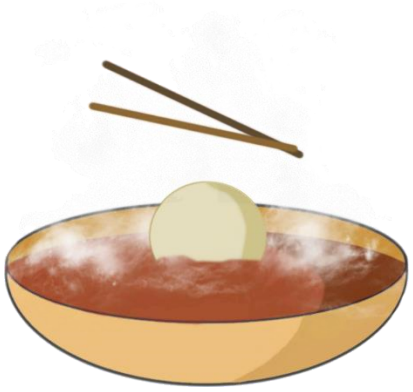


COCINA FÁCIL

Memoria del Proyecto Final de Ingeniería Web

Bruned Alamán, Jorge	*Esain Lizoain, Ainara*
Boychuk, Vladyslav	*Sotés Garciandia, Carla*

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Tabla de Contenidos

1. Introducción

1.1. Presentación del grupo de trabajo

Nuestro grupo de trabajo está formado por Carla Sotés, Vladyslav Boychuk, Ainara Esain y Jorge Bruned. Es un grupo de trabajo algo peculiar dado que uno de los miembros, por un lado ya tenía algo de experiencia en el campo del desarrollo web, pero por otro lado no ha podido acudir a las sesiones presenciales, de modo que hemos tenido que coordinarnos utilizando herramientas como *Discord* y *WhatsApp* (además del repositorio en *Gitlab*).

De hecho, tal y como explicaremos más adelante, esto ha desembocado en el desarrollo de dos implementaciones distintas de forma paralela; por una lado, Carla, Vlad y Ainara han desarrollado una página web (a partir de ahora, "Implementación A"), y por el otro lado Jorge ha desarrollado otra ("Implementación B"). Esto, como más adelante detallaré en la sección de trabajo en equipo es porque, según el criterio de mis compañeros, mi implementación tenía elementos quizá algo complejos de entender sin haber desarrollado ningún proyecto de este tipo anteriormente. Sin embargo, cabe destacar que, durante el periodo en el que trabajamos de forma conjunta, fuimos capaces de coordinarnos correctamente gracias al repositorio común y las herramientas de comunicación mencionadas anteriormente.

1.2. Objetivos del proyecto

El proyecto trata del desarrollo de una web de temática libre en la que plasmar los conocimientos adquiridos durante el semestre en la asignatura de Ingeniería Web; no solo conocimientos de programación, como los adquiridos en *HTML*, *CSS*, *JavaScript*, *JQuery*, *PHP*, *MySQL*..., sino también en aspectos tan importantes como la usabilidad, accesibilidad, posicionamiento SEO, rendimiento, integración continua y un largo etcétera.

2. Especificación de requisitos

Por desgracia, Jorge no pudo ser partícipe de la especificación inicial de requisitos por no estar presente en ese momento, pero a posteriori, ha contribuido a refinar estos requisitos, en parte gracias a la experiencia adquirida en otros proyectos web.

2.1. Documento de requisitos

El documento de requisitos, redactado por el grupo *Alpha*, está adjunto en la sección de "Anexos".

2.1.1. Requisitos funcionales

Algunos de los requisitos funcionales más destacables son:

A. El sistema deberá ser capaz de mostrar recetas, registrar usuarios, crear nuevas recetas.
B. El servidor deberá tener disponible una base de datos relacional MySQL, usada para almacenar los datos persistentemente.
C. El sistema deberá posibilitar el registro de nuevos usuarios. Estos se almacenarán en el servidor.
D. El sistema deberá posibilitar que usuarios previamente registrados accedan a su cuenta.
E. El sistema deberá presentar en alguna de sus secciones las recetas incluidas en la base de datos.
F. El sistema deberá categorizar las recetas según criterio (mediante etiquetas).
G. El sistema deberá implementar una barra de búsqueda para buscar las recetas.
H. Un usuario registrado y *logueado* deberá ser capaz de subir su propia receta al sitio web.
I. Un usuario registrado podrá subir imágenes, ya sea en la foto de perfil o en alguna receta.

2.1.2. Requisitos no funcionales

A. Un usuario medio debe poder ver recetas en menos de 5 clics.
B. Un usuario medio debe poder ser capaz de registrarse en menos de 30 minutos desde su primer acceso.
C. Un usuario registrado debe ser capaz de subir una receta básica con el menor número de clics posibles desde la página principal.
D. Las imágenes deben contener texto alternativo.
E. Los *inputs* deben estar asociados a un *label*, o en su caso a un *aria-label*.

F. Se deberá considerar en la implementación la defensa ante ataques *SQLInjection*.

2.2. Prototipos y *wireframes* iniciales

En cuanto al diseño de la página, también se definieron una serie de requisitos iniciales, describiendo las principales páginas de la web:

2.2.1. Página de inicio

Deberá contener:

- Menú: al pulsar en cualquier opción, se irá a la lista de recetas de esa categoría.
- Botón para iniciar sesión/registrar, ir a mi perfil.
- Barra de búsqueda: deberá aparecer en todas las páginas.

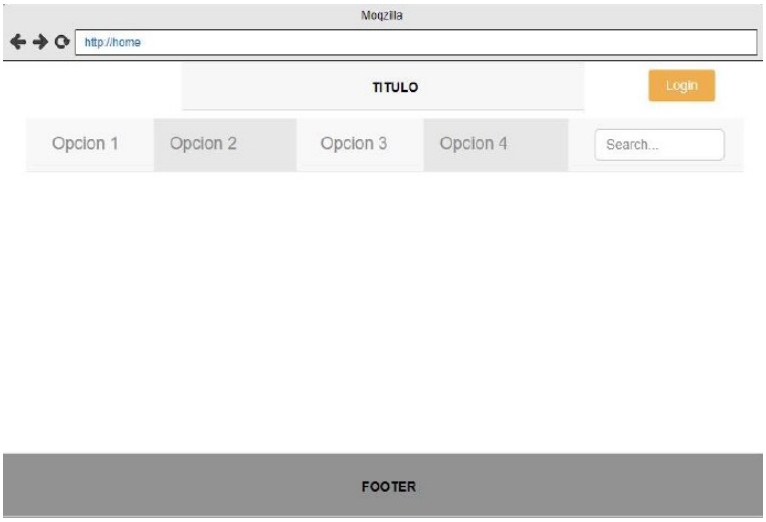


Figura X. Wireframe de la página de inicio

2.2.2. Página de una categoría

Deberá contener una lista de recetas.

Figura X. Wireframe de la página de una categoría

2.2.3. Formulario de registro/inicio sesión

Se encontrarán en páginas separadas de la principal.

2.2.4. Página de perfil

Contendrá:

- Botón de añadir receta
- Información sobre el usuario
- Lista de las recetas del usuario

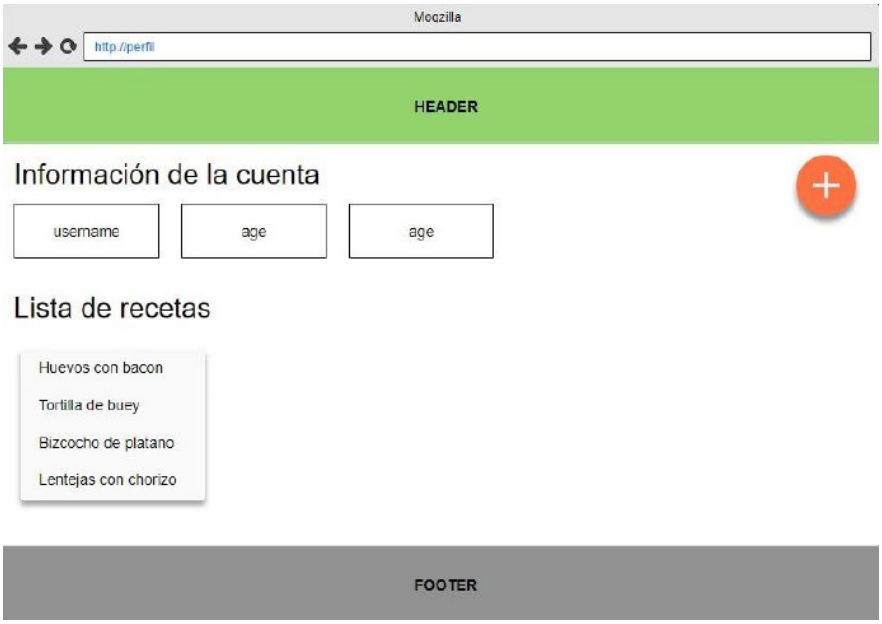


Figura X. Wireframe de la página de perfil de usuario

2.3. Cambios efectuados sobre los requisitos originales

Pese a mantenernos bastante fieles al documento de requisitos original, en aquel momento no teníamos del todo claro cómo queríamos organizar toda la información, lo que nos ha permitido partir de los *wireframes* y requisitos base, al tiempo que los íbamos adaptando para exprimir todo su potencial.

Un claro ejemplo de ello es la página principal: teníamos claro que tendría un menú con las categorías de las recetas y permitiría la navegación; sin embargo, aparte de la cabecera y el pie de página, el resto se quedó en blanco. Eso nos ha permitido dar rienda suelta a las posibilidades que ofrece el mundo del desarrollo web y crear una buena página de inicio, de aspecto llamativo y minimalista, que resulte atractiva a los usuarios/visitantes.

Sin embargo, podemos afirmar que hemos sido bastante fieles al documento de requisitos y *wireframes* originales.

2.3.2. Modificaciones implementación B

Jorge para la implementación B ha cambiado el nombre y logotipo para adaptarme al tipo de público que se describe en el documento de requisitos (público joven, según estudios de mercado muy interesados en recetas saludables).

Además, ha introducido funciones que no estaban especificadas inicialmente como las que siguen (esto se debe a que no estaba presente cuando se definieron los requisitos de forma que, al tener que hacer una implementación de forma individual, los ha adaptado un poco):

- *Reviews/ratings* a las recetas por parte de los usuarios registrados
- Un *API REST* sencillo
- Recetas relacionadas
- Carga de las recetas conforme hacemos *scroll* utilizando *JavaScript, JQuery, AJAX* y mi propio *API REST*
- Botón para ver las recetas paso a paso
- Un rol de *administrador* que puede gestionar todas las recetas, los usuarios y las recetas. En el caso de la implementación A, también han introducido un rol *admin*

3. Detalles de análisis y diseño

3.1. Modelado de la Base de Datos

Una de las primeras tareas para la implementación fue realizar un diagrama de clases y un diagrama entidad-relación que posteriormente plasmamos en tablas. Sin embargo, ese no fue el producto final ya que, conforme se fue desarrollando la web, para aumentar el abanico de posibilidades, decidimos modificar ligeramente lo que ya teníamos.

Inicialmente, partimos de las tablas usuario, categoría, receta y foto. Sin embargo, tal y como se ha comentado, nos percatamos de que el almacenar los pasos y los ingredientes en sus propias tablas nos resultaba bastante útil para darle formato a la visualización de cada receta (la alternativa era guardar todo el texto en un solo campo tipo *TEXT*: tanto la introducción como todos los ingredientes y los pasos). De esta forma, además, conseguimos homogeneizar la visualización de todas las recetas, aportando un *look&feel* similar a toda la página, ya no solo utilizando la misma Interfaz de Usuario, sino mostrando todas las recetas de la misma forma.

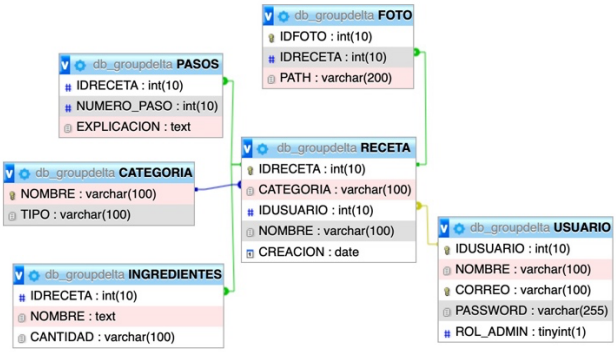


Figura X. Diagrama entidad-relación (draw.io)

3.2. Patrones de diseño

En este apartado, ya podemos observar claras diferencias entre ambas implementaciones; de hecho, este fue uno de los motivos principales por los que una parte del grupo decidió llevar a cabo su propia implementación de forma algo más sencilla. Es decir, cuando Jorge comenzó a aplicar mediante clases *PHP* los patrones de diseño vistos en la parte teórica de la asignatura, aunque trató de que el resto de miembros comprendieran el funcionamiento, decidieron realizar su propia implementación utilizando técnicas que les fueran más familiares y sencillas de implementar.

3.2.1. Implementación A (Carla, Vlad, Ainara)

Los patrones que hemos considerado más adecuados son:

- Para el controlador: el front controller delegar las tareas a un controlador que irá llamando a lo que sea necesario según la información que venga.
- Para las vistas: template view usar plantillas permite ahorrar código y que éste sea más limpio, delegando lo que es la información importante al servidor.
- Para los datos: table gateway tras realizar una clase php para la conexión con la base de datos, ésta es usada por un gateway que gestiona las operaciones sobre la base de datos evitando así que haya la gestión de sentencias SQL en múltiples ficheros. Siendo la clase DBGateway un Singleton pues, como hemos visto en clase, beneficia para accesos a bases de datos.

3.2.2. Implementación B (Jorge)

El objetivo ha sido desde el principio obtener una estructura de clases sencilla, clara y que permita la reutilización de código. Un claro ejemplo es unificar todos los métodos de persistencia en una clase abstracta (*ORMPersistent*), de forma que en las clases que extiendan de ella (*Receta*, *Usuario*...) no sea necesario redefinir todos los métodos (*insert(), update(), delete(), etc*). En todo caso, en algunas ocasiones se puede utilizar polimorfismo (equivalente al *@Override* de *Java*) en caso de querer modificar el comportamiento por defecto en alguna clase, llamando al método de la superclase siempre que sea posible para no duplicar código.

Aunque parece que los patrones utilizados son los mismos que en la otra implementación, en este caso se diferencia claramente la separación lógica entre elementos (lo cual facilita enormemente aplicar el patrón *MVC*).

Algunos de los patrones que ha utilizado son *Front Controller, Page Controller* (para los controladores), *Two-step View, Template View* (para las vistas) y *Database Gatway, Table Module*, *Singleton* (modelos y persistencia), dando lugar a una estructura clara siguiendo el patrón *MVC*. Da más detalles, especialmente de la implementación en la siguiente sección (detalles de implementación → uso de patrones).

4. Detalles de implementación

4.1. Uso de patrones

4.1.1. Implementación A

Controlador: Como hemos mencionado anteriormente hemos seleccionado el patrón *front controller* para la implementación del controlador. De esta manera hemos creado un fichero php con nombre *index*, este fichero recibirá todas las peticiones realizadas por el cliente. Vamos a definir las acciones que debe realizar en base a dos parámetros: acción e id. En un swicth el controlador ira comprobando la acción y dentro de esta el id y realizara acciones necesarias con ayuda del modelo y la vista. Además, al principio se cargas las categorías de cada comida y la cabecera para pasársela a todas las vistas que la requieran.

Modelo: es un fichero php con nombre modelo que se encarga de realizar todas las consultas a la base de datos mediante la clase *DBConexion*. De esta manera implementa el patrón *table Gateway*. El modelo únicamente contiene todas las funciones necesarias para responder a la solicitud del controlador, proporcionándole datos de la base de datos.

Vistas: es un fichero php con nombre vista que se encarga de servir al cliente la página web. Contiene todas las funciones necesarias para devolver una vista al cliente. Estas funciones toman como base las plantillas HTML que hemos definido, a partir de las plantillas y con los datos que el modelo le ha proporcionado al controlador modifica la plantilla creando la página final que obtendrá el cliente como resultado. De esta manera la vista implementa el patrón *template view*.

4.1.2. Implementación B (Jorge)

Ahora sí, voy a dar más detalles sobre los patrones utilizados y su implementación:

A. Controladores:
   - *Front Controller*: un único controlador que procesa todas las peticiones que recibe el servidor y delega la ejecución de la acción pertinente (y/o el renderizado de la página adecuada) en un *Page Controller*. Concretamente, en mi clase el *front controller* está implementado en la clase *WebRecetas*, que además contiene una serie de variables globales para acceder fácilmente a ellos.

- *Page Controller*: en este apartado, tendríamos controladores individuales para cada página, mediante clases como *LandingPage*, *RecipePage*, *UserManagerPage*, etc.
  - Todos ellos extienden de *PaginaPublica*, *PaginaRegistrados* o *PaginaAdmin*, que gestionan los permisos de acceso.

B. Vistas
   - *Two Steps View*: las vistas se basan en que, primero, se transforman los datos/modelos necesarios en una presentación lógica, a la que luego se le aplica un formato global (encabezado, estilado *CSS*...), facilitando que todas las páginas tengan un *look&feel* similar. Para ello, se utilizan tanto *Tranform View* (modelos → representación *HTML*) como *Template View* (rellenado de huecos o *placeholders*) en los documentos *HTML*.
     Cada controlador a un archivo *HTML* (aunque hay alguno como el *Sitemap* que está asociado a un *XML*), y mediante la clase *Template*, se rellenan los *placeholders* o huecos que dejamos en la plantilla. Esto nos facilita separar las vistas (documentos HTML) de los controladores, manteniendo el código limpio y organizado.

C. Para los modelos y la persistencia de los mismos:
   - *Table Data Gateway*: encapsula el código *SQL* necesario para realizar consultas y operación contra la base de datos, permitiéndonos abstraernos, de forma que no tengamos que "esparcir" el *SQL* a lo largo de todos los modelos. Contiene métodos como *insert(), update(), delete(), get()* genéricos. En concreto a mí me ha sido muy útil realizar esta abstracción porque he realizado el apartado extra consistente en utilizar un *SGBD* diferente, y me ha bastado con crear una clase para cada motor de BD. Es decir, tengo una clase abstracta (*DBGateway*), de la que extienden otras como *MySQLGateway* y *SQLServerGateway*.
   - *Table Module*: serían las clases correspondientes a cada clase. Como he comentado anteriormente, he unificado todos los métodos relacionados con mapeo relacional en una clase abstracta (*ORMPersistent*), de forma que en las clases que extiendan de ella (*Receta*, *Usuario*...) no sea necesario redefinir todos los métodos.

Esto implica que, obviamente, ha utilizado el patrón *MVC* (*Model View Controller*). Por último, cabe destacar el uso, en ambos grupos, de:

D. *Singleton*: para tener en todo momento una instancia única del *Front Controller* y de las *Database Gateway*.

A continuación, adjunta un diagrama de clases donde se ve un poco por encima cómo se relacionan las distintas clases que he implementado:

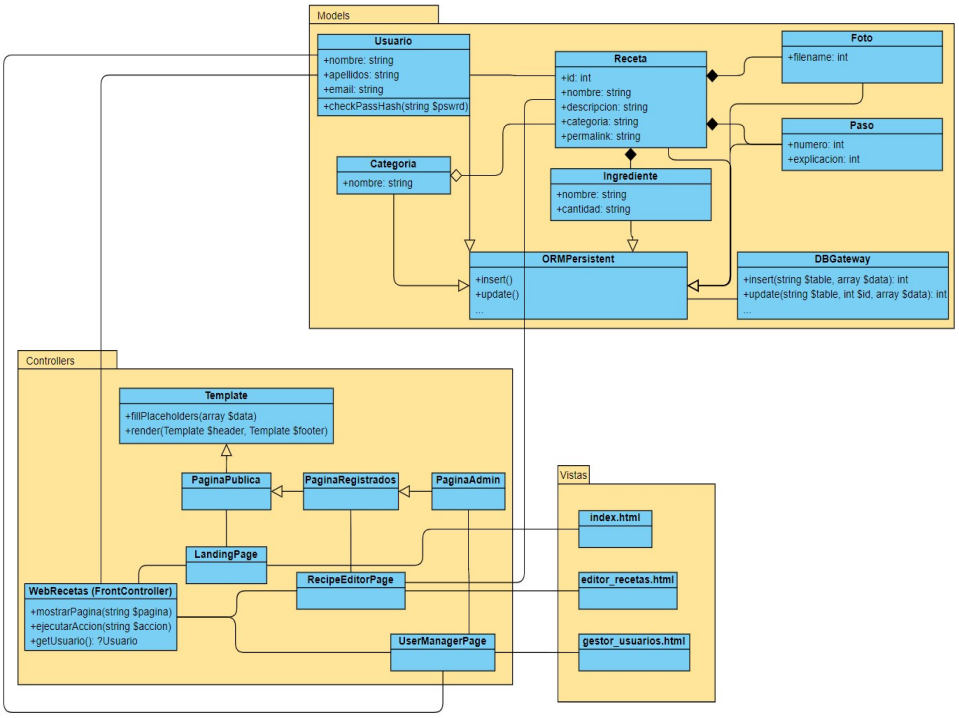


Figura X. Diagrama de clases de la implementación B

4.2. APIs

4.2.1. Implementación A

No se han utilizado *APIs*.

4.2.2. Implementación B (Jorge)

En mi caso, me habría gustado utilizar *APIs* externas como ya he hecho en otros proyectos (especialmente, algún servicio *REST* como *Firebase*, para poder implementar notificaciones *push*), pero esto ha quedado en el apartado de líneas futuras de desarrollo dado que han sido muchas las cosas que he tenido que implementar de forma individual, al realizar la implementación yo solo.

Sin embargo, sí que he implementado un *API* para utilizarlo en mi propio *fronend* y, ante la posibilidad de, en un futuro, documentarlo debidamente, y que otras aplicaciones puedan interactuar con la mía a través de dicho *API*.

Se trata de un *API* tipo *REST* que permite realizar operaciones *CRUD*, principalmente. Ejemplo:

Endpoint	Método	Campos de la petición	Contenido de la respuesta
`/API/receta/<id>`	`GET`	-	Información de la receta en formato *JSON*
`/API/receta`	`POST`	nombre descripcion categoría paso1 … pasoN ingr1 … ingrM cant1 … cantM	Id de la receta insertada
`/API/receta/<id>`	`POST`	nombre descripcion categoría paso1 … pasoN ingr1 … ingrM cant1 … cantM	0 (error) / 1 (receta actualizada correctamente)
`/API/receta/<id>`	`DELETE`	-	0 (error) / 1 (receta borrada correctamente)
`/API/receta/<id>`	`HEAD`	-	No implementado
`/API/receta`	`OPTIONS`	-	No implementado

Figura X. Documentación del API correspondiente al modelo receta

Este mismo modelo está implementado para *reviews*, fotos de recetas… y me ha resultado útil para poder hacer peticiones desde *JavaScript* utilizando *JQuery* y más específicamente *AJAX* (por ej: añadir un comentario sin recargar la página: envío una petición tipo `POST` a `/API/review` con el contenido deseado.

4.3. Servidores de Bases de Datos

4.3.1. Implementación A

Para pruebas, se ha utilizado una base de datos local y así, tener la definitiva en el servidor proporcionado para que se mantenga intacta. Amabas con la misma estructura y datos. Se consideró adecuado trabajar así pues si se trabajara sobre una buena desde el principio, podría llevar a conflictos y pérdidas de datos relevantes.

Además, se instaló en las máquinas virtuales phpmyAdmin y SQL Server tal como se indica en el fichero colgado en recursos. Todo desde la máquina virtual de Carla, lugar donde se encuentra también la web.

4.3.2. Implementación B (Jorge)

Además de las dos versiones de *MySQL* (local y en un servidor centralizado), ha realizado el apartado extra de la práctica de Bases de Datos, añadiendo un motor de bases de datos (*SQL Server*). Como se indica en la práctica, para obtener puntuación extra, documento a continuación el proceso de instalación:

1. Instalación de *SQL Server* en *Ubuntu*: para ello, he descargado e instalado el paquete correspondiente a *Ubuntu 18.04 Bionic*, que es la versión del *SO* que tenemos instalada en nuestras máquinas.

```
wget -qO- https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add – sudo add-apt-repository "$(wget
-qO- https://packages.microsoft.com/config/ubuntu/18.04/mssql-server-2019.list)" sudo apt-get update sudo apt-
get install -y mssql-server sudo /opt/mssql/bin/mssql-conf setup systemctl status mssql-server --no-pager
```

2. Migración de la *BD* de *MySQL* a *SQL Server*: este proceso no ha sido sencillo del todo, puesto que, incluso seleccionando "MSSQL" como variante de *SQL* al exportar desde *PHPMyAdmin*, he tenido problemas con la sintaxis y he tenido que reescribir gran parte del código.

3. Instalación de los *drivers* para conectarme a la nueva *BD* y ejecutar consultas contra ella desde los *scripts* de *PHP*: ha sido un proceso algo más complejo de lo normal dado que he tenido que buscar una versión antigua para que fuera compatible con nuestro servidor *Apache* y nuestra versión de *PHP*. La he obtenido a través de *pecl* (repositorio de extensiones de *PHP*):

```
curl https://packages.microsoft.com/keys/microsoft.asc | apt-key add - curl
https://packages.microsoft.com/config/ubuntu/16.04/prod.list > /etc/apt/sources.list.d/mssql-release.list apt-
get update
ACCEPT_EULA=Y apt-get install msodbcsql17
sudo pecl install sqlsrv-5.9.0preview1 pecl
install pdo_sqlsrv
```

Una vez instaladas, he tenido que incluir estas extensiones en el archivo *php.ini*, añadiendo las líneas:

```
extension=sqlsrv.so extension=pdo_sqlsrv.so
```

4. Por último, he implementado una nueva clase *SQLServerGateway*, que, junto con la ya existente *MySQLGateway*, extienden de la clase abstracta *DatabaseGateway*. En estas clases se abstrae la implementación específica a cada plataforma, pudiendo utilizarse la una o la otra indistintamente.
   Cabe destacar que las funciones para el *SQL Server* de *Microsoft*, como es natural, son diferentes que las de *MySQL*, he aquí algunos ejemplos:

```
$this->conexion = sqlsrv_connect($host, [

"Database"=>$db,
```

```
"UID"=>$user,
"PWD"=>$pass,
    "CharacterSet"=>"UTF-
8"
]);
```

```
$statement = sqlsrv_query($this->conexion, $sql, $params);
```

```
$result = sqlsrv_fetch_array($stmt)
```

```
sqlsrv_rows_affected($stmt);
```

```
print_r(sqlsrv_errors())
```

5. Detalles de despliegue

5.1. *CI/CD*

En este caso, ninguno de los dos grupos de trabajo hemos realizado una *Pipeline* compleja ni nada por el estilo, porque excede los objetivos de este proyecto, al menos con la cantidad de tiempo que tenemos disponible. En el caso de Jorge, sí que he implementado un *Pipeline* en *Gitlab* para llevar a cabo un proceso de *CI/CD* (*Continuous Integration, Continuous Delivery & Continuous Deployment*).

5.1.2. Implementación B

Mediante el archivo *.gitlab-ci.yml*, se ha configurado una *Pipeline* que solo tiene una fase (*Stage*), correspondiente al despliegue (*deployment*). Como el único *Runner* que he asignado para ejecutar los *Jobs* es la propia máquina servidor, se ha configurado este *Pipeline* para que, cada vez que haga un *commit* al repositorio, se ejecute *git pull* en la máquina servidor, quedando continuamente desplegados los cambios. Para ello:

1. Instalar el *runner*

```
curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh | sudo bash
sudo apt-get install gitlab-runner sudo service gitlab-runner status
```

2. Registrar el *runner*

```
sudo gitlab-runner register
```

Ahora introduciríamos la *URL* y el *token* extraídos de la configuración de *Gitlab* (repositorio → configuración → CI/CD → runners). También hay que activar la

casilla para que se le asignen trabajos sin etiquetas a este *runner* (aunque podríamos utilizar etiquetas para asegurarnos que la web se despliega en el *runner* correcto, si añadiésemos más *runners*).

3. Probar el funcionamiento haciendo un *commit* y un *push* desde otra máquina.

Este proceso se hea montado de esta forma porque realmente no se trabaja en una máquina en producción y, si hay algún fallo al ejecutar el nuevo código, no pasa nada. Sin embargo:

a) Debería hacer que el *git pull* se ejecutara en un entorno de preproducción (carpeta *pre* en lugar de *html*)
b) Debería incluir muchas fases anteriormente, para verificar el código y realizar tests tanto unitarios como de integración, auditorías automáticas... como los que figuran a continuación (basados en los apuntes de la asignatura):

    - Linting HTML/CSS/JS: analizar código fuente → errores, warnings, malas prácticas
    - Transformar imágenes (comprimir...)
    - Beautify código
    - Minificar HTML/CSS/JS ([fichero].min.*)
    - "Mergear" CSS/JS: reducir número de peticiones al servidor
    - Generar Cachés estáticos de servidor (reducir *TTFB*)
    - Tests unitarios: verificación y validación → code coverage
    - Tests de integración
    - Auditar y generar informes ○ Usabilidad/Accesibilidad/SEO/Rendimiento: Chrome LightHouse CI ○ Seguridad: "DevSecOps", análisis en busca de vulnerabilidades
    - Automatizar despliegues: local, remoto (FTP, AWS...)
    - Actualizar documentación y readme.md
    - Actualizar badges/shields al repositorio

6. Auditorías

6.1. Implementación A

6.1.1. Auditoría de rendimiento

Las pruebas realizadas han sido:

Prueba 1: página principal.

Prueba 2: listado de recetas.

Prueba 3: vista de una receta.

Prueba 4: galería de imágenes de una receta.

Prueba 5: perfil de usuario.

Prueba 6: formulario de creación de una receta.

Prueba 7: formulario de modificación de una receta.

Prueba 8: formulario de registro / log in.

Resultados obtenidos:

Prueba 1: la auditoria de rendimiento de Chrome otorga a la página principal un 88%. Los resultados obtenidos son los siguientes: first content Paint 0.9 s, speed index 1.1 s, largest contentful paint 2.2 s, time to interactive 1.3 s, total blocking time 0 y cumulative layout shift 0.006 s. Además ofrece una serie de indicaciones, eliminar los recursos que bloquean el renderizado, eliminar los ficheros CSS no utilizados y precargar solicitudes de claves.

Prueba 2: la auditoria de rendimiento de Chrome otorga al listado de recetas un 89%. Los resultados obtenidos son los siguientes: first content Paint 1 s, speed index 1 s, largest contentful paint 2 s, time to interactive 1.1 s, total blocking time 0 y cumulative layout shift 0 s. Además ofrece una serie de indicaciones, utilizar imágenes de tamaño adecuado, eliminar los recursos que bloquean el renderizado, eliminar ficheros CSS y JS no utilizados.

Prueba 3: la auditoria de rendimiento de Chrome otorga a la vista de una receta un 89%. Los resultados obtenidos son los siguientes: first content Paint 0.9 s, speed index 0.9 s, largest contentful paint 2 s, time to interactive 1.5 s, total blocking time 0 y cumulative layout shift 0.011 s. Además ofrece una serie de indicaciones que son las mismas que ha ofrecido en la prueba anterior.

Prueba 4: la auditoria de rendimiento de Chrome otorga a la galería de imágenes de una receta un 89%. Los resultados obtenidos son los siguientes: first content Paint 0.8 s, speed index 0.8 s, largest contentful paint 2 s, time to interactive 1.1 s, total blocking time 0 y cumulative layout shift 0 s. Además ofrece una serie de indicaciones, utilizar imágenes de tamaño adecuado, eliminar los recursos que bloquean el renderizdo y eliminar los ficheros CSS que no se utilizan.

Prueba 5: la auditoria de rendimiento de Chrome otorga al perfil de usuario un 95%. Los resultados obtenidos son los siguientes: first content Paint 0.9 s, speed index 1 s, largest contentful paint 1.3 s, time to interactive 1.5 s, total blocking time 30 ms y cumulative layout shift 0 s. Además ofrece una serie de indicaciones, eliminar los recursos que bloquean el renderizado, utilizar imágenes con tamaño adecuado, eliminar los ficheros CSS no utilizados y precargar solicitudes de claves.

Prueba 6: la auditoria de rendimiento de Chrome otorga al formulario de creación de receta un 96%. Los resultados obtenidos son los siguientes: first content Paint 0.9 s, speed index 0.9 s, largest contentful paint 1.3 s, time to interactive 1 s, total blocking time 0 s y cumulative layout shift 0 s. Además ofrece una serie de indicaciones, eliminar los recursos que bloquean el renderizado y eliminar los ficheros CSS que no se utilizan.

Prueba 7: la auditoria de rendimiento de Chrome otorga al formulario de modificación un 91%. Los resultados obtenidos son los siguientes: first content Paint 1.2 s, speed index 1.2 s, largest contentful paint 1.5 s, time to interactive 1.2 s, total blocking time 0 s y cumulative layout shift 0 s. Además ofrece una serie de indicaciones, eliminar los recursos que bloquean el renderizado, eliminar los ficheros CSS no utilizados y utilizan imágenes de tamaño adecuado.

Prueba 8: la auditoria de rendimiento de Chrome otorga al formulario de login/registro un 94%. Los resultados obtenidos son: first content Paint 0.9 s, speed index 1 s, largest contentful paint 1.4 s, time to interactive 1.3 s, total blocking time 0 s y cumulative layout shift 0 s. Además ofrece una serie de recomendaciones, eliminar los recursos que afectan al renderizado y eliminar los ficheros CSS no utilizados.

Modificaciones propuestas:

Prueba 1: en base a los resultados obtenidos proponemos utilizar imágenes más pequeñas o que no afecten tanto al renderizado y eliminar todos los ficheros CSS y JS que no se utilizan.

Prueba 2: en base a los resultados obtenidos proponemos redimensionar las imágenes al tamaño en el que van a ser mostradas en la web y eliminar todos los ficheros CSS y JS que no se utilizan.

Prueba 3: en base a los resultados obtenidos las modificaciones propuestas son las mismas que han sido propuestas en la prueba anterior.

Prueba 4: en base a los resultados obtenidos proponemos redimensionar las imágenes al tamaño en el que van a ser mostradas en la web y eliminar los ficheros CSS que no se utilizan.

Prueba 5: en base a los resultados obtenidos las modificaciones propuestas son redimensionar las imágenes al tamaño en el que van a ser mostradas y eliminar los ficheros CSS que no se utilizan.

Prueba 6: en base a los resultados obtenidos se propone eliminar todos los ficheros CSS que no se utilizan.

Prueba 7: en base a los resultados obtenidos se propone redimensionar las imágenes al tamaño en el que van a ser mostradas y eliminar los ficheros CSS no utilizados.

Prueba 8: en base a los resultados obtenidos se propone eliminar todos los ficheros CSS que no se utilizan.

6.1.2. Auditoría SEO

Para las pruebas de SEO hemos realizado una prueba por página y para ello hemos utilizado las herramientas de desarrolladores de Chrome.

Prueba 1: página principal.

Prueba 2: listado de recetas.

Prueba 3: vista de una receta.

Prueba 4: galería de imágenes de una receta.

Prueba 5: perfil de usuario.

Prueba 6: formulario de creación de una receta.

Prueba 7: formulario de modificación de una receta.

Prueba 8: formulario de registro / log in.

Resultados obtenidos:

Prueba 1: la auditoria SEO de Chrome a otorgado a la página principal un 90%. Indica que le falta un atributo *meta description*. Además, hay que añadir que 4 auditorias no han podido ser evaluadas.

Prueba 2: la auditoria SEO de Chrome a otorgado a la página que contiene un listado de recetas un 80%. Las indicaciones recibidas son las siguientes: falta un atributo *meta description* y las imágenes de la página no tienen atributo *alt*.

Prueba 3: la auditoria SEO de Chrome a otorgado a la página de la vista de una receta un 80%. Los motivos son los mismos que en la prueba anterior, falta el atributo *meta description* y las imágenes no tienen el atributo *alt.*

Prueba 4: la auditoria SEO de Chrome a otorgado a la galería de imágenes un 80%. Los motivos vuelven a ser los mismos, falta el atributo *meta description* y las imágenes no tienen el atributo *alt.*

Prueba 5: la auditoria SEO de Chrome a otorgado al perfil de usuario un 80%. Los motivos vuelven a ser los mismos, falta el atributo *meta description* y las imágenes no tienen el atributo *alt.*

Prueba 6: la uditoria SEO de Chrome a otorgado al formulario de creación de una receta un 90%. La indicación que ofrece es que falta el atributo *meta descripction.*

Prueba 7: la auditoria SEO de Chrome a otorgado al formulario de modificación de una receta un 80%. Los motivos vuelven a ser los mismos, falta el atributo *meta description* y las imágenes no tienen el atributo *alt.*

Prueba 8: la auditoria SEO de Chrome a otorgado al formulario de registro/login un 90%. La indicación que nos ofrece es que falta el atributo *meta description.*

Modificaciones propuestas:

Prueba 1: la modificación que se propone es añadir el atributo indicado a la plantilla HTML del header.

Prueba 2: debemos introducir el atributo *alt* en todas las imágenes, ya que también es importante para valorar la accesibilidad, además de añadir el atributo *meta description* a la plantilla HTML del header.

Prueba 3: las modificaciones adecuadas con las mismas que las propuestas para la prueba anterior.

Prueba 4: las modificaciones adecuadas vuelven a ser las mismas que han sido propuestas para la prueba 2.

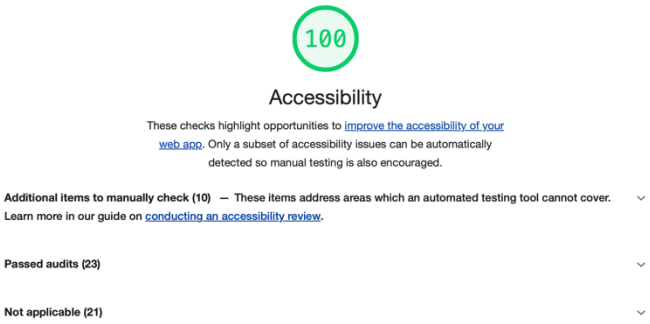
Prueba 5: las modificaciones adecuadas vuelven a ser las mismas que han sido propuestas para la prueba 2.

Prueba 6: la modificación adecuada en este caso es la misma que ha sido propuesta para la prueba 1.

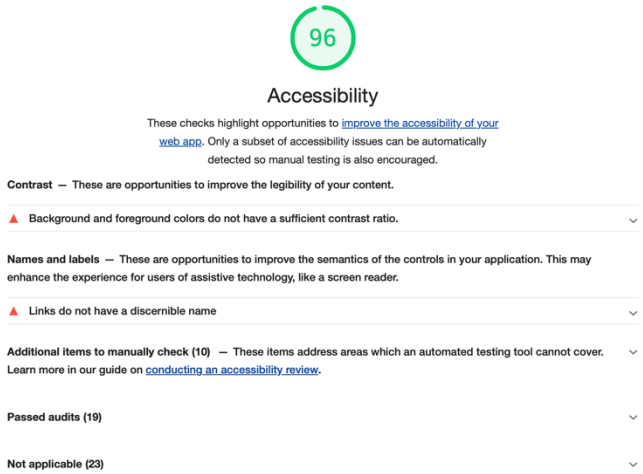
Prueba 7: las modificaciones adecuadas son las mismas que han sido propuestas para la prueba 2.

Prueba 8: la modificación adecuada en este caso en la misma que ha sido propuesta en la prueba 1.

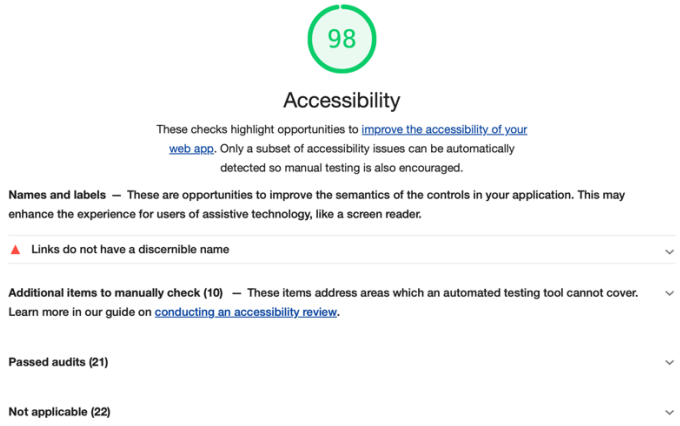
6.1.3. Auditoría de accesibilidad



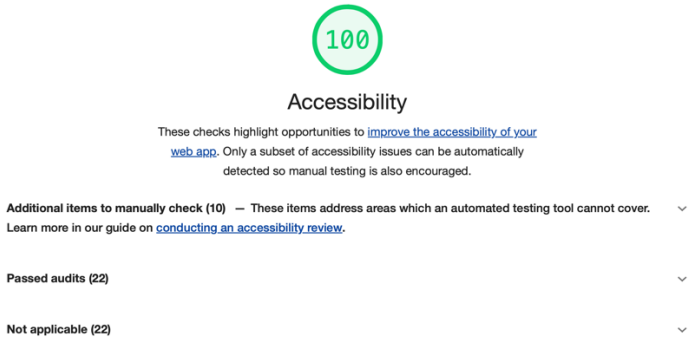
Auditoría realizada en la página de login.



Auditoría realizada en la página de una receta. Los cambios aplicados son los indicados a corregir en la imagen.



Auditoría realizada en la página del listado de recetas. Los cambios aplicados son los indicados a corregir en la imagen.



Auditoría realizada en la página principal.

6.2.8. Conclusiones accesibilidad

Como el ordenador de Carla es MAC, está dotado de la herramienta VoiceIver la cual es una ayuda de voz tal como hemos visto. Ha sido utilizada para la web y hemos podido observar que da muy buenos resultados pues tiene atributos alt, title, name, *aria-describedby…*, además de un buen diseño ya que es sencillo, alto contraste…) y el uso de librerías *JS, como SweetAlert,* también *accesibility-friendly*.

6.2.9. Conclusiones SEO

Al igual que en el caso de la accesibilidad, se puede ver el gran resultado que conseguimos gracias a:

- La etiqueta *title* y metaetiquetas.
- El correcto uso de encabezados y semántica.

6.2. Implementación B

Al igual que mis compañeros, he utilizado las completísimas herramientas que *Google Chrome* nos facilita (*Lighthouse*) para llevar a cabo la mayoría de auditorías en mi página web.

Voy a hacer todas las auditorías de forma conjunta en cada página, y luego haré las conclusiones de forma individual para cada tipo de prueba. Para ver las auditorías completas, se puede consultar el directorio "auditorias" de mi carpeta en el repositorio grupal.

6.2.1. Auditoría página de inicio

Podemos ver cómo en los resultados, la página de inicio tiene un rendimiento algo deficiente; esto se debe principalmente a la carga de *assets*: concretamente las imágenes del *slider*, que deben tener una resolución alta al ocupar toda la pantalla, y las librerías *Bootstrap* y *JQuery*. En cuanto a accesibilidad, buenas prácticas y SEO, prácticamente no habría nada a mejorar (el 3% restante en accesibilidad se debe a que hay algún elemento con poco contraste como las estrellas grises que indican el *rating* de cada receta).

Figura X. Resultados auditoría página de inicio (implementación B)

6.2.2. Página de una receta

En este caso, nos encontramos con una situación muy parecida, nos quedaríamos algo cortos de rendimiento por culpa de la carga de varias imágenes, estando el resto de los apartados bien optimizados.

Figura X. Resultados auditoría página de una receta

6.2.3. Página con una lista recetas

Misma situación que en las pruebas anteriores, al tener tantas imágenes (una por cada receta). En la sección de "modificaciones propuestas" sugeriré alguna forma de solucionar esta merma en el rendimiento por culpa de las imágenes.

Figura X. Resultados auditoría página lista de recetas

6.2.4. Página de una categoría

En la página de la categoría obtenemos un resultado un poco mejor:

Figura X. Resultados auditoría página de una categoría

6.2.5. Página editor de recetas

En este caso, cabe destacar que la cantidad de elementos de interacción (*inputs*, botones, diálogos, interacciones, subida de archivos, etc). Por lo tanto, me parece un resultado realmente satisfactorio un 97% en el apartado de accesibilidad; esto significa que he hecho un buen uso de las etiquetas (*label*) y de los atributos *aria-*.

Figura X. Resultados auditoría editor de recetas

6.2.6. Página gestionar recetas

En este caso, al igual que antes, tenemos muchos elementos: en concreto, una tabla con varias recetas que hayamos creado, cada una de ellas con 3 botones, correctamente descritos con *aria-label*.

Figura X. Resultados auditoría página gestionar recetas

6.2.7. Conclusiones rendimiento

El rendimiento es el peor de los cuatro aspectos, y es que las librerías *Bootstrap* y *JQuery*, junto con las imágenes en alta resolución, retrasan el renderizado de la página notablemente (aunque no es algo grave, como se puede observar). A reducir este tiempo contribuye el hecho de que uso una *Content Delivery Network*, aunque también tenemos que tener en cuenta que todas las imágenes están guardadas en local; es decir, probablemente al viajar por internet provocarían mayores retrasos.

Las sugerencias de mejora, que dejo como líneas futuras de desarrollo, son:

- Utilizar una *CDN* para las imágenes
- He asignado políticas de *caché* tras realizar las pruebas para evitar peticiones innecesarias de recursos o *assets* estáticos (imágenes, css, js, etc).
- Redimensionar las imágenes: o bien al subirlas, escalarlas a 3 tamaños (thumbnail, HD y FHD o resolución original); o bien redimensionarlas en tiempo real con *PHP* utilizando parámetros *GET* (*/getimagen.php?width=X&height=Y*). Tendría que probar el impacto de cada una de las dos formas, tanto en la calidad de la imagen como en el rendimiento.
- Sustituir las librerías *Bootstrap* y *JQuery* por otras más ligeras o cargar solo la parte que vaya a utilizar.

6.2.8. Conclusiones accesibilidad

Se puede ver reflejado en los buenos resultados el correcto uso de descripciones alternativas, hints y ayudas para los lectores de pantalla (alt, title, name, *aria-label, aria-describedby…*), además de un buen diseño minimalista (sencillo, alto contraste, librería *Bootstrap accesibility-friendly*) y el uso de librerías *JS, como SweetAlert,* también *accesibility-friendly*.

6.2.9. Conclusiones SEO

Al igual que en el caso de la accesibilidad, se puede ver el gran resultado que conseguimos gracias a:

- La etiqueta *title* y metaetiquetas como *keywords* y *description*, generados dinámicamente en cada página.
- El correcto uso de encabezados y semática.
- La generación de un *sitemap* dinámicamente con todas las etiquetas y categorías.

7. Pruebas de usabilidad

Dentro del marco de desarrollo del proyecto final de la asignatura, un concepto clave a tener en cuenta durante todo el proceso, es la usabilidad. Por ello, es necesario especificar y documentar las pruebas de usabilidad que se pretenden realizar una vez el proyecto final de la asignatura esté implementado.

En nuestro caso, buscamos desarrollar una página intuitiva, fácil de usar (*userfriendly*) y estéticamente agradable, para que el usuario tenga una percepción positiva, lo cual pretendemos corroborar mediante las pruebas de usabilidad que siguen.

Por supuesto, y también dentro de la usabilidad, vamos a aplicar una serie de buenas prácticas de accesibilidad, para evitar cualquier barrera que provoque que nuestro servicio sea inaccesible, y así permitir que sea usable por el mayor número posible de usuarios, independientemente de sus conocimientos o sus capacidades personales.

Es por ello que se llevará a cabo un análisis del perfil de usuario, se describirán las tareas que deben realizar los usuarios, sus flujos de trabajo, etc. De esta forma, el diseño y la implementación cumplirán desde un principio con las bases de usabilidad y accesibilidad, y podremos realizar *tests* de accesibilidad con resultados potencialmente positivos.

En conclusión, vamos a aplicar el denominado “diseño centrado en el usuario”, consistente en la implicación activa por parte del usuario en el desarrollo. Para ello, se realizan varias iteraciones contando con el *feedback*, que permiten refinar los requisitos.

7.1. Pruebas realizadas

7.1.1. Entrevistas con el usuario

Se tendrá en cuenta la opinión de usuarios representativos, mediante una serie de preguntas guionizadas, para detectar cuáles son las necesidades, expectativas y comportamiento de los usuarios finales.

7.1.2. Cuestionarios

Recoger información sobre la experiencia y las preferencias de usuarios mediante la circulación de un cuestionario breve. De esta forma, como con el método anterior, trataremos de detectar cuáles son las necesidades, expectativas y comportamiento de los usuarios finales.

7.1.3. Observar al usuario

Para entender su flujo de trabajo habitual, podemos observar al usuario final tratar de realizar la misma tarea que pretendemos que realicen en nuestro sitio web (buscar o publicar recetas), ver cómo le resulta más cómodo hacerlo, etc.

7.1.4. Estado del arte

Además de informarnos del sector (por ejemplo, publicaciones relevantes), en nuestro caso la competencia es muy grande (hay muchas páginas de recetas), por lo que es clave ver qué es lo que hace bien y qué es lo que hace mal cada una de ellas.

En este sentido, y relacionado con el apartado anterior (observar al usuario), es una buena idea observar cómo el usuario utiliza sitios similares, qué le resulta cómodo, qué no, etc.

7.1.5. Pruebas de usabilidad en un entorno real

Conforme se vayan desarrollando las diferentes funcionalidades, se puede hacer uso de usuarios reales que sean representativos, y pedirles que hagan uso de estas funciones (lista de tareas). Es muy importante tener en cuenta los comentarios y observaciones del usuario en vivo, así como el lenguaje no verbal.

7.1.6. Co-descubrimiento

En este caso, dos participantes realizarían una tarea juntos de forma colaborativa.

7.1.7. Medidas de rendimiento

En ambos casos anteriores, y en los que siguen, haremos uso de medidas de rendimiento como:

- Tiempo en alcanzar el objetivo
- Número de clicks
- Número de errores
- Número de pantallas visitadas
- Porcentaje de tiempo con el ratón activo

Estableceremos objetivos como los siguientes:

- El usuario debe ser capaz de encontrar la receta deseada en menos de 1 minuto.
- El usuario debe ser capaz de eliminar una receta que previamente haya creado en un máximo de 10 clics.

- El usuario no debe cometer ningún error a la hora de publicar una receta; y si lo hace, debemos asegurarnos de que el sistema sea capaz de predecirlo y/o avisarle para que pueda deshacer/cancelar la acción, en caso de que la haya realizado por error (por ejemplo, si intenta guardar la receta sin especificar ningún ingrediente, o si el tiempo de elaboración especificado es desorbitado, podría tratarse de un error).

7.1.8. Pequeña variación de A/B Testing

En el caso de que se nos ocurran dos formas diferentes de realizar la misma tarea, estudiar con cuál de las dos se sienten más cómodos, se consigue mayor porcentaje de acierto o se completa la tarea en menor tiempo. Para ello, le pedimos a dos grupos de usuarios que hagan la tarea con los dos métodos diferentes y comparamos las métricas recogidas entre ambos grupos.

En nuestro caso, esto se podría aplicar perfectamente dado que hemos realizado dos implementaciones distintas, y podríamos tratar de averiguar cuál gusta más a los usuarios, cuál logra mayor porcentaje de retención de audiencia, cuál consigue que los usuarios logren su objetivo con menos clics/menos fallos...

7.1.9. Testeo en diferentes navegadores y Sistemas Operativos

Durante el desarrollo, se irán testeando las diferentes funcionalidades en diferentes navegadores (tanto los más modernos como algunos antiguos, con diferentes motores de renderizado, etc). Así mismo, se probará a acceder con diferentes resoluciones de pantalla y sistemas operativos (móviles, portátiles con Windows, ordenadores de sobremesa con pantalla 4:3 con Linux, etc).

La página debe ser totalmente funcional independientemente del sistema operativo (*OS*), motor de renderizado, resolución de pantalla, navegador, etc.

7.2. Resultados obtenidos

7.2.1. Pruebas con usuarios reales

Aunque el tiempo que tenemos para realizar estas pruebas es muy reducido, hemos realizado pruebas de usabilidad con personas pertenecientes a nuestros respectivos círculos, recogiendo su *feedback* y reflejándolo en modificaciones de la interfaz de usuario, nuevas funcionalidades, etc.

7.2.2. A/B testing

A algunos usuarios se les ha pedido que utilicen la Implementación A, y a otros, la implementación B, de modo que hemos recogido métricas de rendimiento con ambas páginas, así como la opinión de los usuarios, y en general ha resultado ser más intuitiva, sencilla de usar y minimalista la Implementación b, destacando por tener un *look&feel* más uniforme.

7.2.3. Testeo en diferentes navegadores y Sistemas Operativos

Las pruebas realizadas en diferentes dispositivos, resoluciones, navegadores, motores de renderizado y Sistemas Operativos han sido exitosas. En mi caso (Jorge, implementación B), he utilizado páginas como "*Can I Use?*", que muestran la compatibilidad de diferentes funciones *JavaScript*, atributos *CSS* y *HTML* con diferentes navegadores y plataformas, para asegurarme de que la página es accesible desde cualquier tipo de navegador y/o dispositivo.

7.3. Modificaciones propuestas

Entre otras, las modificaciones propuestas son la unificación del *look&feel* de la implementación A y la introducción de un botón para borrar comentarios en la implementación B.

8. Gestión de configuración

En proyectos como este donde se trabaja con muchos archivos que cambian constantemente, y, además, de forma grupal, es crucial una buena gestión de la configuración.

8.1. Uso de repositorios

En nuestro caso, el control de versiones y la gestión de configuración la realizaremos a través de un repositorio grupal de *git*, concretamente utilizando *Gitlab*. Este, además de para llevar un control de versiones y la gestión de configuración, nos permitirá coordinarnos y sincronizar nuestro trabajo, al tratarse de un sistema distribuido.

Queremos destacar que el grupo de implementación A, tiene su propio repositorio en GitHub con toda la implementación, para no entorpecer el trabajo de Jorge. Url del sitio: https://github.com/vlad-b1122/ProyectoIngWeb.git. Se decidió GitHub porque nos vemos muy cómodos trabajando con él.

8.2. Integración continua

En el caso de la implementación B (Jorge), se ha especificado en el apartado "Detalles de despliegue" cómo he configurado una *Pipeline* muy sencilla para realizar una pequeña toma de contacto con el *CI/CD*.

8.3. Trabajo en equipo

Como se comenta en apartados anteriores, el miembro Jorge, no podía asistir a clases. Se decidió como un miembro potencial ya que sabía mucho y podía enseñar bastante. Sin embargo, se comprobó que no era un equipo bien coordinado. Este nuevo miembro, pese a saber mucho, vimos que no éramos compatibles para trabajar y cada vez implementaba más cosas que no se entendían, pese a intentar hablar entre todos, no se llegó a una coordinación adecuada ya que el nuevo miembro, estaba falto de tiempo para explicarnos al detalle su modo de trabajar y así poder hacer un trabajo colaborativo. Se perdió más tiempo en intentar comprender que en aprender y ejecutar.

Tras comentarlo con el profesor, se vio que la mejor solución era una doble implementación pues Vlad, Ainara y Carla solo querían demostrar que habían aprendido en esa asignatura (queremos apuntar que nos ha gustado mucho). Destacar que a partir de entonces, hemos trabajado muy bien entre los tres.

Aunque no sea el mismo nivel, creemos que la implementación A está también muy bien ejecutada y que es correcta. A veces, lo más simple, no es peor. Se trata de aprender, no de lucirse.

Las primeras tareas fueron:

- Diseño del logotipo: Carla
- Primer diseño de la base de datos: Carla y Jorge
- Poblado de la BD: Carla (recetas de ejemplo para poder realizar pruebas durante la implementación de los modelos y la conexión con la base de datos).

Algo que vino de novedad fueron las aportaciones de Jorge; como ya tenía algo de experiencia en realizar webs dio a conocer el mundo de *Bootstrap*, un *framework* de *CSS* muy completo, reemplazando el primer boceto *HTML* y *CSS* de Vlad y Ainara. Para enseñar su funcionamiento y el gran abanico de posibilidades que ofrecía, se encargó de diseñar las plantillas (o vistas, en lo que al patrón *MVC* se refiere). Dicha tarea consistió, en primer lugar, en el diseño de plantillas *HTML*, estilado con *CSS*, definición del comportamiento mediante *JavaScript*.

Posteriormente, Jorge también se encargó de rellenar dichas plantillas en el *backend* para crear contenido dinámico, además de crear un documento "*.htaccess*", que redirigiese todas las URLs a un mismo script, que también implementó ("controller.php"). Dicho script se encarga de crear y mostrar las vistas y, en el futuro, se encargará de realizar los cambios pertinentes en el modelo (haciendo de nuevo referencia al patrón MVC).

La idea era, a partir de aquí (controladores y la base de todas las vistas ya implementados), hacer entre todos los modelos y las clases de conexión a la BD, y utilizar las "piezas" que Jorge había preparado para confeccionar la web. Sin embargo, tras la incompatibilidad (se quería aprender a hacer todo, no comprender el funcionamiento y copiar), se decidió proceder cada uno con su propia implementación.

Acceso a la web de Vlad, Ainara y Carla, en el navegador:
localhost/ProyectoIngWeb

11. Conclusiones

12. Anexos

Anexo I) Documento de especificación de requisitos

Anexo II) Auditorías de la Implementación A

Prueba 1: página principal

Performance

Metrics

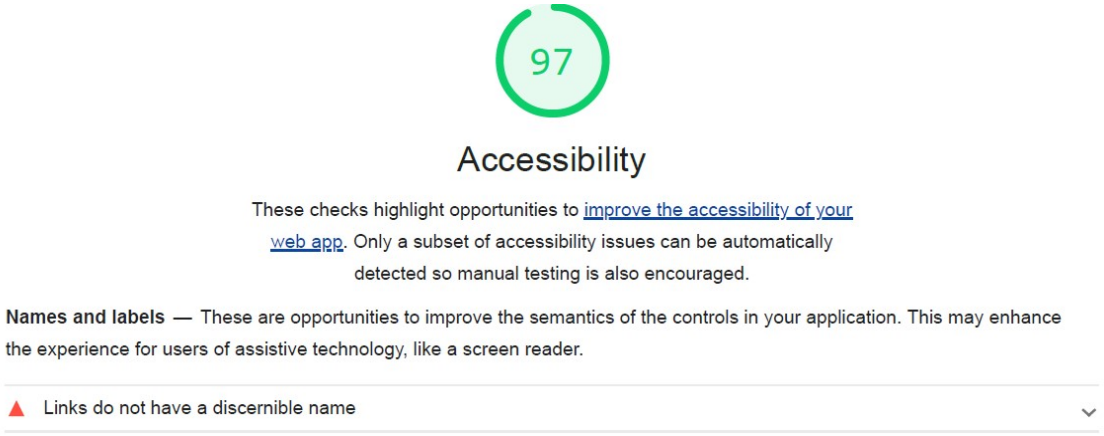
First Contentful Paint	0.9 s	Time to Interactive	1.2 s
Speed Index	0.9 s	Total Blocking Time	0 ms
Largest Contentful Paint	2.2 s	Cumulative Layout Shift	0.006

Prueba 2: listado de recetas

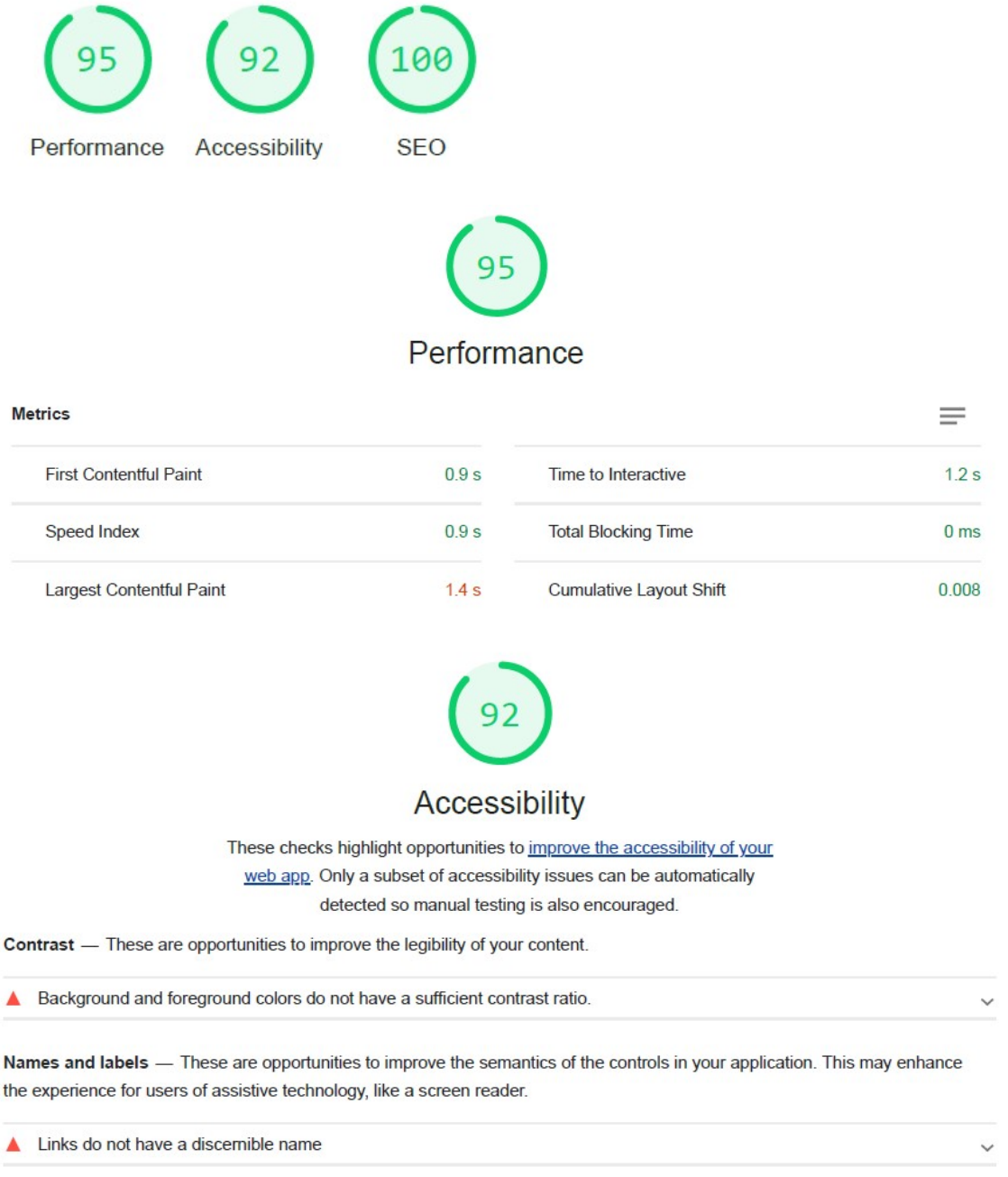
Performance

Metrics

First Contentful Paint	0.9 s	Time to Interactive	0.9 s
Speed Index	0.9 s	Total Blocking Time	0 ms
Largest Contentful Paint	1.2 s	Cumulative Layout Shift	0



Prueba 3: vista de una receta



Prueba 4: galería de imágenes de una receta

Performance

Metrics

First Contentful Paint	0.9 s	Time to Interactive	1.4 s
Speed Index	0.9 s	Total Blocking Time	10 ms
Largest Contentful Paint	1.5 s	Cumulative Layout Shift	0

Prueba 5: perfil de usuario

Performance

Metrics

First Contentful Paint	0.9 s	Time to Interactive	1.0 s
Speed Index	0.9 s	Total Blocking Time	0 ms
Largest Contentful Paint	1.9 s	Cumulative Layout Shift	0

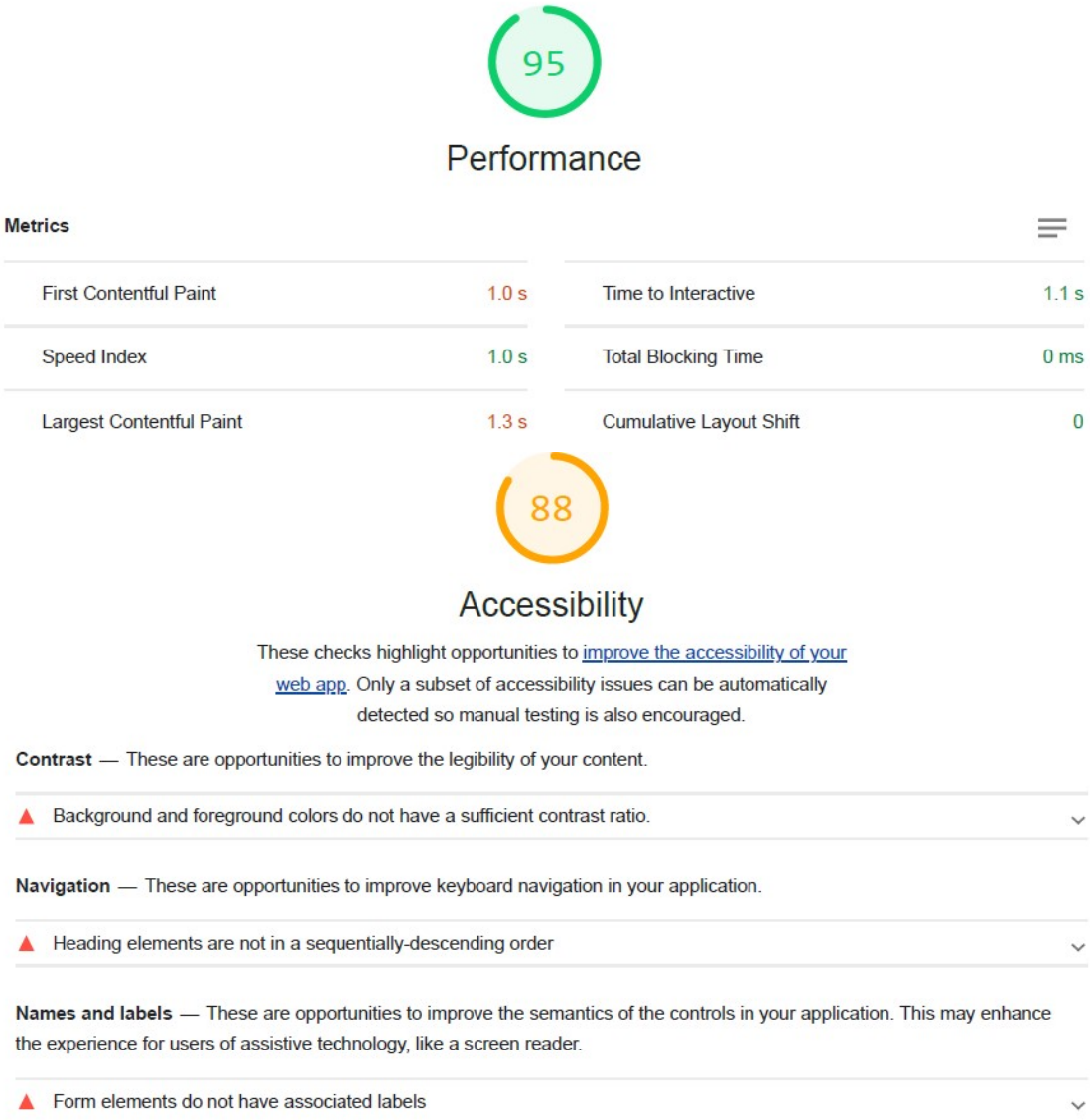
Accessibility

These checks highlight opportunities to improve the accessibility of your web app. Only a subset of accessibility issues can be automatically detected so manual testing is also encouraged.

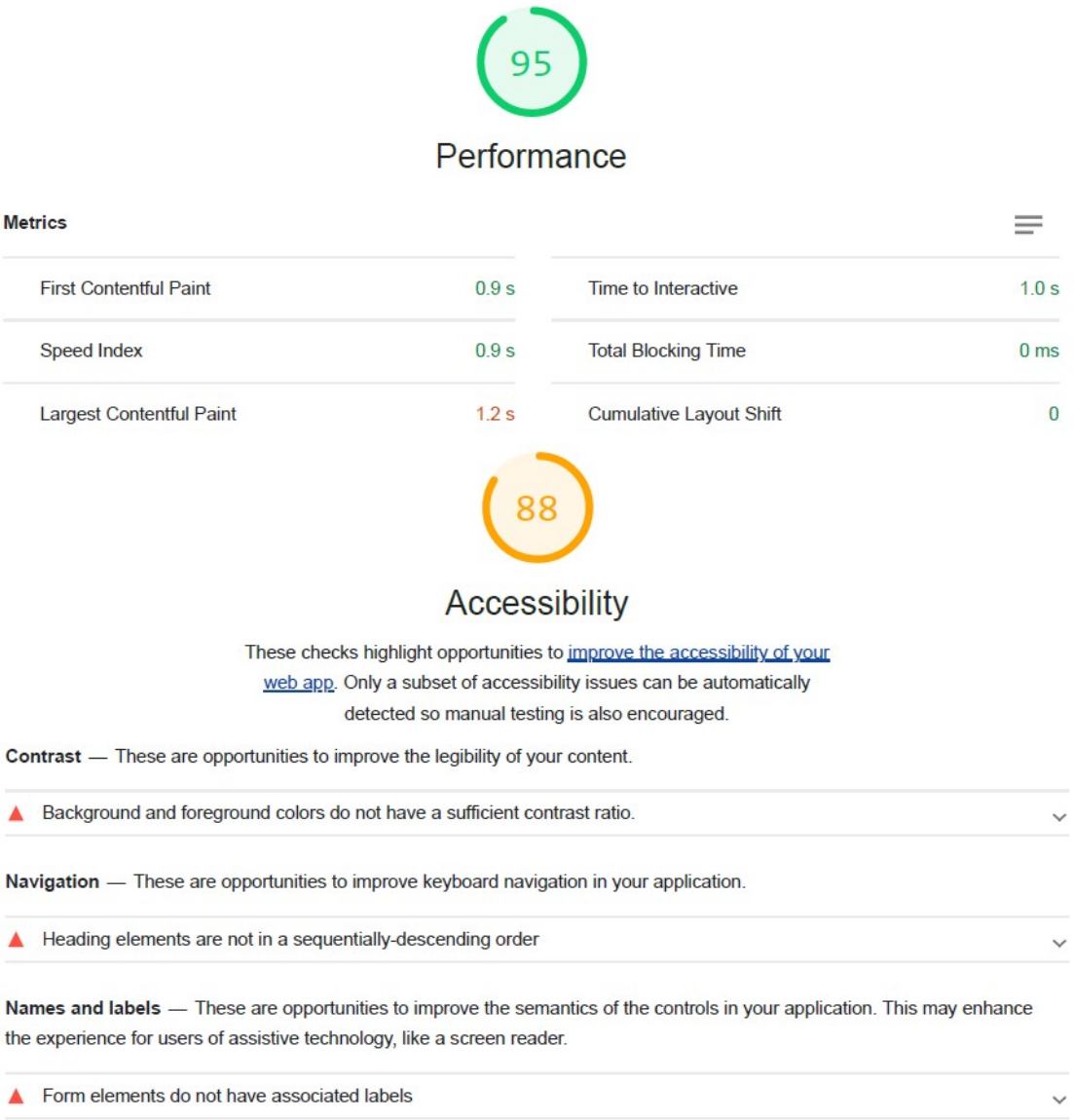
Names and labels — These are opportunities to improve the semantics of the controls in your application. This may enhance the experience for users of assistive technology, like a screen reader.

▲ Buttons do not have an accessible name

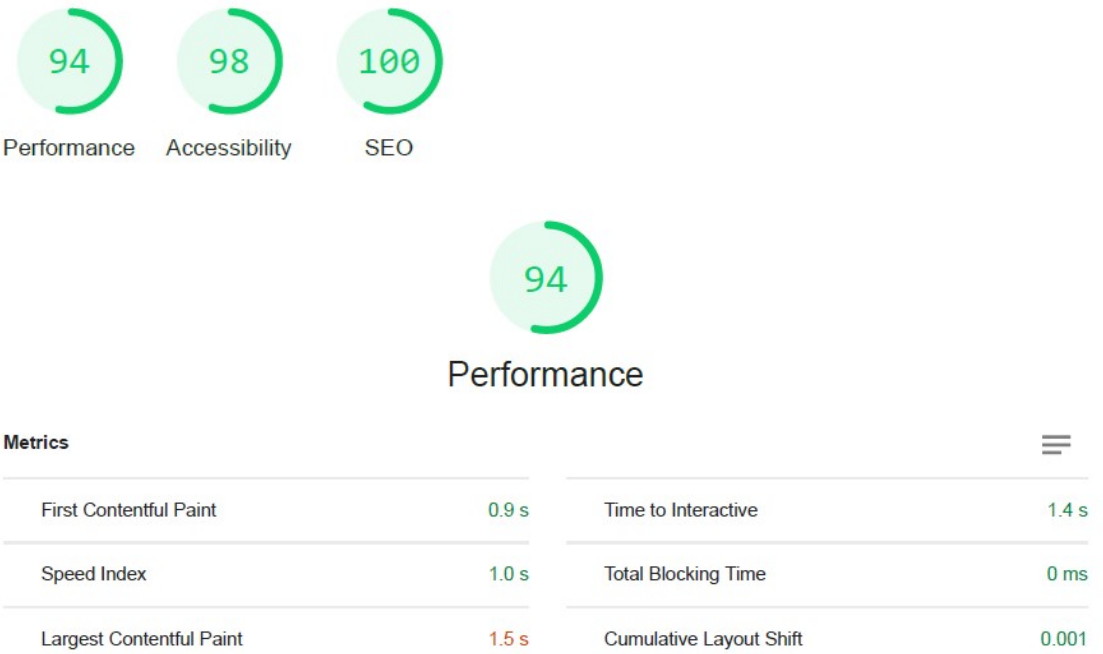
Prueba 5: formulario añadir receta

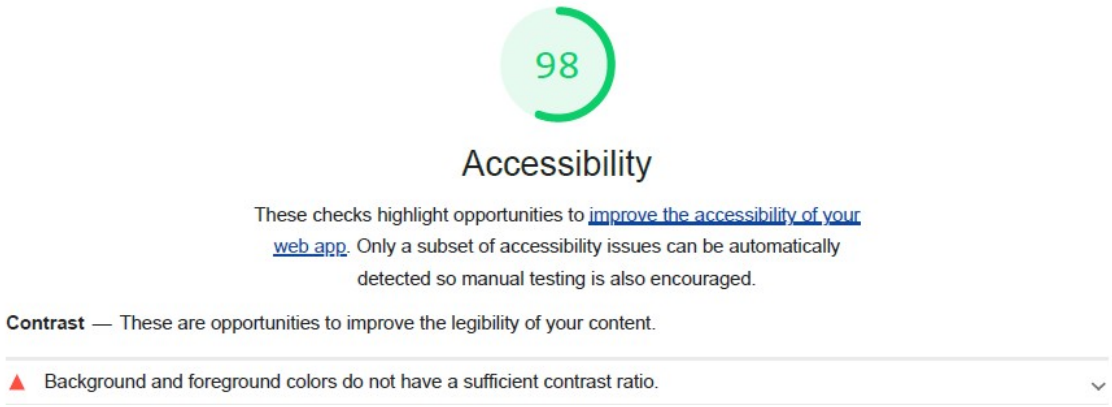


Prueba 7: formulario modificar receta



Prueba 8: formulario registro/ log in





Para obtener información más extensa puede consultar los archivos en la carpeta *Auditorias.*