



Universidad ORT Uruguay

Facultad de Ingeniería

Documentación Obligatorio 2

Diseño de Aplicaciones 1

Link al Repositorio: https://github.com/IngSoft-DA1-2023-2/231810_280070_301178

Integrantes:

- Angelina Maverino - 280070
- Yliana Otero - 301178
- María Belén Rywaczuk - 231810

Tutores:

- Mario Souto
- Joaquín Mendez
- Emiliano Hernandez

Índice

Descripción del trabajo.....	3
Cambios realizados.....	3
Descripción y justificación de diseño.....	3
Diagrama de paquetes.....	4
Diagramas de clases.....	4
Clases del Dominio (en DepoQuick).....	5
Controllers con exceptions (en BusinessLogic).....	6
Dependencias del dominio y los controllers.....	6
Relación entre los controllers y IRepository (intermediario de la base de datos).....	7
Diagramas de interacción.....	7
Exportar reporte de reservas.....	7
Reject Reservation.....	7
Login.....	8
Modelo de tablas.....	9
Principales decisiones de diseño.....	9
Correcciones de la primera entrega.....	9
División de controladores.....	10
Interfaz IRepository<T>.....	10
Separación de BusinessLogic y DepoQuick.....	11
Manejo de fechas de disponibilidad.....	11
Análisis de los criterios seguidos para asignar las responsabilidades.....	11
Análisis de dependencias.....	14
Exportar reporte de reservas.....	14
Dependencias.....	15
Análisis del nivel de cohesión y acoplamiento.....	15
Cobertura de pruebas unitarias.....	15
Anexos.....	17
Anexo 1 - Tabla de responsabilidades.....	17
Anexo 2 - Guía de ejecución.....	22

Descripción del trabajo

En esta segunda parte del proyecto obligatorio, decidimos trabajar en equipo desde el principio para definir el diseño de nuestro programa. Tomamos en cuenta los puntos destacados por los profesores en la devolución del primer obligatorio, lo que nos llevó a realizar varios cambios en el diseño original. Además, tuvimos que adaptar nuestro proyecto a la base de datos introducida, lo que implicó convertir los atributos en propiedades y crear constructores sin parámetros para que Entity Framework pudiera crear las tablas automáticamente.

Todo este trabajo se realizó utilizando TDD y aplicando los conocimientos adquiridos en las clases teóricas y prácticas. Nos basamos en principios de Clean Code, SOLID y GRASP para mejorar el diseño y la calidad del código. La base de datos fue implementada con Entity Framework, lo que garantiza la persistencia de los datos.

También cumplimos con los requerimientos adicionales para tres estudiantes sobre las notificaciones. Pusimos foco en que nuestro proyecto fuera flexible a cambios, especialmente en áreas donde previmos que podrían surgir modificaciones, como la exportación de reservas, que se discutirá más adelante.

En conclusión, el programa desarrollado permite la creación de un administrador único y varios clientes, así como la realización de diversas operaciones relacionadas con la reserva y gestión de depósitos, además de la creación y gestión de promociones. La funcionalidad del sistema varía según si el usuario se identifica como cliente o administrador al iniciar sesión, permite además gestionar los pagos y las reservaciones de tal manera que no se produzcan conflictos y realizar exportaciones de los datos importantes de las reservaciones, incluidas las rechazadas.

El proyecto se llevó a cabo siguiendo el enfoque de Desarrollo Dirigido por Pruebas (TDD). Lo que nos brindó una sólida garantía de que las nuevas funcionalidades agregadas funcionarán según lo previsto.

Para ver cómo correr la aplicación, leer el segundo anexo.

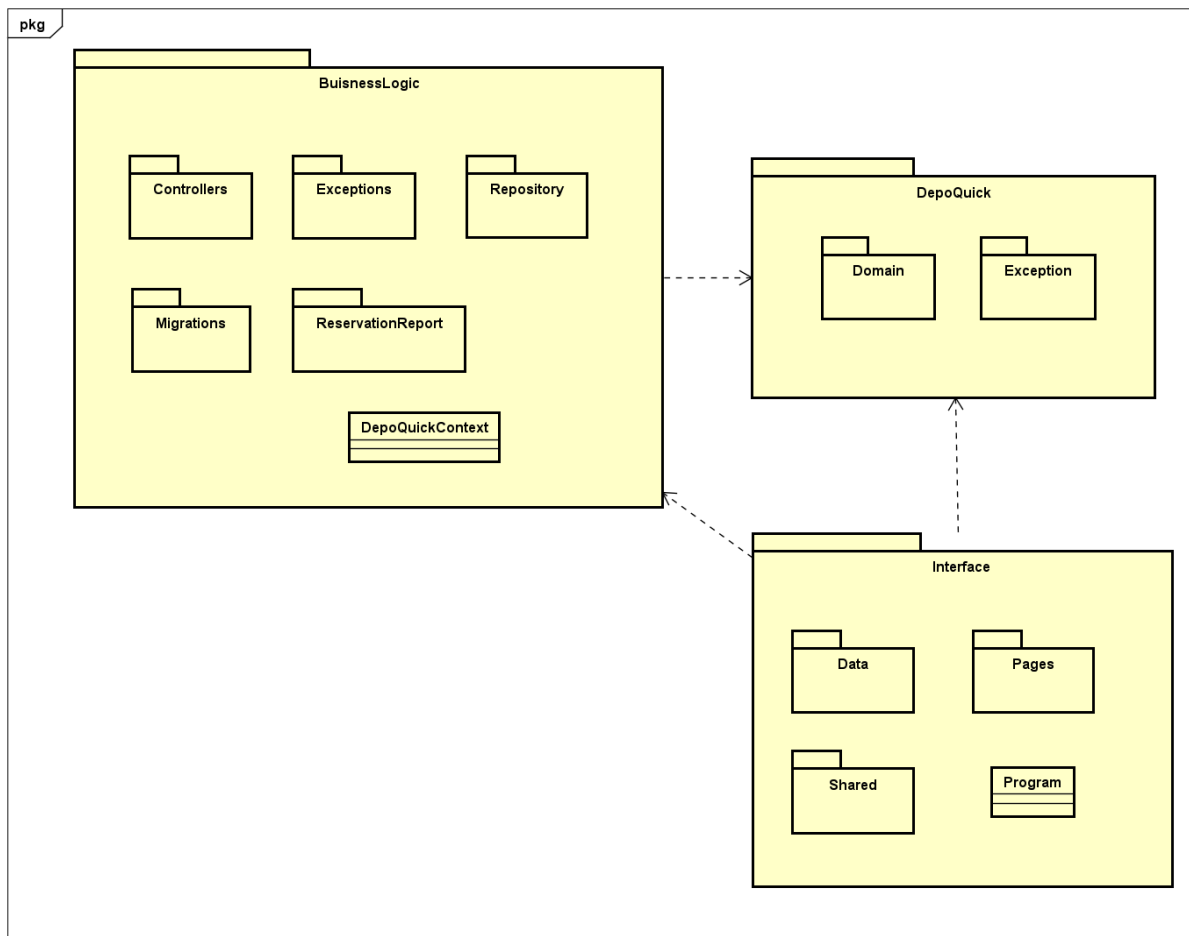
Cambios realizados

Se realizaron muchos cambios en cuanto al diseño y la lógica de negocio. Estos se encuentran descritos en el siguiente capítulo: Descripción y justificación de diseño, más específicamente en la sección Principales decisiones de diseño.

Descripción y justificación de diseño

Los diagramas son presentados como imágenes, pero debajo de ellas se encuentran los links a los archivos astah para poder visualizarlos con mayor facilidad.

Diagrama de paquetes

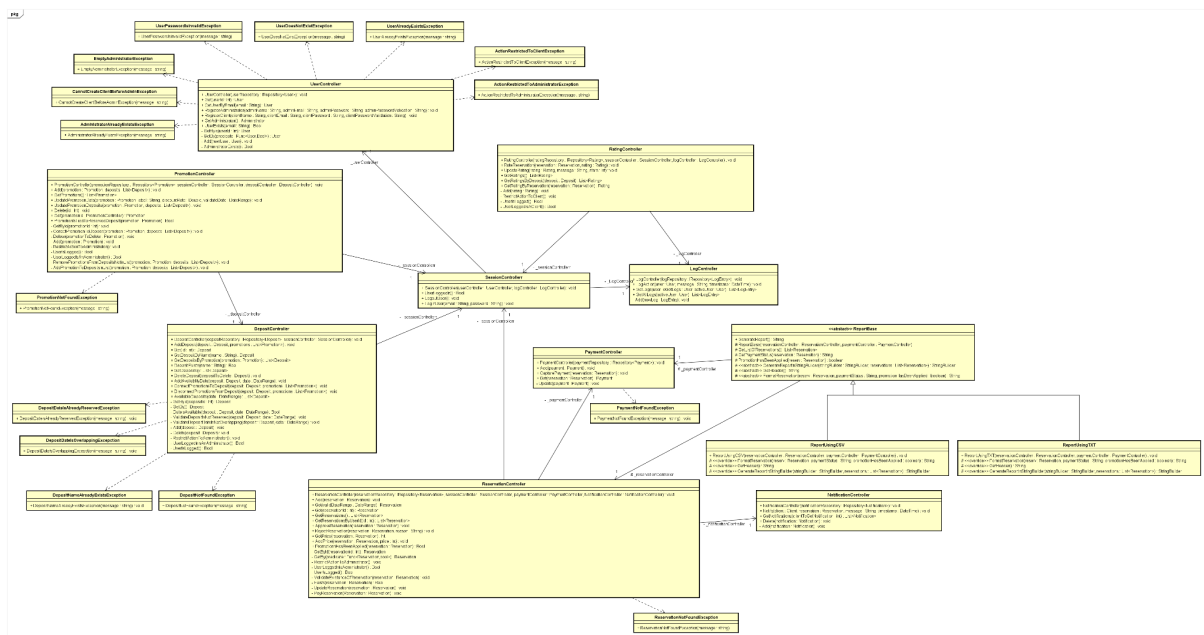


Diagramas de clases

Aqui el link

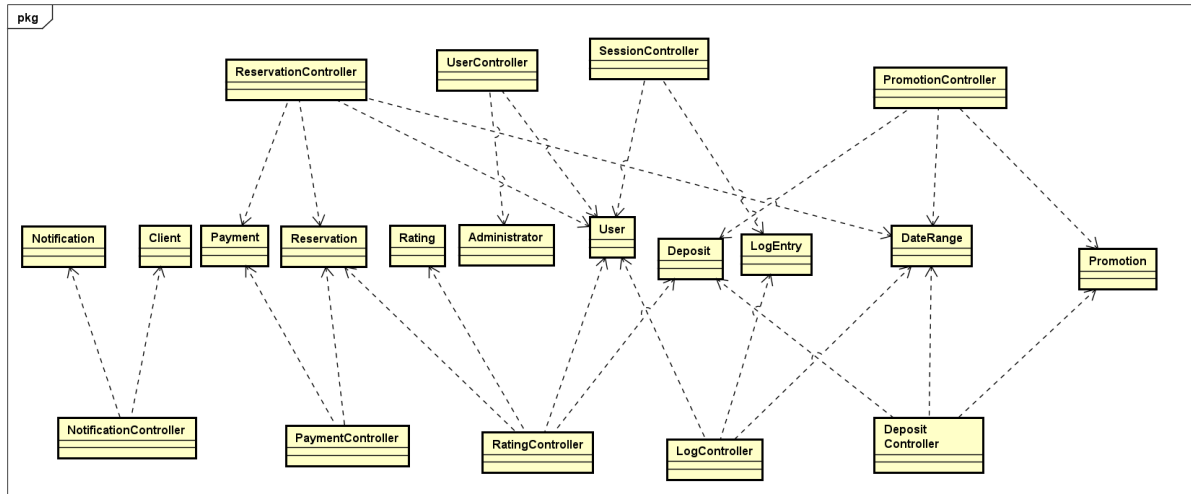


Controllers con exceptions (en BusinessLogic)



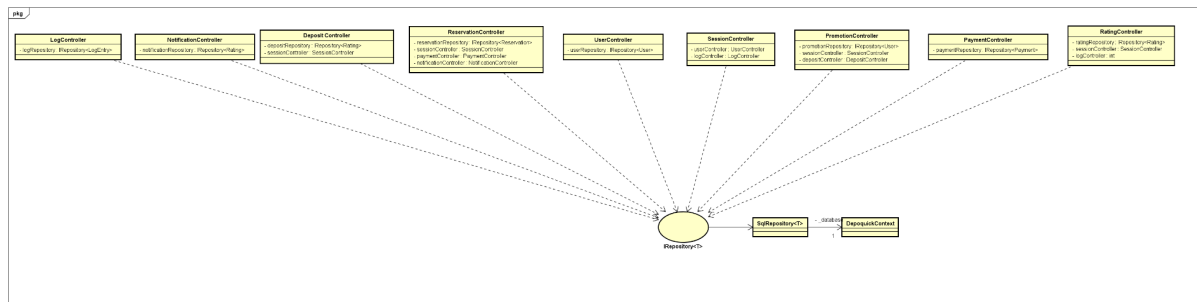
[Aqui el link](#)

Dependencias del dominio y los controllers



[Aqui el link](#)

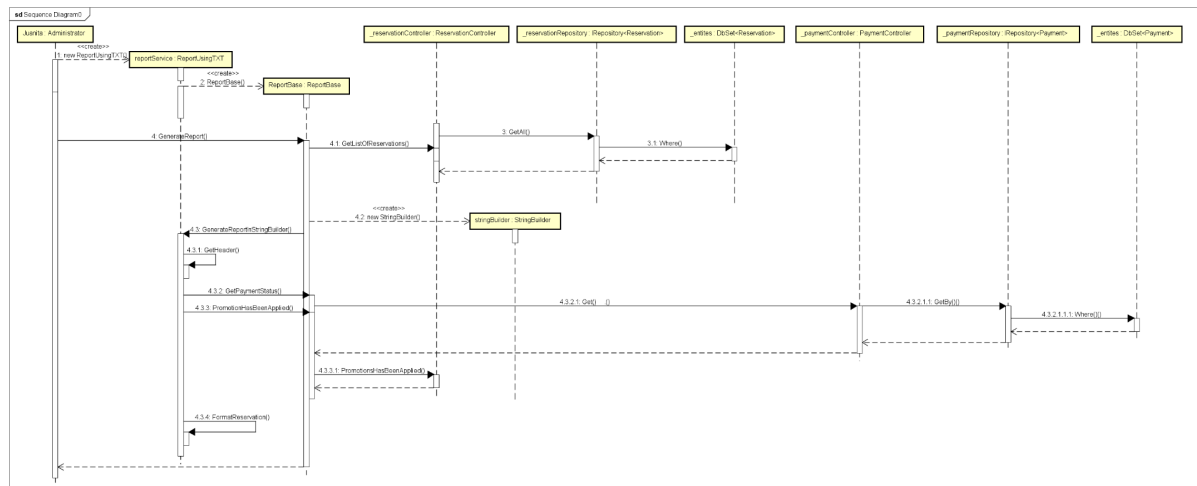
Relación entre los controllers y IRepository (intermediario de la base de datos)



[Aqui el link](#)

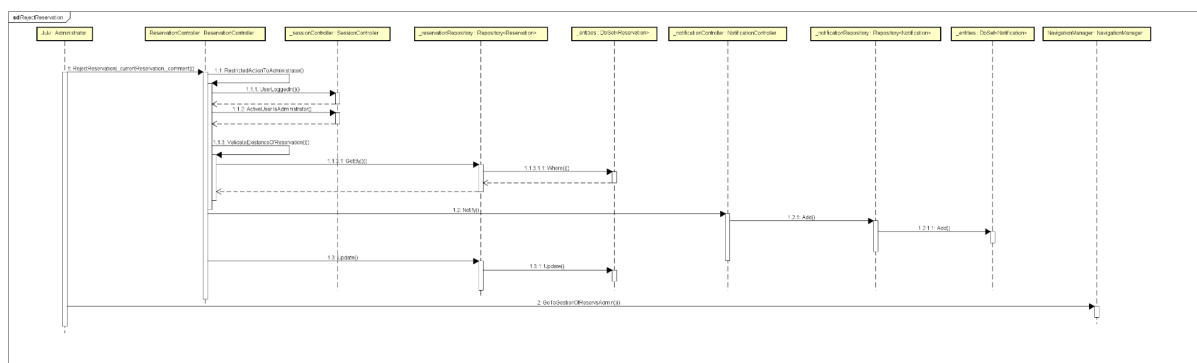
Diagramas de interacción

Exportar reporte de reservas.



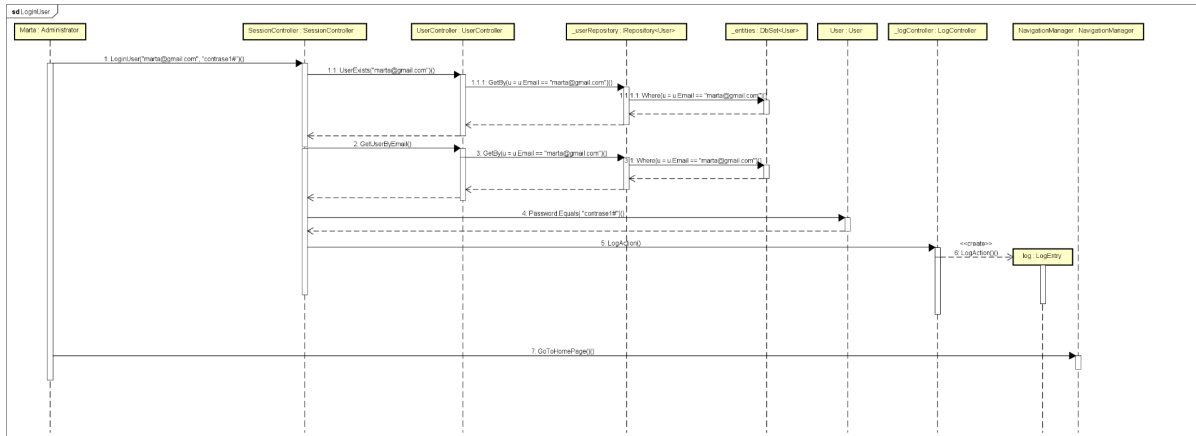
[Aqui el link](#)

Reject Reservation

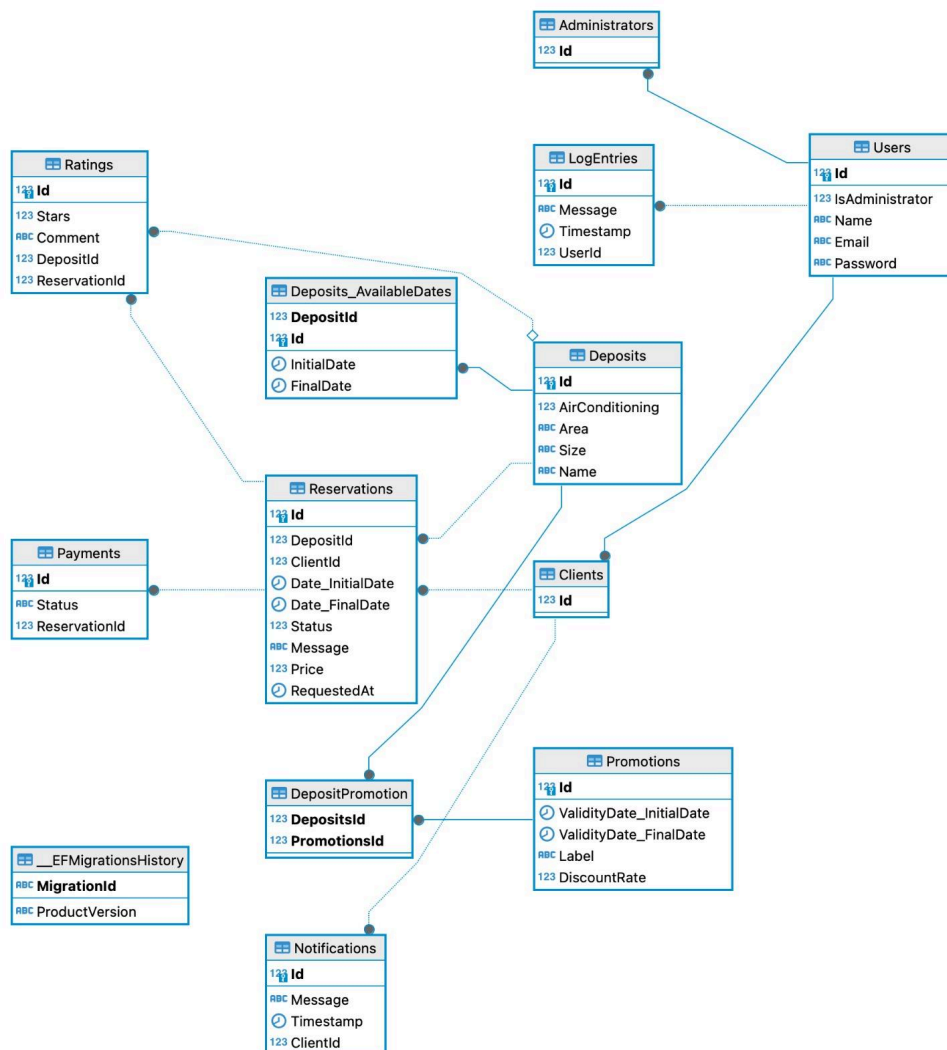


[Aqui el link](#)

Login



[Aqui el link](#)



Principales decisiones de diseño

Correcciones de la primera entrega

Para la segunda entrega tuvimos en cuenta las correcciones propuestas en la primera. Para ello, invertimos en reiterados métodos el orden de los `if` para que se evalúe primero la excepción y luego, si la misma no fue lanzada, se ejecute el método sin necesidad de tener un `else`.

Por otro lado, para seguir con los principios de clean code, se dividieron los métodos largos en varios métodos. Esto último a excepción de algunos add de la interfaz donde decidimos dejar una función con varios catch para evitar problemas en la base de datos. Notamos que al dividir los intentos de crear objetos en varios métodos no se tomaban bien las decisiones.

Asimismo, separamos la clase Controller en múltiples controladores para reducir el tamaño de Controller y aumentar la cohesión entre las clases.

Finalmente, otra de las decisiones tomadas fue la del uso de “Singleton” para evitar crear numerosas instancias de las distintas clases. Para ello, en la clase “Program.cs” creamos todos los controladores y los repositorios teniendo así una instancia única que es compartida en todo el proyecto. Los controladores son creados en el contenedor de servicios de ASP.NET Core. Esto se hace utilizando el método “AddSingleton” del contenedor de servicios. Gracias al uso de este patrón podemos “inyectar” las distintas clases utilizando “Dependency Injection”.

División de controladores

En la parte final del primer obligatorio nos dimos cuenta de que la decisión de crear un único controlador que gestione todas las clases del dominio no era adecuada. Esta estrategia no sigue las prácticas de Clean Code, ya que resulta en una clase con demasiadas dependencias y responsabilidades diversas, lo que disminuye su cohesión.

En esta segunda parte del obligatorio nos enfocamos en separar el controlador en varios controladores más pequeños, cada uno centrado en una clase del dominio. De este modo, aseguramos que cada controlador tuviera una alta cohesión y una única responsabilidad, manteniendo las clases cortas y enfocadas.

Interfaz IRepository<T>

Una de las mayores lecciones que nos dejó esta entrega es la importancia de separar la lógica de almacenamiento de datos con la lógica de los controladores. Los controladores no sólo no deberían llamar directamente al sistema de almacenamiento que se esté utilizando, sino que tampoco deberían conocer qué tipo de sistema es.

Decidimos crear IRepository<T>: una interfaz que define las firmas para los métodos básicos de manipulación de datos (CRUD) para una entidad/clase T genérica. Luego, todos los controladores se comunican con algún objeto de tipo IRepository<T> que corresponda (por ejemplo, DepositController conocerá IRepository<Deposit>), más desconocen cómo se están implementando estos métodos por detrás.

Para esta entrega debimos utilizar base de datos SQL, por lo que creamos la clase SqlRepository<T> que implementa la interfaz IRepository<T>. Esta clase implementa todos los métodos de la interfaz pero trabajando con Entity Framework y la base de datos. Para ello, se le pasa el DepoQuickContext por parámetro en el constructor.

Si en la primera entrega hubiésemos tenido IRepository<T>, y separado toda la lógica de manipulación de datos, pasar a usar una base de datos SQL habría sido considerablemente más sencillo, pues bastaría con crear la clase SqlRepository<T> e instanciar los repositorios en la interfaz y los tests.

En el caso de que en el día de mañana se quiera almacenar los datos en, por ejemplo, un bucket de S3, bastaría con implementar S3Repository<T> y cambiar las instancias de SqlRepository<T> de la interfaz y los tests por S3Repository<T>.

Separación de BusinessLogic y DepoQuick

El proyecto DepoQuick contiene las clases del dominio del sistema DepoQuick, que luego se mapean a nuestro modelo de datos. El proyecto BusinessLogic contiene todo lo relacionado a la lógica de negocio: los controladores, repositorios, sistemas de generación de reportes, contexto de base de datos, etc.

Esta es una forma de mejorar la cohesión entre los distintos módulos de nuestra aplicación. El dominio únicamente se encarga del núcleo de la aplicación: qué entidades tiene y cómo son esas entidades. Si necesito modificar las propiedades de una entidad, sé que debo ir a DepoQuick.

Por otro lado, BusinessLogic es responsable de cómo se comportan esas entidades: por ejemplo, cómo y dónde se almacenan los datos, qué debería tener un reporte de reservas, gestionar la aprobación de una reserva, etc. Si el día de mañana cambia la lógica de negocio (por ejemplo, no se puede pagar una reserva que aún no ha sido aprobada), basta con ir a BusinessLogic y modificar el método correspondiente en el controlador de reserva, y no es necesario modificar el dominio o modelo.





Manejo de fechas de disponibilidad

Por otro lado, para implementar las nuevas funcionalidades del requerimiento 1 decidimos que para crear un depósito, el administrador deba ingresar un rango válido de fechas. Luego de creado el mismo, el administrador podrá eliminarlo, ver las valoraciones o agregar nuevas disponibilidades. Esto último siempre y cuando las fechas sean válidas.

Otra decisión tomada fue que a la hora de reservar un depósito, el mismo usuario o dos usuarios distintos puede reservar para la misma fecha siempre y cuando no haya una reserva confirmada para la misma. La decisión de cuál de las reservas se aprueba queda a cargo del administrador, quien aprueba o rechaza la misma siempre y cuando el pago esté reservado. Sin embargo, si el administrador aprueba una de las reservas con fechas superpuestas, al intentar aprobar la otra se le notificará de que no puede aprobar porque el depósito ya no se encuentra disponible en ese rango de fechas.

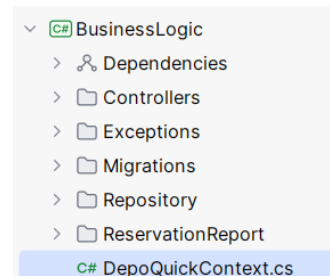
Análisis de los criterios seguidos para asignar las responsabilidades

Se ha realizado una separación del código en cuatro proyectos independientes:

- >  BusinessLogic
- >  DepoQuick
- >  DepoQuickTests
- >  Interface

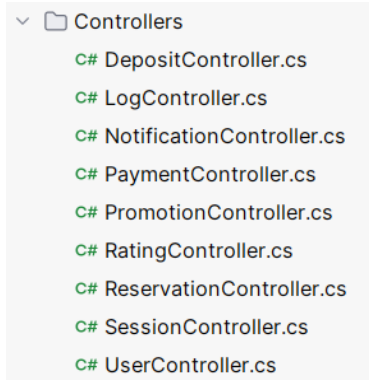
1. BusinessLogic:

Dentro de businessLogic decidimos poner todas las clases que se relacionan con la base de datos, esto para tener un código más organizado.

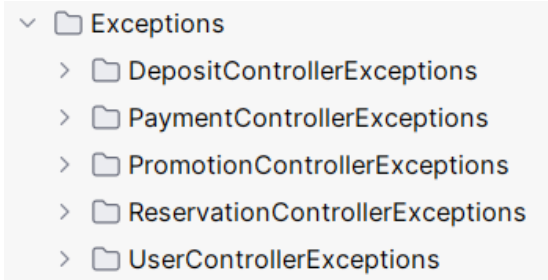


Como se observa, businessLogic se encuentra organizado por carpetas:

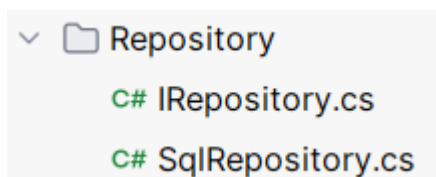
- 1) **Controllers:** Aquí se encuentran todos los controladores que se comunican con el dominio. Para la primera entrega, los controladores estaban en una sola clase, pero ahora están separados en clases individuales. Esto nos permite mantener una alta cohesión y un bajo acoplamiento. Además, seguimos el principio de responsabilidad única, el cual establece que una clase debe tener una única responsabilidad o motivo para cambiar.



- 2) **Exceptions:** Aquí se encuentran organizadas en carpetas, las excepciones específicas de los controladores.

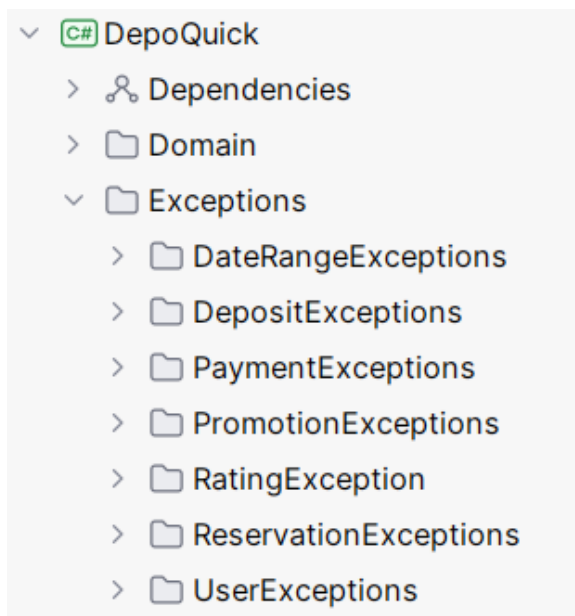


- 3) **Migrations:** Aquí se encuentran las migraciones realizadas por Entity Framework. Las migraciones son una forma de gestionar los cambios en el esquema de la base de datos a lo largo del tiempo, permitiendo actualizar y sincronizar el esquema con el modelo de datos de la aplicación de manera controlada y automática.
- 4) **Repository:** En esta carpeta se encuentra la interfaz del repositorio y el repositorio SQL. Un repositorio es un patrón de diseño que actúa como intermediario, en este caso, entre la aplicación y la base de datos, proporcionando una capa de abstracción para realizar operaciones de acceso y manipulación de datos. Para cualquier operación en la base de datos, se utiliza este intermediario.



- 5) **ReservationReport:** Aquí se implementó la gestión de los reportes de reservaciones, lo cual ya fue explicado previamente.
- 6) Por último, **DepoQuickContext** se utiliza para la configuración de la base de datos, básicamente, actúa como un puente entre el modelo de datos y la base de datos.
2. **DepoQuick:** Dentro de este proyecto, hay una carpeta denominada Domain, que contiene todas las clases relacionadas con el dominio del sistema, como User, Administrator, Client, Deposit, DateRange, Promotion, Rating, Reservation, Payment, Notification y LogEntry. Estas clases tienen un espacio de nombres (namespace) llamado DepoQuick.Domain. Optamos por separarlas de la lógica de negocio porque no interactúan directamente con la base de datos; más bien, son tipos de datos y clases que se utilizan en todo el proyecto, lo que nos ayudó a organizarnos mejor.

Por otro lado, la carpeta Exceptions alberga cada excepción en su propia carpeta, organizadas por clase. Esto se hizo para mejorar la legibilidad y mantener organizadas las excepciones según las clases a las que pertenecen.



3. **DepoQuickTests:** En esta carpeta se encuentran todas las clases destinadas a las pruebas unitarias de todo el proyecto, abarcando tanto el namespace DepoQuick como BusinessLogic.
4. **Interface:** Dentro de la carpeta Interface decidimos hacer un directorio por página. Utilizamos el “index” como página principal (o raíz) y luego dependiendo de la acción tomada nos movemos entre directorios. Al cargar la página por primera vez se le informa al usuario que no hay un administrador registrado y se le da como única opción el registrar administrador. Una vez registrado este último se le informa al usuario que el administrador fue correctamente registrado y se le da la opción de registrar cliente o acceder al sistema. Como tenemos algunas secciones que pueden ser vistas únicamente por el administrador y otras por el cliente, el menú lateral permanece oculto hasta que el usuario ingresa y es identificado en el backend como cliente o como administrador. Una vez verificadas las credenciales el administrador podrá ver las secciones inicio, registro, estadísticas, valoraciones, logs, gestión de reservas, depósitos, promociones y logout. Por otro lado, el cliente podrá visualizar las secciones inicio, reservar, gestión de reservas, depósitos, promociones y logout. Por último, en la esquina superior derecha decidimos agregar una

etiqueta que muestra constantemente si el usuario está logueado o no y de estarlo el nombre que ingresó.

Análisis de dependencias

Exportar reporte de reservas

Para el requerimiento de exportar el reporte de reservas, nos enfrentamos a algunos problemas. Sabíamos que deberíamos implementar la exportación de reservas en diferentes clases, , donde cada clase dependería del tipo de archivo que se quería descargar, esto porque separar en clases por tipo de archivo a exportar facilita la cohesión y la responsabilidad única, haciendo que cada clase tenga una única razón para cambiar. Además, simplifica el mantenimiento, pruebas y extensibilidad del código, ya que los cambios en un formato no afectan a los otros, alineándose con los principios SOLID, GASP y Clean Code.

El problema era que esto llevaría a mucha repetición de código, ya que las clases serían casi idénticas, con sólo algunos cambios que dependen de la estructura del archivo y no tanto de la búsqueda de información o métodos similares. Además, si en el futuro se quisiera agregar otro tipo de archivo, se tendría que repetir todo el trabajo, implementando el mismo código que las demás clases, con cambios sólo en la estructura.

Por lo tanto, dado que consideramos que este requerimiento es probable que cambie y que se agreguen nuevos tipos de archivos, decidimos implementarlo de una manera que sea consistente y que facilite la adición de nuevos tipos de archivos.

Para lograr esto, tuvimos en cuenta varias consideraciones para decidir qué estructura implementar.

Primeramente sabíamos que todos los tipos de exportación de reservas (ya sea PDF, TXT, etc.) tendrían que utilizar el método `GenerateReport()`, el cual se encarga de buscar la lista de reservas y crear un `StringBuilder` para manejar los strings de manera más eficiente y generar el reporte. Además, había métodos que todas las clases debían tener, como `GetListOfReservations()`, que devuelve la lista de reservas, o `GetPaymentStatus(Reservation reserv)`, que dado una reserva, devuelve su estado de pago.

Inicialmente, pensamos en implementar esto con una interfaz. Sin embargo, el problema con este enfoque era que queríamos tener métodos predefinidos que no necesitaran ser implementados en las clases derivadas.

Por ello, decidimos implementar esto usando polimorfismo, es decir, tener una clase base con todos los métodos básicos para generar el reporte. Estos métodos son `protected` y pueden ser sobrescritos por las clases derivadas. Además, la clase base incluye métodos abstractos (solo la firma de los métodos) que deben ser implementados por las clases derivadas. Estos métodos son aquellos que creemos dependen completamente del tipo de archivo, como `GetHEADER`, que agrega el encabezado del reporte y debe adaptarse a la estructura específica del archivo.

Todo esto se basó en el patrón de diseño (Template Method), que nos permitió definir la estructura general del algoritmo en la clase base y delegar la implementación de ciertos pasos a las subclasses. Este método es fundamentalmente lo que hemos aplicado aquí, permitiéndonos mantener una estructura coherente y extensible para la generación de reportes en distintos formatos.

Dependencias

Las clases encargadas de exportar archivos con la información de las reservaciones dependen de los controladores de Payment y Reservation. Además, también tienen dependencias directas con las clases Payment y Reservation.

Los controladores son fundamentales, ya que les permiten acceder a toda la información registrada en la base de datos. Por otro lado, la dependencia en la clase de Reservation y Payment les brinda acceso directo a los datos específicos de las reservaciones y de los pagos que deben ser exportados.

Análisis del nivel de cohesión y acoplamiento

Se buscó mantener un nivel de abstracción elevado para reducir el acoplamiento y aumentar la cohesión. Esto se logró al manejar los objetos a través de las funciones proporcionadas por sus respectivos controladores. Este enfoque evita que las clases dependan de un número excesivo de otras clases, promoviendo así una estructura más mantenible, y con bajo acoplamiento.

Además, al organizar las clases según el tipo de archivo a exportar, se logra una alta cohesión. Estas clases son pequeñas y se relacionan estrechamente entre sí, centrándose exclusivamente en el formato del archivo. No se preocupan por la búsqueda de datos o información, ya que esto se maneja a través de la clase base y de los controladores.

Este enfoque también se alinea con los principios SOLID, en particular con el Principio de Responsabilidad Única (SRP). Según este principio, una clase debería tener solo un motivo para cambiar. En el contexto de este sistema, el motivo para cambiar sería la modificación en la estructura del archivo.

Cada clase de exportación está dedicada a un tipo específico de archivo, lo que asegura que los cambios en la estructura del archivo solo afecten a una clase específica, manteniendo así la facilidad de mantenimiento del sistema.

Cobertura de pruebas unitarias

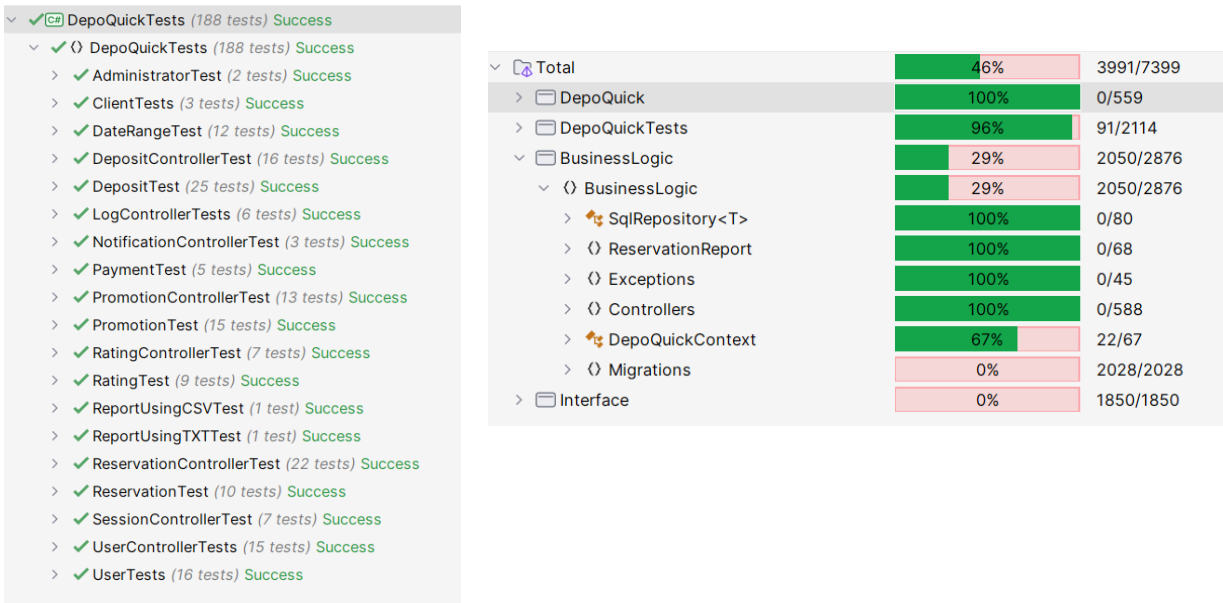
El proyecto, en esta parte, también se llevó a cabo siguiendo el enfoque de Desarrollo Dirigido por Pruebas (TDD). Este método garantiza que la cobertura de las pruebas unitarias sea del 100%. El proceso de TDD consiste en primero escribir pruebas para la funcionalidad que se desea implementar, asegurándose de que fallen inicialmente. Luego, se implementa el código necesario para que las pruebas pasen satisfactoriamente.

Este ciclo se repite iterativamente para cada nueva funcionalidad o modificación en el código.

Implementar TDD en todo nuestro proceso de desarrollo nos permitió lograr una cobertura del 100% en las pruebas, asegurando así que cada aspecto de nuestro código esté validado. Esta práctica nos brindó una sólida garantía de que las nuevas funcionalidades agregadas funcionarán según lo previsto. Además, nos otorgó la confianza necesaria para realizar modificaciones en el código existente o agregar nuevas funcionalidades, sabiendo que contábamos con pruebas exhaustivas respaldando cada

cambio realizado. Esto no solo nos ayudó a detectar errores tempranamente, sino que también nos proporcionó un proceso de desarrollo más seguro y eficiente.

A continuación se evidencia la cobertura total de pruebas unitarias para los proyectos DepoQuick y BusinessLogic.



Anexos

Anexo 1 - Tabla de responsabilidades

Namespace	Clase	Responsabilidad
DepoQuick.Domain	User	Se encarga de la gestión de la información del usuario y de la validación de los datos del usuario.
DepoQuick.Domain	Administrator	Comparte las responsabilidades de User y se encarga de la aprobación o rechazo de las reservas.
DepoQuick.Domain	Client	Comparte las responsabilidades de User y se encarga de la gestión de las reservas del cliente.
DepoQuick.Domain	DateRange	Se encarga de la gestión y validación de un rango de fechas.
DepoQuick.Domain	Deposit	Se encarga de la gestión de la información del depósito, la validación de los datos del depósito, y la gestión de las promociones y reservas del depósito.
DepoQuick.Domain	Promotion	Se encarga de la gestión de la información de la promoción, la validación de los datos de la promoción, y la gestión de los depósitos asociados a la promoción.
DepoQuick.Domain	Rating	Se encarga de la gestión de la información de la calificación y la validación de los datos de la calificación.
DepoQuick.Domain	Reservation	Se encarga de la gestión de la información de la reserva y la validación de los datos de la reserva.
DepoQuick.Domain	LogEntry	Se encarga de la gestión de logs.
DepoQuick.Domain	Notification	Se encarga de la gestión y creación de las notificaciones.
DepoQuick.Domain	Payment	Se encarga de la creación y gestión de pagos, como también el cambio del estado del pago.

DepoQuick.Exceptions.DateRangeExceptions	EmptyDateRangeException	Indicar un error específico cuando se intenta crear un par de fechas vacío.
DepoQuick.Exceptions.DateRangeExceptions	InvalidDateRangeException	Indicar un error específico cuando se intenta crear un par de fechas inválido.
DepoQuick.Exceptions.DepositExceptions	DepositWithInvalidAreaException	Indicar un error específico cuando se intenta crear un depósito con área inválida.
DepoQuick.Exceptions.DepositExceptions	DepositWithInvalidSizeException	Indicar un error específico cuando se intenta crear un depósito con tamaño inválido.
DepoQuick.Exceptions.DepositExceptions	DepositNameIsNotValidException	Indicar un error específico cuando se intenta crear un depósito con nombre inválido.
DepoQuick.Exceptions.PaymentExceptions	CannotCapturePaymentIfDoesNotHaveAnAssociatedReservation	Indicar un error específico cuando se intenta capturar un pago que no tiene una reservación asociada.
DepoQuick.Exceptions.PromotionExceptions	InvalidPercentageForPromotionException	Indicar un error específico cuando se intenta crear una promoción con porcentaje de descuento inválido.
DepoQuick.Exceptions.PromotionExceptions	PromotionLabelHasMoreThan20CharactersException	Indicar un error específico cuando se intenta crear una promoción con una etiqueta con más de 20 caracteres.
DepoQuick.Exceptions.PromotionExceptions	PromotionWithEmptyLabelException	Indicar un error específico cuando se intenta crear una promoción con una etiqueta vacía.
DepoQuick.Exceptions.RatingException	InvalidCommentForRatingException	Indicar un error específico cuando se intenta crear una reseña con un comentario inválido.
DepoQuick.Exceptions.RatingException	InvalidStarsForRatingException	Indicar un error específico cuando se intenta crear una reseña con estrellas inválidas.
DepoQuick.Exceptions.ReservationExceptions	ReservationMessageHasMoreThan300CharactersException	Indicar un error específico cuando se intenta agregar a una reserva un comentario de más de 300 caracteres.
DepoQuick.Exceptions.ReservationExceptions	ReservationWithEmptyMessageException	Indicar un error específico cuando se intenta agregar a una reserva un comentario vacío.
DepoQuick.Exceptions.UserExceptions	EmptyActionLogException	El sistema debe impedir que se registren mensajes de log vacíos y solo permitir eventos válidos como inicio de sesión, cierre de

		sesión o valoraciones.
DepoQuick.Exceptions.UserExceptions	EmptyUserEmailException	Indicar un error específico cuando se intenta crear un usuario con un email vacío.
DepoQuick.Exceptions.UserExceptions	EmptyUserNameException	Indicar un error específico cuando se intenta crear un usuario con un nombre vacío.
DepoQuick.Exceptions.UserExceptions	EmptyUserPasswordException	Indicar un error específico cuando se intenta crear un usuario con una contraseña vacía.
DepoQuick.Exceptions.UserExceptions	InvalidUserEmailException	Indicar un error específico cuando se intenta crear un usuario con un email inválido.
DepoQuick.Exceptions.UserExceptions	InvalidUserNameException	Indicar un error específico cuando se intenta crear un usuario con un nombre inválido.
DepoQuick.Exceptions.UserExceptions	InvalidUserPasswordException	Indicar un error específico cuando se intenta crear un usuario con una contraseña inválida.
DepoQuick.Exceptions.UserExceptions	PasswordTooShortException	Indicar un error específico cuando se intenta crear un usuario con una contraseña que no cumple el mínimo de caracteres.
DepoQuick.Exceptions.UserExceptions	UserNameTooLongException	Indicar un error específico cuando se intenta crear un usuario con una contraseña que supera el máximo de caracteres.
DepoQuick.Exceptions.UserExceptions	UserPasswordsDoNotMatchException	Indicar un error específico cuando se intenta crear un usuario con una validación de contraseña diferente a la contraseña.
BusinessLogic	DepoquickContext	Es un intermediario entre la aplicación y la base de datos.
BusinessLogic.ReservationReport	ReportBase	Es una clase abstracta base que se utiliza para crear los reportes de las reservaciones.
BusinessLogic.ReservationReport	ReportUsingCSV	Hereda de ReportBase, es la que implementa la creación de los reportes de tipo CSV.
BusinessLogic.ReservationReport	ReportUsingTXT	Hereda de ReportBase, es la que implementa la creación de los reportes de tipo TXT.
BusinessLogic.Repository	IRepository	Es la interfaz con las firmas de los métodos CRUD requeridos para almacenar y gestionar los datos en

		algún sistema de almacenamiento.
BusinessLogic.Repository	SqlRepository	Implementa IRepository, es la que tiene los métodos para manejar la base de datos SQL.
BusinessLogic.Migrations	-	Se encuentran todas las migraciones que se realizaron.
BusinessLogic.Controllers	DepositController	Es el intermediario entre el sistema de almacenamiento de datos y la interfaz. Realiza operaciones relacionadas con los depósitos.
BusinessLogic.Controllers	LogController	Es el intermediario entre el sistema de almacenamiento de datos. Realiza operaciones relacionadas con los Logs.
BusinessLogic.Controllers	NotificationController	Es el intermediario entre el sistema de almacenamiento de datos y la interfaz. Realiza operaciones relacionadas con las notificaciones.
BusinessLogic.Controllers	PaymentController	Es el intermediario entre el sistema de almacenamiento de datos y la interfaz. Realiza operaciones relacionadas con los pagos.
BusinessLogic.Controllers	PromotionController	Es el intermediario entre el sistema de almacenamiento de datos y la interfaz. Realiza operaciones relacionadas con las promociones.
BusinessLogic.Controllers	RatingController	Es el intermediario entre el sistema de almacenamiento de datos y la interfaz. Realiza operaciones relacionadas con las valoraciones.
BusinessLogic.Controllers	ReservationController	Es el intermediario entre el sistema de almacenamiento de datos y la interfaz. Realiza operaciones relacionadas con las reservaciones.
BusinessLogic.Controllers	SessionController	Es el intermediario entre el sistema de almacenamiento de datos y la interfaz. Realiza operaciones relacionadas con el inicio y cierre de sesión.
BusinessLogic.Controllers	UserController	Es el intermediario entre el sistema de almacenamiento de datos y la interfaz. Realiza

		operaciones relacionadas con los usuarios.
BusinessLogic.Exceptions.UserControllerExceptions	ActionRestrictedToAdministratorException	Indicar un error específico cuando un cliente intenta realizar una acción restringida a un administrador.
BusinessLogic.Exceptions.UserControllerExceptions	ActionRestrictedToClientException	Indicar un error específico cuando un administrador intenta realizar una acción restringida a un cliente.
BusinessLogic.Exceptions.UserControllerExceptions	AdministratorAlreadyExistsException	Indicar un error específico cuando se intenta registrar un administrador que ya existe.
BusinessLogic.Exceptions.UserControllerExceptions	CannotCreateClientBeforeAdminException	Indicar un error específico cuando se intenta registrar un cliente sin que exista un administrador.
BusinessLogic.Exceptions.UserControllerExceptions	EmptyAdministratorException	Indicar un error específico cuando no existe administrador registrado.
BusinessLogic.Exceptions.UserControllerExceptions	UserAlreadyExistsException	Indicar un error específico cuando se intenta registrar un cliente que ya existe.
BusinessLogic.Exceptions.UserControllerExceptions	UserDoesNotExistException	Indicar un error específico cuando se intenta loguear un usuario que no existe.
BusinessLogic.Exceptions.UserControllerExceptions	UserPasswordIsInvalidException	Indicar un error específico cuando la contraseña es inválida.
BusinessLogic.Exceptions.DepositControllerExceptions	DepositNotFoundException	Indicar un error específico cuando se intenta buscar o acceder a un depósito que no existe.
BusinessLogic.Exceptions.DepositControllerExceptions	DepositNameAlreadyExistsException	Indicar un error específico cuando se intenta agregar un depósito con un nombre que ya existe.
BusinessLogic.Exceptions.PromotionControllerExceptions	PromotionNotFoundException	Indicar un error específico cuando se intenta buscar o acceder a una promoción que no existe.
BusinessLogic.Exceptions.ReservationControllerExceptions	ReservationNotFoundException	Indicar un error específico cuando se intenta buscar o acceder a una reserva que no existe.
BusinessLogic.Exceptions.PaymentControllerExceptions	PaymentNotFoundException	Indicar un error específico cuando se intenta buscar o acceder a un pago que no existe.
BusinessLogic.Exceptions.DepositControllerExceptions	DepositDateIsOverlappingException	Indicar un error específico cuando dos reservaciones se superponen.
BusinessLogic.Exceptions.DepositControllerExceptions	DepositDateIsAlreadyReservedException	Indicar un error específico cuando se intenta reservar un depósito que

		ya está reservado para esa fecha.
--	--	-----------------------------------

Anexo 2 - Guía de ejecución

Para correr la aplicación de forma local, ejecutar los siguientes pasos:

1. Clonar el repositorio de GitHub. El link está en la portada.
2. Dirigirse a la carpeta *Scripts*.
3. Ejecutar el script *container_setup.sh* para descargar la imagen docker del servidor SQL y correr el contenedor.
4. Esperar aproximadamente un minuto para que el contenedor se levante correctamente. Para ver cómo va, abrir Docker Desktop y ver el contenedor *sqlserver*.
5. Dentro del contenedor, ejecutar el script SQL *CreateSchema.sql* para crear la base de datos *depoquick* y sus tablas. Si lo prefiere, en lugar de entrar al contenedor puede conectarse a la base de datos mediante DBeaver y ejecutar el script desde allí.
6. Ir a la carpeta *publish* y correr el ejecutable *Interface*. Esto levantará la aplicación de forma local y le indicará qué URL (localhost y puerto) utilizar.
7. Si se desea, se puede correr el script SQL *InsertData.sql* para cargar múltiples datos de prueba a la base de datos. Estos datos contienen varios usuarios, depósitos, reservas, promociones, notificaciones y logs. Ver el script para ver los usuarios y contraseñas para loguearse luego.