PLEXUS

Sistema Integral de Reservas de Espacios
(aplicación web)

Análisis, diseño y puesta en marcha inicial

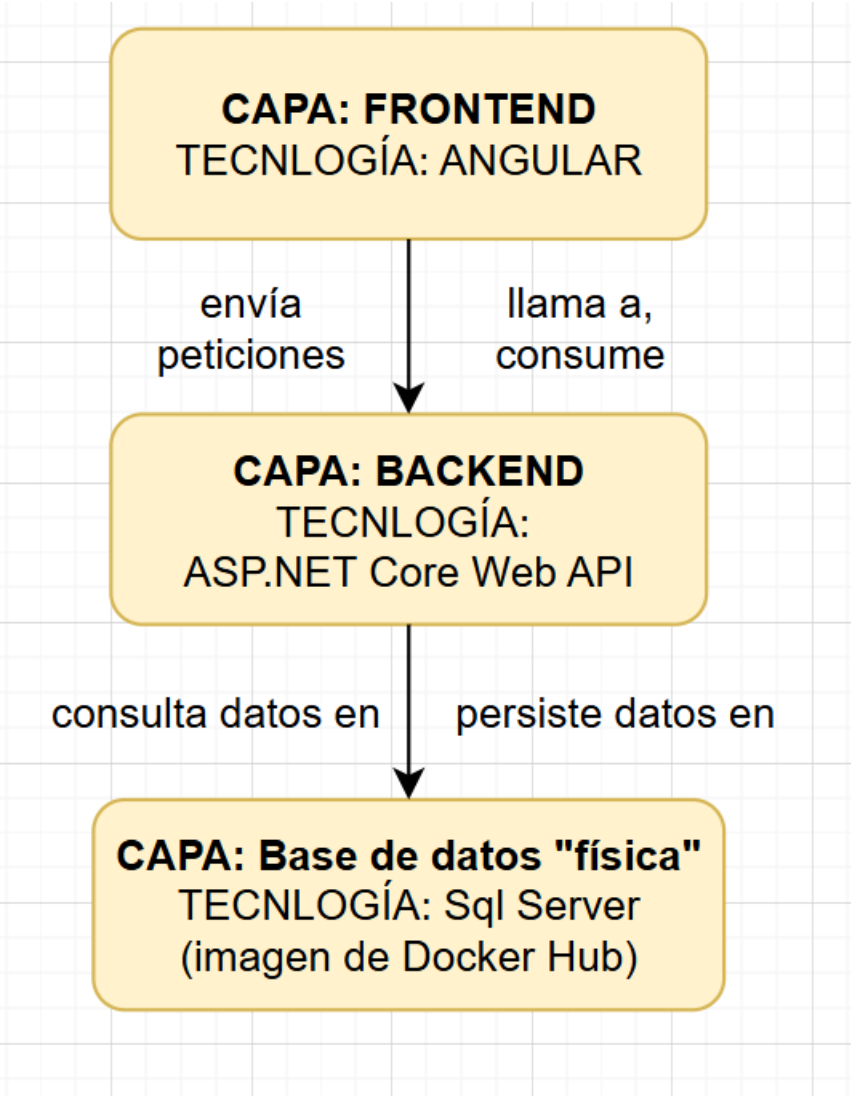
Autoría: Iago Xesús Cadavid González

Fecha: Noviembre de 2025

Índice de contenidos

1. Diseños del sistema software

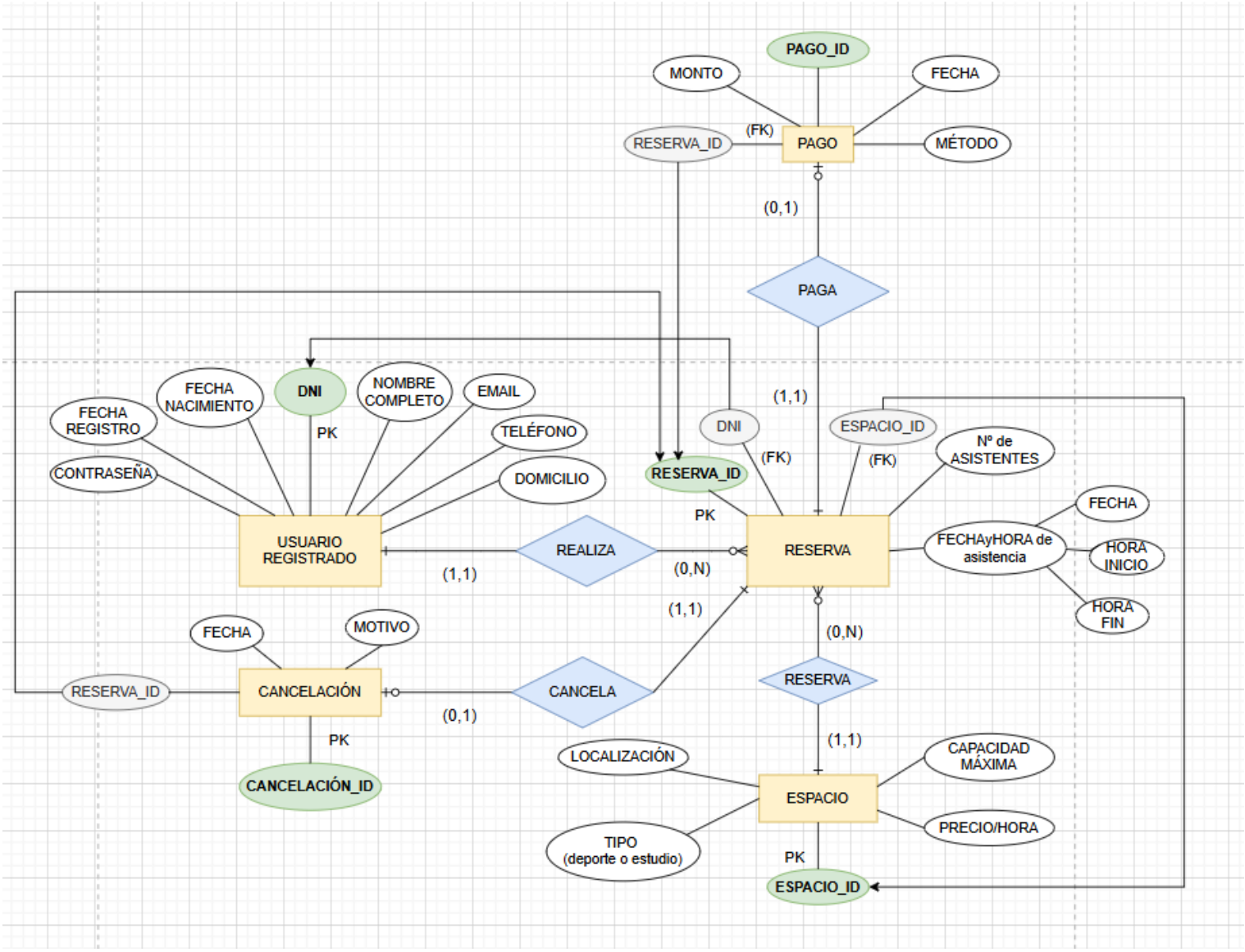
1.1. Diseño de la arquitectura del sistema



Fuente: elaboración propia con draw.io aka app.diagrams.net

¿Cómo incorporo a ese diseño, si es que procede, el volumen y *containers* de Docker Desktop; el servicio Cron; la gestión de eventos con Kafka de la realización/confirmación de resevas?

1.2. Diseño de la base de datos mediante diagrama Entidad-Relación



Fuente: elaboración propia empleando draw.io, actualmente app.diagrams.net

¿Debería incluir alguna tabla de eventos por la gestión de eventos con Kafka?

Explicación de la simbología empleada en mi diagrama:

Con **PK** me refiero a la ***Primary Key*** de la tabla; también la pongo en **negrita** y en verde.

Con (FK) me refiero a que el campo es *Foreign Key*; con una flecha indico a qué PK hace referencia.

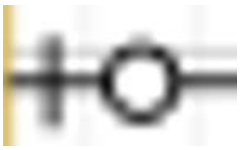
Para la simbología de las relaciones empleé la siguiente referencia:
https://www.smartdraw.com/entity-relationship-diagram/

Nótese que soy redundante en la representación de las relaciones en el siguiente sentido:

Si por ejemplo observamos la relación entre la tabla de usuarios registrados y la tabla de reservas, vemos que las cardinalidades mínima y máxima de usuario registrado respecto de reserva son (0,N) (un usuario registrado puede haber realizado de 0 a varias reservas), pero este (0,N) también lo representa con el siguiente símbolo:

En el otro sentido de la relación, las cardinalidades mínima y máxima de reserva respecto a usuario registrado son (1,1), ya que una reserva siempre es realizada por un único usuario (y no puede existir una reserva sin haber sido realizada por nadie). Este (1,1) también lo represento así:

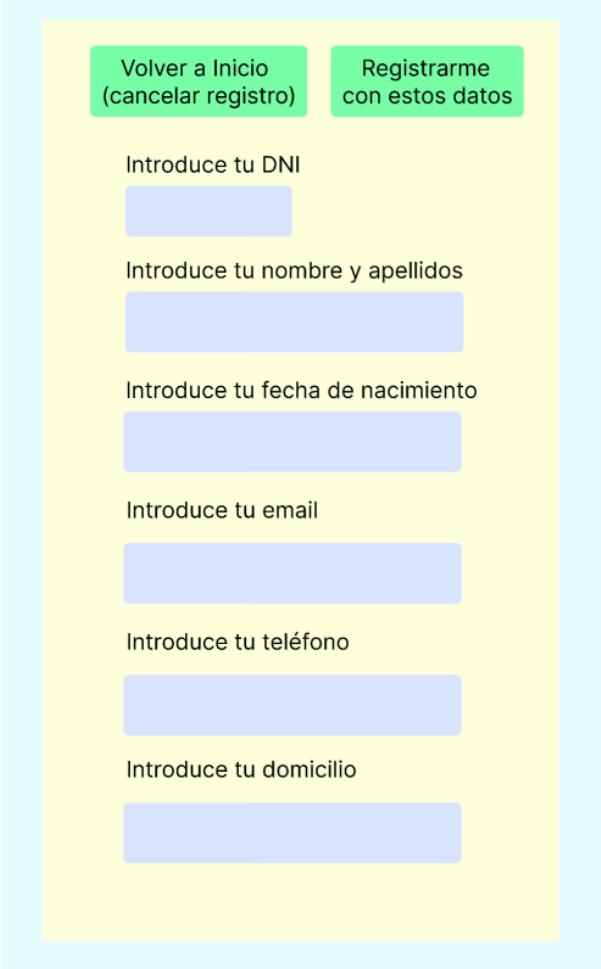
El otro símbolo de relación empleado ha sido el de cardinalidades (0,1):



(PÁGINA SIGUIENTE)

1.3. Diseño de la interfaz de usuario (elaborado con Figma)

PANTALLA HOME PAGE



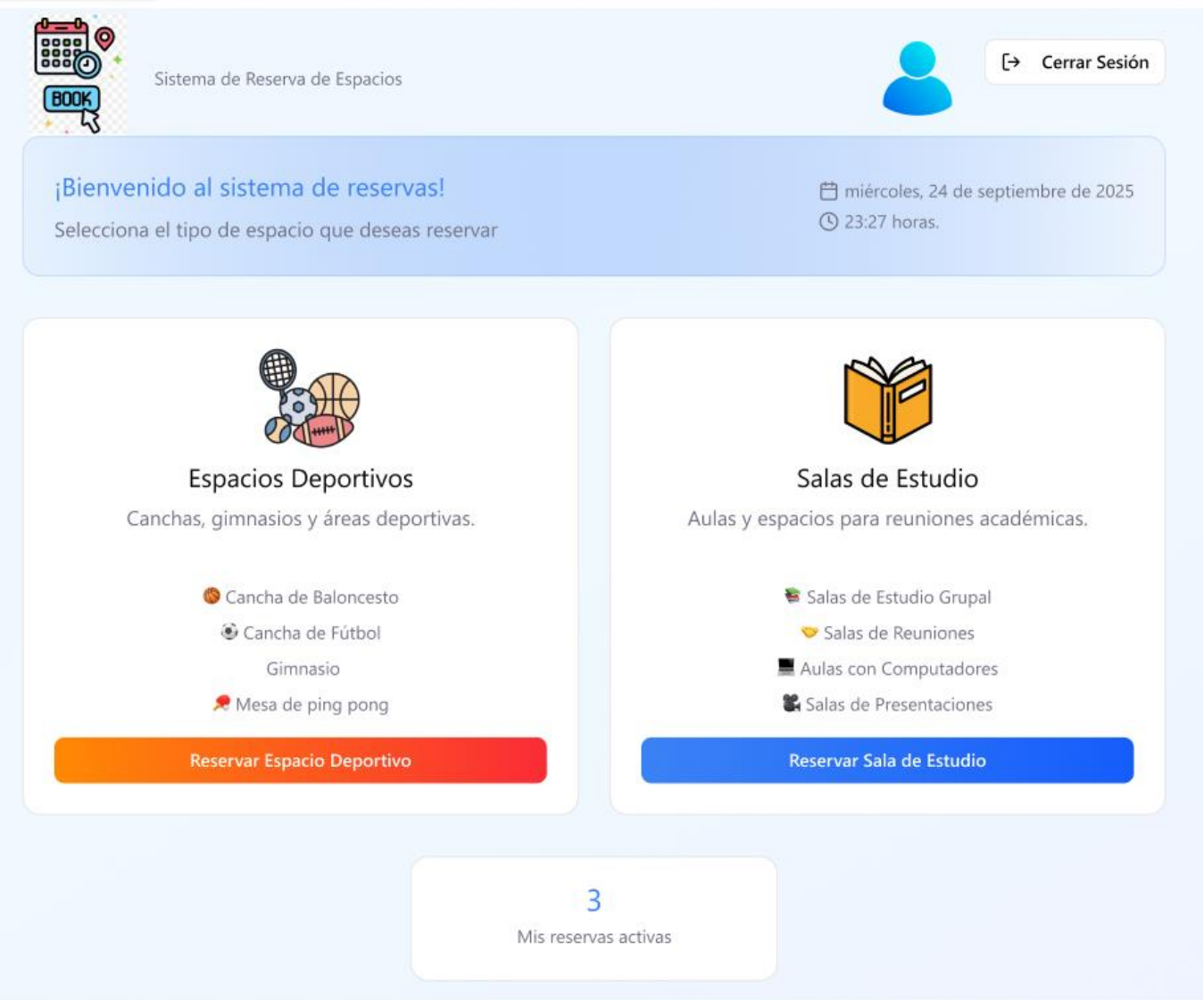
PANTALLA DE REGISTRO

(PÁGINA SIGUINETE)

PANTALLA DE LOGIN

(PÁGINA SIGUIENTE)

Primera pantalla en el proceso de realizar una reserva (a esta pantalla se llega logueándote) :



Eliminé dos botones que había en la parte inferior, a ambos lados de “Mis reservas activas”.

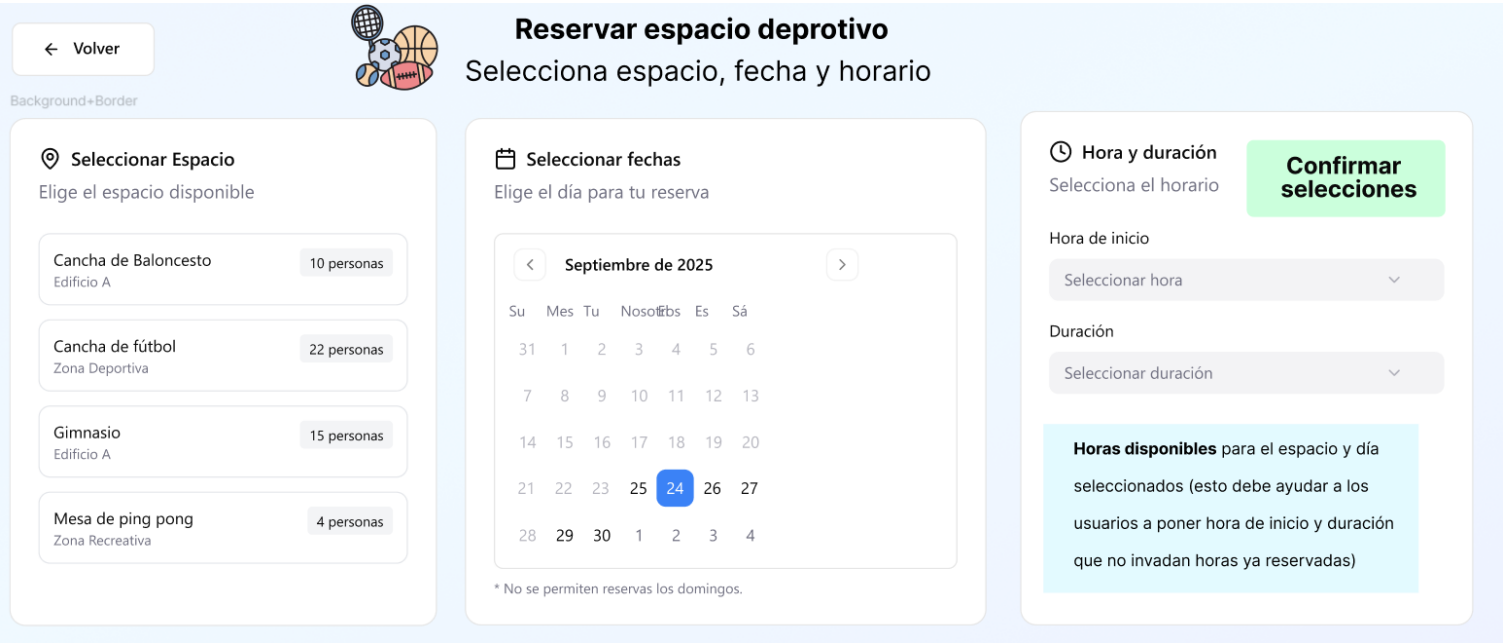
El botón que había a la derecha lo quité porque no entendía para que era.

El botón que había a la izquierda lo quité porque no me tenía mucho sentido; a mi modo de ver, casi siempre estarán disponibles (por lo menos en alguna fecha y hora) todas las instalaciones con las que cuenta la empresa, salvo que alguna esté bajo labores de mantenimiento.

El icono de perfil tendrá que ser el propio del usuario (o uno aleatorio si el usario no estableció ninguno); la idea es que de un vistazo a ese icono, el usuario piense “okay, efectivamente estoy logueado”; amén de que cliando en dicho icono debería poder acceder a la info de su perfil.

(PÁGINA SIGUIENTE)

Segunta pantalla en el proceso de realización de reserva:



(invertí sin querer el 25 y el 24 xd)

El espacio seleccionado + el día seleccionado en el calendario determinarán las “horas disponibles” en el rectángulo de fondo azul claro.

Una vez el usuario haya puesto hora de inicio y duración válidos (osease sin invadir horas no disponibles porque ya están reservadas), podrá clicar en Confirmar selecciones para avanzar a la pantalla de pago.

(PÁGINA SIGUIENTE)

Pantalla análoga a la anterior, pero para reservar salas de estudio

Me falta añadir una pantalla para la realización del pago y con ello confirmación de la reserva.

2. Documento detallado de requisitos

2.1. Requisitos funcionales

En esta sección se describe con detalle qué operaciones/acciones se van a poder realizar a través de la aplicación web, indicando para cada caso: qué actores intervienen, qué acciones realizan, cuáles son los datos de entrada y salida y cuál es el proceso (qué se hace con esos datos).

Se enumeran las funcionalidades para que en la fase de diseño se pueda crear el caso de uso correspondiente a cada una de ellas.

2.1.1. Registro, autenticación y gestión de usuarios (roles: user y admin)

- Actores que intervienen: usuario final y administrador.

- Acciones, datos y proceso:

- Para el registro o *sign up*, el usuario final introduce su DNI, nombre y apellidos, email y teléfono de contacto, domicilio, fecha de nacimiento y el nombre de usuario y contraseña con los que se logueará (el nombre de usuario es prescindible si en su lugar emplea su DNI para loguearse).
- Para el *log in,* el usuario final introducirá sus credenciales (nombre de usuario o DNI + contraseña).
- Para la gestión de usuarios, el usuario final introducirá sus nuevos datos de perfil en caso de editar el mismo.
  El administrador se encargará de la asignación de permisos y roles a los usuarios, así como también tendrá capacidad para observar/auditar la actividad de los mismos y desactivarlos o eliminarlos bajo razón justificada (por ejemplo, por haber observado actividad sospechosa o de algún modo ilícita). Para todo esto, el administrador contará con un panel de administración para gestión de usuarios.

3.1.2. Gestión CRUD de reservas

(Gestión CRUD de reservas para distintos tipos de espacios (deportivos y de estudio, según la segunda layer en Figma).
CRUD de reservas incluyendo la posibilidad de cancelar, modificar o finalizar reservas)

(CRUD = Create, Read, Update y Delete, son las 4 funciones básicas de la persistencia de bases de datos.)

- Actores que intervienen: usuario final y administrador.

(PÁGINA SIGUIENTE)

- Acciones, datos y proceso:

- Durante el proceso de **creación** de una reserva, el usuario final seleccionará en primer lugar qué tipo de espacio desea reservar (espacio deportivo o sala de estudio). En segundo lugar, seleccionará el espacio que desea reservar. Al hacerlo, se actualizará el calendario para mostrar cuándo dicho espacio está o no disponible. En tercer lugar, el usuario final escogerá un día del calendario de entre aquellos días en que el espacio esté disponible. En cuarto lugar, deberá indicar fecha de inicio y fecha de fin i.e. duración de su reserva. También deberá indicar cuántas personas acudirán al espacio reservado.
  Una vez complete el formulario de reserva y lo envíe, le aparecerá en pantalla un mensaje agradeciéndole el haber solicitado la reserva e indicándole que una vez la empresa confirme dicha solicitud, el usuario final recibirá el email de confirmación pertinente. El administrador también podrá crear una reserva a nombre del usuario final en caso de que éste así lo solicite por teléfono.
- Tanto el usuario final como el administrador deben poder **consultar *(read)*** las reservas realizadas/existentes; en concreto, deben poder comprobar los detalles de cada reserva (todo aquello que seleccionó al hacerla) y el estado en que se encuentra la misma (confirmada, pendiente, cancelada)
- En lo que al ***Update*** se refiere, el usuario final debe tener la capacidad de modificar ciertos detalles de su reserva ya realizada, como por ejemplo cuántas personas acudirán finalmente o la fecha y hora (pero en este caso deberá estar todavía disponible el espacio a dicha fecha y hora). Nuevamente, el administrador podrá hacer las mismas modificaciones de detalles de reserva a nombre del usuario si éste así lo solicita por teléfono. Por simplicidad de gestión, si el usuario desea modificar el espacio reservado, deberá eliminar la reserva y crear una nueva. Al confirmarse la modificación, aparecerá en pantalla una notificación al respecto.
- Finalmente, en lo relativo al ***Delete***, el usuario podrá eliminar una reserva ya realizada (pero solamente habrá reembolso si lo hace con suficiente antelación). Igualmente, el administrador podrá eliminar reservas a nombre del usuario final si éste así lo solicita por teléfono. Al confirmarse la eliminación, aparecerá en pantalla una notificación al respecto y se le enviará un email al usuario final.

2.1.3. Consulta de disponibilidad y visualización en calendario

Véase el punto relativo a “creación de una reserva” en gestión CRUD de reservas.

(NEXT PAGE)

2.1.4. Búsqueda y filtrado de reservas por usuario, espacio y fecha

La app contará con un panel de administración para gestión de reservas.

- Actores: administradores.

- Acciones, datos y proceso:

- Por defecto, no aparecerá ninguna reserva hasta que el administrador seleccione los filtros pertinentes (esto será así para evitar que la aplicación se bloquee, deje de funcionar o tarde demasiado en cargar absolutamente todas las reservas).
- En caso de querer consultar las reservas de un usuario concreto, el administrador introducirá el DNI del mismo para así filtrar por dicho usuario.
- Si lo que el administrador necesita es filtrar las reservas por espacio o período de tiempo, independientemente de los usuarios que realizaron las reservas, el administrador debe contar también con ambas posibilidades.
  Cada espacio tendrá un número entero que lo identifique (clave primaria de tabla de espacios en base de datos), de tal modo que el administrador podrá consultar en la tabla de espacios de la base de datos el id del espacio por el que desee filtrar. Si lo que desea es filtrar por *tipo* de espacio, habrá para esto un filtro adicional (y una columna buleana con esta info en la tabla de espacios; buleana porque solo hay dos tipos de espacios, el deportivo y el de salas de estudio).
  Para filtrar por período de tiempo, el administrador introducirá fecha de inicio y fecha de fin del rango temporal que desee consultar; no se tratará de cuándo se realizaron las reservas, sino de *para* cuándo se realizaron (i.e. cuándo se asistirá al espacio).
- Una vez el administrador ha establecido todos los filtros como desee, clicará en un botón de actualización de los resultados, el cual actualizará las reservas que le aparecen en pantalla al administrador. Por si acaso la modificación es poco notable, se le notificará con un Toast de que la consulta se ha actualizado.
- El administrador podrá entonces exportar la reserva que quiera de entre todas las consultadas; para ello habrá dos botones por cada reserva consultada, uno para exportar a PDF y otro para exportar en formato CSV (*comma-separated values*).
- En caso de que el administrador necesite acceder al historial completo de reservas, simplemente seleccionará la opción “sin filtro” en cada uno de los filtros antes mencionados y acto seguido clicará en el botón de “Actualizar”. Mientras la aplicación carga el historial, se le mostrará un mensaje en pantalla indicándole de este hecho.

2.1.5. Gestión de espacios reservables

La aplicación web contará con un panel de administración para gestión de espacios reservables.

- Actores: administradores.

- Acciones, datos y proceso: (PÁGINA SIGUIENTE)

- Los administradores podrán **crear** nuevos **espacios** reservables y **editar** los datos de los ya existentes. Los datos a introducir serán: el tipo de espacio (deportivo o sala de estudio), su nombre, una breve descripción del mismo, su localización y su capacidad máxima (máximo número de asistentes posible). El identificador del espacio será un campo auto-incrementable. Una vez creado o editado un espacio, se le notificará en pantalla al administrador de que la acción ha sido realizada con éxito; idem subsiguiente punto.
- Los administradores también podrán **eliminar espacios** (pej. porque empresa deja de ser propietaria o deja de reservar alguno de ellos) o deshabilitarlos como reservables (pej. porque el espacio en cuestión se encuentra bajo labores de mantenimiento).

2.1.6. Reservas finalizadas mediante eventos gestionados por Kafka

- Actores: Kafka y administradores (consulta de la lista)

- Acciones y proceso:

1. Cuando se aproxima la fecha de fin de la reserva (i.e. cuando los asistentes tienen que abandonar el espacio reservado), **se crea**/genera/dispara **un evento Kafka** para la reserva en cuestión **indicando que la misma está lista para su finalización.**
2. El sistema deberá mantener una lista de las reservas que estén pendientes de finalización. Los administradores deberán poder consultar esta lista.
3. Finalmente, cuando la reserva efectivamente alcance la fecha de fin, se marcará la misma como completada/finalizada en la base de datos y **se generará un evento Kafka indicando que la reserva en cuestión ha sido finalizada**. Esto deberá disparar ciertos procesos, como la actualización de la disponibilidad de las reservas. Si es necesario, se notificará a los servicios pertinentes de que se ha completado el ciclo de vida de la reserva.

2.1.7. Docker Compose para todos los servicios

En Windows y macOS, Docker Desktop incluye Docker Engine, Docker CLI y Docker Compose. Docker Compose es una herramienta que permite definir y ejecutar aplicaciones Docker multi-contenedor usando un único archivo de configuración (docker-compose.yml); simplifica la orquestación de múltiples servicios, entendiendo por "orquestación" la gestión de cómo todos los servicios funcionan en conjunto, la coordinación de todos ellos para que trabajen en conjunto óptimamente.

- Actores: administradores y la propia herramienta Docker Compose de Docker Desktop.

- Acciones:

- Los administradores deben poder iniciar, parar y gestionar todos los servicios de la app de reservas usando –para cada acción- un solo comando de Docker Compose.

- Los administradores deben poder configurar todos los servicios por medio de variables de entorno o archivos de configuración gestionados por Docker Compose.
- También deben tener acceso a través de Docker Compose a *logs*/registros de todos los servicios para monitorear su estado y solucionar/*troubleshoot* problemas.
- Además, deben tener la capacidad de escalar los servicios por medio de ajustar la configuración de Docker Compose sin necesidad de modificar el código de la aplicación.
- Docker Compose debe permitir la intercomunicación de los servicios y la gestión automática de dependencias entre ellos.

2.1.8. Diseño responsive adaptado a dispositivos móviles y pc escritorio/laptops.

- Compatibilidad multi-dispositivo: la aplicación debe renderizarse correctamente tanto en móviles como ordenadores y laptops.
- Debe adaptarse el diseño y el contenido al tamaño de pantalla del dispositivo. Debe mantenerse la usabilidad y legibilidad a través de diferentes resoluciones de pantalla.
- Los elementos interactivos de la aplicación deben ser accesibles vía tacto dactilar en dispositivos móviles y vía ratón/teclado en ordenadores de escritorio y portátiles sin que se pierda funcionalidad.
- La apariencia estética/visual de la app *(look)* y la experiencia de su uso *(feel)* deben mantener una consistencia a través de todos los dispositivos.
- La aplicación debe cargar eficientemente tanto en dispositivos móviles como *desktop*, ajustando para ello imágenes y recursos para coincidir con las capacidades del dispositivo.

En definitiva, el usuario debería ser capaz de emplear todas las funcionalidades de la aplicación con igual comodidad tanto en móvil como pc (y a poder ser también *tablet*). Buena parte de esto consiste en que la interfaz de usuario debería adaptarse automáticamente al tamaño de la pantalla y método de input del dispositivo.

2.2. Requisitos no funcionales

2.2.1. Seguridad

Se debe garantizar que el software protege datos y recursos frente a accesos no autorizados, usos indebidos *(misuse)* o vulneraciones *(security breaches)*. Las medidas de seguridad deberán salvaguardar tanto la información de los usuarios como la integridad del sistema por medio de minimizar el riesgo de accesos no autorizados o de comprometer datos sensibles.

En el apartado de seguridad debemos incluir:

- Encriptación o cifrado de contraseñas: las contraseñas de usuario nunca debemos de almacenarlas como texto plano. En su lugar, deberían estar *hasheadas* de forma segura empleando para ello algoritmos y con un *salting* apropiado para dificultarle a los *hackers* descubrir las contraseñas (*crack passwords*). El "*salting*" consiste en añadirle a una contraseña un string de datos aleatorio y único (el "*salt*") antes de

hashearla. Se creará un hash único para cada contraseña, de tal modo que incluso si dos usuarios tienen la misma contraseña, los *hashes* almacenados serán diferentes (esto previene ataques basados en tablas de arcoiris pre-computadas.

- Validación de sesiones: el sistema debe verificar la autenticidad y validez de las sesiones de usuario para evitar *hijacking*/secuestros de sesión o acceso no autorizados. Esto involucra gestionar de modo seguro tokens de sesión, imponer tiempos de expiración de sesión por inactividad (*session timeouts*) e invalidar sesiones tras *logout* o tras un periodo de tiempo predefinido.
- Control de roles y permisos: el sistema software debe implementar un mecanismo claro de autorización que defina qué acciones pueden ser realizadas por cada tipo de usuario. Roles típicos son el de administrador, editor o solo *viewer*. Los roles deben ir asociados a permisos específicos para garantizar que los usuarios solo puedan acceder a datos y funciones que se correspondan con sus responsabilidades.

2.2.2. Rendimiento y escalabilidad

El rendimiento se refiere a cuán eficientemente responde el sistema software a acciones del usuario, procesamiento de datos y manejo de carga de trabajo *(workload handling)*. El rendimiento contribuye a determinar la percepción que el usuario tiene de la velocidad, *responsiveness* (en el sentido de la capacidad del sistema software de reaccionar rápidamente al input del usuario) y la calidad general del sistema.

En el apartado del rendimiento debemos tener en cuenta los siguientes aspectos:

- Tiempo de respuesta: para operaciones críticas, el sistema software debe responder en no más de 2 segundos bajo condiciones normales de carga de trabajo. Las operaciones críticas incluyen la autenticación, el acceso y extracción (*data retrieval*) de datos almacenados en la base de datos y el procesamiento de transacciones. Con respecto a las operaciones no críticas, serán tolerables tiempos de respuesta ligeramente mayores a 2 segundos, pero igualmente se debe mantener la fluidez del sistema y la *responsiveness* ante input o acciones del usuario.
- Escalabilidad (v.t. 2.2.4.) y *throughput**:
  El sistema software debería ser capaz de procesar múltiples peticiones/*requests* simultáneamente sin degradación significativa del servicio. A medida que se incrementa el número de usuarios o transacciones, la *performance* debería degradarse de manera controlada o escalar horizontalmente por medio de balanceo de carga o procesamiento distribuido, manteniendo así tiempos de respuesta aceptables.
  * El *throughput* es una métrica fundamental del rendimiento del sistema software y se refiere a la ratio a la cual el sistema software procesa o transmite una cantidad dada

de datos o trabajo dentro de un rango temporal específico; frecuentemente se mide en peticiones por segundo (RPS) o bits por segundo (bps).

- Rendimiento o *performance* bajo carga de trabajo: el sistema software deberá ser testeado en condiciones de estrés osease en condiciones de picos esperables de trabajo, con el objetivo de garantizar que las operaciones críticas rendirán adecuadamente en tales condiciones. A conseguir esto pueden ayudar el empleo de mecanismos como el *caching*, el procesamiento asíncrono o la realización de consultas (a la base de datos) optimizadas; todo esto contribuye a la *responsiveness* del sistema.
- El sistema debería contar con herramientas de monitoreo continuo (como por ejemplo software APM –*Application Performance Management*-) para llevar un control de los tiempos de respuesta, tasas de error y empleo de recursos en tiempo real. Todo esto permite detectar prematuramente problemas de rendimiento y llevar a cabo una optimización proactiva.
- Optimización y *tuning*: el código, las consultas y las configuraciones del sistema deberían ser revisadas periódicamente para garantizar un rendimiento sostenido en el tiempo, en especial según vayan incrementándose el volumen de datos y el número de usuarios.
- Utilización de recursos: el consumo de CPU, memoria y *network* debería mantenerse en *thresholds* aceptables para evitar cuellos de botella. El sistema software debe estar diseñado para hacer uso eficiente del hardware disponible y evitar computaciones o transferencia de datos innecesarias.

2.2.3. Mantenibilidad/Maintenance del sistema software

- Una de las estrategias más fundamentales y efectivas para logar un software mantenible (así como también escalable y testeable) es **estructurar su código en diferentes capas**, lo que se conoce como arquitectura por capas o *n-tier architecture*. Esta arquitectura organiza el código en diferentes capas lógicas o módulos, cada uno con su propia responsabilidad. Esta separación en capas permite aislar las preocupaciones *(isolate concerns)*, lo cual hace que el sistema software sea más fácil de comprender, mantener, testear y ampliar o extender (escalar) a lo largo del tiempo.

  Una típica estructura por capas es la siguiente: (PÁGINA SIGUIENTE)

- Capa de presentación o Capa de interfaz de usuario (***UI Layer)***: maneja las interacciones de los usuarios, presentándoles la información y recibiendo su input. En el caso de este sistema software, esta capa se corresponderá con un ***frontend* web** elaborado **con Angular**.

- **Capa de aplicación o** capa **de servicios**: esta capa coordina el flujo de datos entre la capa de UI y la capa de lógica de negocio.

- **Capa de lógica de negocio**: define lo qué el sistema software hace desde un punto de vista de negocio, no define cómo se presenta (la UI) ni de dónde vienen los datos (*database*). En otras palabras, implementa cómo el negocio realmente funciona, de modo independiente a cómo se almacenan los datos (*database*) y a cómo los usuarios interactúan con la aplicación (UI).

  En la capa de lógica de negocio se determinan las reglas y procesos de decisión, tales como la exigencia de que el usuario esté logueado antes de hacer una resserva, la imposibilidad de reservar el mismo espacio dos veces a la misma fecha y hora o la posibilidad de cancelar reserva solo dentro de un rango de tiempo determinado (por ejemplo, no dentro de las 24 horas antes de la hora de asistencia).

- **Capa de acceso a datos** o capa **de persistencia**: maneja las interacciones con la base o fuente de datos.

  En lo que a la persistencia de datos respecta, para el *backend* emplearemos tecnología **.NET (API RESTful).** En otras palabras, nuestro *backend* será una API que se consumirá.

  Muy probablemente emplearemos la herramienta de gestión de base de datos **DBeaver** para gestionar **una base de datos que estará corriendo dentro de un contenedor Docker**. Docker correrá un motor de base de datos por nosotros dentro de un contenedor aislado. Para ello, **nos bajaremos una imagen Docker de la base de datos Sql Server.** Al hacer esto, no necesitaremos instalar un motor de base de datos como SQL Server en nuestro ordenador.

En esa estructura de capas, cada capa se comunica solamente con la que tiene directamente debajo suya, estableciendo así fronteras o límites claros y reduciendo el *coupling*.

- Una de las herramientas más fuertes que existen para mejorar la mantenibilidad del software es la **documentación técnica**. Esta documentación debería incluir, entre posiblemente otros, los siguientes:

  - **Documentación de la arquitectura** del sistema: describe cómo el sistema software se estructura en general, lo que los anglosajones llaman la *big picture*.

- **Documentación del código**: describe cómo el código funciona internamente, esta es la documentación más próxima al código fuente de la aplicación. Explicar qué hace algo el código y por qué.
- **Documentación de la base de datos** y del modelo de datos: explica cómo se almacenan los datos y cómo se relacionan con el sistema en su conjunto. Muy importante elaborar un diagrama entidad-relación de la base de datos, de las tablas, sus campos y las relaciones entre ellas (*primary key-foreign key*).
- **Documentación de despliegue y configuración** del sistema software: una guía sobre cómo configurar (*set up*), ejecutar (*run)* y desplegar (*deploy*) el sistema software. En nuestro caso esta documentación versará, entre posiblemente otros, sobre los archivos de configuración de Docker Compose. Esta documentación también suele incluir las instrucciones de instalación (dependencias, configuración del entorno) y la descripción de los *pipeline* CI/CD.
- **Documentación de *testing***: describe cómo se estructuran y ejecutan los tests/pruebas de la aplicación, con el propósito de garantizar que los cambios al software no destruyan la funcionalidad ya existente. Los tests se suelen dividir esencialmente en dos tipos: **tests unitarios** y **tests de integración**.
- Si procede, **documentación API** que describa cómo sistemas o clientes externos pueden interactuar con nuestro sistema software.

2.2.4. Sistema dockerizado para fácil despliegue

El sistema software deberá estar debidamente contenedorizado usando Docker (usaremos Docker Desktop y en particular su herramienta Docker Compose) por varias razones: para simplificar el despliegue/*deployment* de la aplicación, para garantizar su consistencia a través de diferentes entornos y también para facilitar la escalabilidad (v. 2.2.2) y el mantenimiento (v. 2.2.3).

Este requerimiento no funcional garantizará que el sistema software pueda ser fácilmente instalado, configurado y ejecutado en diferentes máquinas sin conflictos de dependencias ni un *setup* demasiado complejo.

El proceso de despliegue/*deployment* del sistema software debería ser automatizado por medio de Docker Compose, que es una herramienta de orquestación (*orchestration tool*) de la aplicación Docker Desktop. Gracias a **Docker Compose**, el sistema software podrá ser arrancado (*started*) con un único comando (docker-compose up) o integrado en *pipelines* CI/CD para un despliegue continuo.

En materia de escalabilidad, la contenedorización soporta escalabilidad horizontal osease múltiples instancies de contenedor podrán ser ejecutadas para manejar cargas de trabajo más grandes.

Además, la contenedorización de Docker simplifica el mantenimiento del sistema software, ya que tanto actualizaciones como *rollbacks* podrán realizarse por medio de re-desplegar determinados contenedores sin afectar al sistema en su conjunto.

2.2.5. Pruebas automatizadas e integración automatizada en CI

El *testing* automatizado garantiza que partes críticas del sistema software son verificadas automáticamente por cada cambio que se le hace al código, reduciendo así el error humano y evitando regresiones.

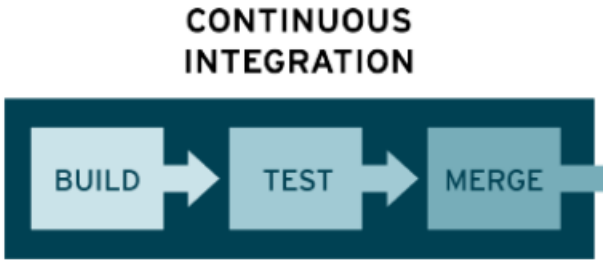
Nuestro sistema software deberá incluir un conjunto de **pruebas automatizadas** para verificar la funcionalidad y estabilidad de la aplicación.

- Las pruebas automatizadas de ***backend*** deberán abarcar por lo menos el 60 por ciento del código *backend* osease dichos tests deberán ejecutar por lo menos un 60 por ciento de ese código. Para testing unitario automatizado del backend (el cual recordemos será elaborado con tecnología .NET), emplearemos **xUnit**.
- En lo que al ***frontend*** respecta, las pruebas automatizadas deberán abarcar/ejecutar por lo menos la mitad del código *frontend*. Para testing unitario automatizado del frontend (el cual recordemos será elaborado con Angular), emplearemos las herramientas **Jasmine** y **Karma**.

Tanto las pruebas automatizadas de *backend* como las de *frontend* deberían incluir tanto **tests unitarios** como **tests de integración**. Esto en *frontend* significa testear componentes UI aislados (test unitario) y testear cómo los componentes de UI funcionan o trabajan en conjunto por ejemplo para la navegación entre pantallas (test de integración).

En cuanto a la **integración automatizada en CI**, lo que esencialmente se pretende satisfacer con este requisito no funcional es que cada vez que un desarrollador hace un *push* de su código, el sistema automáticamente compile (*build*), ejecute los tests de *backend* y *frontend* y valide los cambios (que dicho código introduzca) antes de llevar a cabo

la integración osease el *merge* del nuevo código con la *main codebase* (i.e. la rama main o master del proyecto software). Un esquema visual del CI sería el siguiente:



Fuente: Red Hat

2.2.6. Disponibilidad, monitoreo y *logging*/registro de problemas

La disponibilidad es una medida de cuán frecuentemente el sistema software está operativo y accesible a los usuarios. En nuestro caso, **pretendemos que el sistema sea capaz de operar de modo continuo, con un mínimo *downtime*** (i.e. periodo de tiempo en que el sistema software no está disponible a los usuarios). En otras palabras, nuestros usuarios deberían ser capaces de realizar acciones en cualquier momento, algo que consideramos crítico en servicios de *booking* como lo es el nuestro. Inclusive el mantenimiento planificado del sistema deberá ser diseñado para minimizar las interrupciones en el uso de la app.

Para garantizar ese nivel de disponibilidad, será fundamental contar con un sistema de monitoreo y registro de problemas que sea capaz de detectarlos lo antes posible y de describir las causas de dichos problemas, para así poder resolverlos lo antes posible, minimizando el *downtime* de nuestra aplicación. En este sentido, se deberán realizar:

- Chequeos de la salud del sistema software: verificar que los servicios de la app están ejecutándose y respondiendo correctamente a las acciones de los usuarios.
- Seguimiento de errores (*error tracking*): loguear errores y alertar de los mismos.
- Métricas de rendimiento o *performance*: medir empleo de CPU y memoria, tiempos de respuesta, tasas de solicitud (*request rates*, medidas en RPS i.e. *requests per second*).

Para poder resolver problemas de disponibilidad *(troubleshooting availability issues),* será fundamental loguear los siguientes:
- Los errores del sistema y excepciones que se produzcan en el mismo
- Peticiones o solicitudes fallidas y errores de base de datos
- Fracasos en la autenticación de usuario
- Avisos sobre límites de recursos (CPU, memoria, espacio de disco)

2.2.7. Usabilidad: interfaz intuitiva y accesible

No solo es importante lo que la aplicación hace o permite hacer, sino también la experiencia del usuario al usarla, el *feel* que tiene al emplear nuestra aplicación. En este sentido, la **interfaz de usuario** ha de ser **intuitiva**, es decir, fácil de usar, y **accesible**, es decir, que personas con discapacidades puedan igualmente hacer uso de nuestra aplicación web.

- Que la UI sea **intuitiva** requiere que los usuarios puedan saber usar la aplicación con ningún o mínimo entrenamiento para ello.
- En materia de **accesibilidad**, la **WCAG 2.1 AA** es un estándar de accesibilidad web ampliamente reconocido, desarrollado por el World Wide Web Consortium (W3C), que define cómo debe diseñarse el contenido digital para que sea accesible para todas lasp ersonas –incluyendo aquellas con discapacidad (eye-able.com).
  Dicho estándar incluye los siguientes requisitos sobre la UI:
  - **Perceptible**: el contenido debe presentarse de manera que el usuario pueda percibirlo, lo cual pasa por ofrecer alternativas textuales para las imágenes, subtítulos o transcripciones para vídeos y que el contraste de color sea fuerte).
  - **Operable**: la UI debe ser utilizable para todas las personas; para ello, se debe garantizar: la accesibilidad a través del teclado, que no haya contenido parpadeante y que la UI sea de fácil navegación.
  - **Comprensible**: los usuarios deben poder comprender el contenido y el uso de la interfaz. El comportamiento de la app debería ser predecible y, en el caso de que puedan surgir dudas, debería haber etiquetas (especialmente en formularios) e instrucciones que resuelvan las mismas. En suma, el lenguaje empleado en la interfaz debe ser claro.
  - **Robust**: la interfaz debería funcionar correctamente a través de diferentes dispositivos y navegadores (*browsers*). El contenido debe ser accesible en diferentes formatos y dispositivos.

En definitiva, la usabilidad implica que toda persona pueda usar nuestra aplicación fácilmente y de manera efectiva tanto si se trata de un nuevo usuario o de una persona con algún tipo de discapacidad; para satisfacer la usabilidad, es conveniente alinear nuestra app con los estándares WCAG 2.1 AA.

2.2.8. Fiabilidad

(Tolerancia a fallos en consumo de eventos Kafka y manejo de errores automatizado)

La fiabilidad consiste en garantizar que el sistema software continúa operando correctamente incluso cuando algo va mal. Se trata de lograr la estabilidad del sistema, la tolerancia ante fallos y la resiliencia ante errores.

En el caso de nuestro sistema software, debemos prestar especial atención a que nuestro sistema garantice la **tolerancia ante fallos en el consumo de eventos y procesamiento de mensajes Kafka**.

Para ello, se deberán implementar mecanismos de tolerancia ante dichos fallos, como pueden ser el reintentar automáticamente el procesamiento de un evento fallido o mover los mensajes que repetidamente fallan a un *topic* aparte para ser analizados manual o automáticamente (a esto se le conoce como d*ead-letter queues* –DLQ-).

También de cara a cumplir con el requisito no funcional de fiabilidad del sistema software, deberemos disponer de herramientas o librerías que nos permitan **manejar de modo automático los errores**, reduciendo así la intervención humana.

Típicas prácticas de manejo automático de errores incluyen logueo/registro automático de errores con contexto para *troubleshooting*, lanzamiento automático de alertas antes fallos críticos del sistema o los ya mencionados reintentos automáticos e implementación de DLQs

2.3. Criterios de aceptación (v. sección 2.1 para mayor detalle)

Entendemos por criterios de aceptación un conjunto de condiciones específicas y medibles que un sistema de software debe cumplir para ser considerado completo y funcional. En el caso de nuestra aplicación web de reserva de espacios deportivos y salas de estudio, dichas condiciones exigibles son –por lo menos- las siguientes:

1. **El usuario puede registrarse en nuestra aplicación web** introduciendo los datos pertinentes, tales como: DNI, nombre y apellidos, fecha de nacimiento –con esto se calculará su edad y se irá actualizando conforme pase el tiempo, email, teléfono, domicilio, contraseña de acceso a nuestra app web.
   Se comprobará que todos los campos obligatorios fueron cubiertos, que el DNI cumple con el formato de 8 números y una letra y que la contraseña cumple con los requisitos de seguridad pertinentes (por lo menos 8 caracteres incluyendo números y empleo de signos o símbolos tales como ‘!’ o ‘?’).
2. **El usuario podrá loguearse en nuestra aplicación web,** teniendo para ello la capacidad de introducir su nombre de usuario (que posiblemente será el propio DNI)

y la contraseña que estableció en el registro (o la nueva si es que la modificó desde su perfil).

3. Una vez **logueado, el usuario podrá realizar una o varias reservas** (pero siempre de una en una).

   La aplicación deberá saber en todo momento cuáles son los datos de perfil del usuario que ya logueado se encuentra en proceso de cubrir el formulario de realización de reserva (osease el usuario no debe tener la necesidad de volver a introducir sus datos de perfil que ya estableció en el registro).

   El usuario deberá poder seleccionar el tipo de espacio a reserva (espacio deportivo o sala de estudio) y posteriormente qué espacio concreto quiero y para qué fecha y hora.

   A ese respecto, una vez elegido el espacio, se le mostrará al usuario en un calendario las fechas y horas en que dicho espacio está actualmente disponible. La aplicación NO debe permitir que se reserve dos veces el mismo espacio a la misma fecha y hora.
4. **El usuario deberá poder acceder a sus datos de perfil y modificarlos**, ya que será el responsable de actualizar los mismos en caso de que dichos datos cambien. También **deberá poder eliminar su perfil de usuario**, perdiendo con ello la posibilidad de loguearse y por ende de reservar.
5. **El usuario deberá poder modificar ciertos datos de su reserva** después de haberla realizado. También **deberá poder eliminarla**; en este caso, solo se concederá reembolso si se canceló con suficiente antelación.
6. **Los administradores de nuestra app software deberán poder crear y modificar una reserva a nombre del usuario**, si es que éste así lo solicita por llamada telefónica.

   Asimismo, los administradores **deberán tener la capacidad de asignar roles y permisos a los diversos usuarios de la app**, así como también **podrán observar/auditar la actividad de los usuarios y desactivarlos o eliminarlos bajo razón justificada** (por ejemplo, por haber observado actividad sospechosa o de algún modo ilícita). Para todo esto, el administrador contará con un panel de administración para gestión de usuarios.
7. **Los administradores podrán crear nuevos espacios reservables y editar los datos de los ya existentes. También podrá eliminar espacios reservables**, en caso de que ya no lo sean realmente por la razón que sea.
8. **Los administradores del sistema deberán poder buscar y consultar las reservas existentes filtrando por usuario, espacio y fecha de la reserva** (ésta será la fecha de asistencia, no la fecha en la que se realizó la reserva).
9. **Los administradores deben poder iniciar, parar y gestionar todos los servicios de la app de reservas usando** –para cada acción- un solo comando de **Docker**

Compose. En otras palabras, **la solución podrá ser desplegada y ejecutada sin errores por un administrador usando** un solo comando de **Docker Compose.**

10. **Cada funcionalidad testeada debe disponer de pruebas unitarias y de integración** asociadas y documentadas. En la fase de diseño deben plantearse estas pruebas.
11. **El sistema responde correctamente bajo simulación de carga de 50 usuarios concurrentes.**
12. **Al finalizar automáticamente una reserva** (por fecha –la del fin de asistencia al espacio reservado- u acción de usuario -cancelación de reserva-) **se emite un evento Kafka y el usuario es notificado**.
13. **Todas las acciones de usuarios y administradores que convenga notificar, se notificarán**, o bien directamente en pantalla (pej. “se han modificado sus datos de usuario"), o bien por email (pej. “confirmación de aceptación por parte de la empresa de la reserva solicitada por el usuario”).
14. La aplicación debe renderizarse correctamente tanto en móviles como ordenadores y laptops.
15. Debe adaptarse el diseño y el contenido al tamaño de pantalla del dispositivo.
16. Los elementos interactivos de la aplicación deben ser accesibles vía tacto dactilar en dispositivos móviles y vía ratón/teclado en ordenadores de escritorio y portátiles.
17. La apariencia estética/visual de la app *(look)* y la experiencia de su uso *(feel)* deben mantener una consistencia a través de todos los dispositivos.
18. Finalmente, la aplicación debe cargar eficientemente tanto en dispositivos móviles como *desktop*.