestión del catálogo:} este requisito queda pospuesto para futuras versiones, ya
\begin{itemize}
  \item \textbf{RF 3.1.} Añadir nueva parte configurable.
  \item \textbf{RF 3.2.} Eliminar parte configurable.
  \item \textbf{RF 3.3.} Modificar parte configurable (nombre, descripción, etc.).
  \item \textbf{RF 3.4.} Añadir nueva opción de una parte configurable.
  \item \textbf{RF 3.5.} Eliminar una opción de una parte configurable.
  \item \textbf{RF 3.6.} Modificar una opción de una parte configurable (nombre, descripción, fotos)
\end{itemize}
```

Nombre   Id. RF
Entrada
Procesamiento
Salida

## Partes interesadas y sus preocupaciones

Las partes interesadas que identificamos son:

1. Arquitecto: (máster en ingeniería informática) es el director gerente de una pequeña empresa especializada en el desarrollo de aplicaciones web y móviles. Será el arquitecto encargado de elaborar la descripción arquitectónica y supervisar el proceso de desarrollo
  - En este caso, nosotros cuatro seremos los arquitectos de la aplicación
2. Cliente: Dueño de la empresa de coches que contrata nuestros servicios para el desarrollo de las aplicaciones.
3. Desarrolladores: equipo de desarrolladores (nuestros cuatro) que se encargan del desarrollo de las aplicación web y móvil. Se comunican entre ellos pues hay una parte en común de desarrollo para las dos aplicaciones a desarrollar
4. Técnico de pruebas: realiza las pruebas de los sistemas software desarrollados.
  - En este caso, también seremos nosotros los técnicos de pruebas
5. Usuarios: existirán tres tipos de usuarios
  - Clientes de la app móvil
  - Gestor de catálogo y ventas: usará la app web para introducir nuevos modelos. Aunque pertenezca más al otro sistema, dejamos esto aquí por claridad

La empresa de venta de coches tiene que mantener una gran reputación asociada a su marca. Por tanto, los siguientes objetivos son importantes:

- Los valores de la empresa deben reflejarse en las aplicaciones desarrolladas
- Las aplicaciones deben presentar una cara “humana” al cliente
- Las aplicaciones deben ser sencillas de usar, accesible para todas las personas
- Las aplicaciones deben ser rápidas y responsivas

Los usuarios expresan las siguientes inquietudes:

- Temo que tenga problemas de accesibilidad (paleta de colores no adecuada para la vista, letras muy pequeñas, botones pequeños...)
- Quiero que el sistema gestione distintos tipos de usuario y que analice los datos del usuario para facilitar la navegación / compra a través del sistema

- Temo que la selección de las distintas variantes de las componentes de una configuración sea compleja y poco usable

Todo esto tendrá un impacto directo en los requisitos funcionales y no funcionales del sistema

### Diagrama que muestra la arquitectura

A partir de estos diseños, ha sido fácil generar el código, usando principalmente la instrucción **rails generate scaffold** para escribir la cantidad mínima de código necesaria, siguiendo así un **Desarrollo Dirigido por Modelos** o *MDD* como hemos visto en teoría.

### Criterios de calidad a partir de los requisitos no funcionales

#### Diseño de las pruebas