Obligatorio 1 de Diseño de Aplicaciones II

Se desea implementar una solución que permita administrar y publicar información sobre los partidos para diferentes deportes.

La aplicación cuenta con diferentes perfiles (**administradores** y **seguidores**) que utilizan la misma. Todos los usuarios del sistema son seguidores y además puede ser o no administrador. Estos dos son y serán los únicos roles necesarios en la aplicación, por lo que no es necesario una solución compleja de manejo de roles.

A continuación se describen las funcionalidades según el perfil del usuario utilizando la aplicación.

1. Usuario administrador

Permite el mantenimiento (alta, baja, modificación) de los siguientes datos:

- **Usuario** del sistema registrando la siguiente información: nombre, apellido, nombre de usuario, contraseña, dirección de correo electrónico (se debe validar el formato), y si es administrador.
- **Deporte**: nombre del deporte, y la lista de **equipos** participantes en dicho deporte.
- Para cada **equipo** se conoce: nombre y foto.
- **Encuentro**: fecha y hora, deporte, equipos participantes (dos), y lista de comentarios (textos).

El ***fixture*** de encuentros para cada deporte pueden ser ingresados manualmente o mediante algún algoritmo automático de armado de *fixtures* (por ejemplo partidos de ida y vuelta, grupos, etc.). Lo que implica que debe ser sencillo agregar un nuevo algoritmo para ser utilizado en el sistema. La entrega debe contener al menos dos algoritmos implementados por el grupo.

Reportes:

A. Listar todos los encuentros para un deporte.
B. Listar todos los encuentros de un equipo en particular.

2. Usuario seguidor de equipos

Al ingresar al sistema un usuario con perfil de *seguidor*, se le presenta la siguiente información:

- Sección en la página con la **lista de todos los comentarios de los encuentros de todos los equipos que el usuario está siguiendo** (favoritos). La misma debe estar ordenada por encuentro.
- **Calendario** con todos los eventos agrupados por deporte, un ejemplo de presentación:

	<															AGOSTO 2018															>
DEPORTE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Basquetbol																															
Fútbol			1	4	5			2			5	5						6	4			4		3	2	5					2
Tenis																															
Rugby					4							4							2							1					
Vóleibol				3	3	3			2	1																					

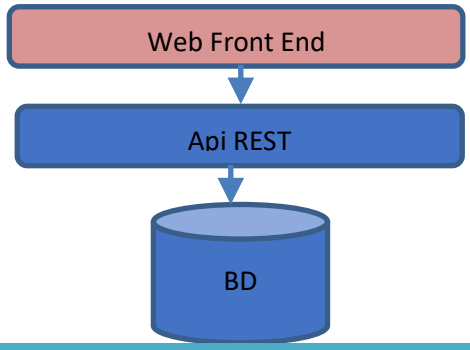
Al pasar el *mouse* sobre una celda (intersección fecha-deporte), se debe mostrar la lista de eventos (la hora y participantes) en un *tooltip*.

Un usuario puede también realizar **comentarios** asociados a un **encuentro** en particular. Selecciona un encuentro y luego ingresa un comentario (texto).

El usuario puede administrar la siguiente información dentro de su perfil:

- El usuario puede visualizar la lista de todos los equipos con sus datos, pudiendo filtrar y ordenar (en forma creciente o decreciente) por nombre.
- Se permite seleccionar uno o más equipos de la lista, de forma que el usuario puede seleccionarlos para seguirlos (**favoritos**) y que se muestren sus comentarios en la página principal.

Para resolver el problema se definió la siguiente **arquitectura de alto nivel**.



Artefacto	Descripción
Base de Datos	Base de datos relacional (SQL Server) en donde se almacenan los datos de la aplicación.
Api REST + Back-end	Toda interacción con el repositorio de datos se realiza mediante un API REST, la que ofrece operaciones para resolver todo lo necesario y el back-end donde se implementa la lógica de negocio.
Web Front End	Aplicación que permite a los usuarios registrarse e interactuar con la aplicación.

Alcance de la primera entrega

Considerando la arquitectura mencionada en la sección anterior, para esta primera **entrega se debe implementar una API REST** que ofrezca todas las operaciones que el alumno considere necesarias para soportar la aplicación descrita en este documento.

Dicho de otra manera, se debe implementar **todo el back-end** de la aplicación (lógica de negocio), pero **no el front-end** de la misma (interfaz de usuario). Se espera que, mediante la descripción de la aplicación, y complementando con consultas a Aulas cuando considere necesario, el alumno sea capaz de identificar las operaciones que el API debe ofrecer.

Dado que la aplicación no tendrá interfaz de usuario, la forma de interactuar con la misma será mediante un cliente HTTP. Se espera que se utilice **Postman** (https://www.getpostman.com/) como cliente, ya que se debe entregar una exportación de una colección de Postman con los end-points definidos para poder probar. Deben tener todos las llamadas a la API preparadas y utilizar la parametrización de Postman en cuanto a la URL a utilizar, token en el header, etc. para no tener que escribir todas las llamadas nuevamente en la demostración. **En caso de**

no entregar dicha colección, los docentes pueden optar por la no corrección de la funcionalidad. Ver la rúbrica de evaluación en las secciones más abajo.

Implementación
Se debe entregar una implementación del API REST necesaria para soportar la aplicación que se describe. La entrega debe contener una solución de Visual Studio que agrupe todos los proyectos implementados.
La solución debe incluir el código de las pruebas automáticas. Se requiere escribir los casos de prueba automatizados con MSTests, documentando y justificando las pruebas realizadas.

Se espera que la aplicación se entregue con una base de datos con datos de prueba, de manera de poder comenzar las pruebas sin tener que definir una cantidad de datos iniciales. Dichos datos de prueba deben estar adecuadamente especificados en la documentación entregada.

Tecnologías y herramientas de desarrollo

- Microsoft Visual Studio Code (lenguaje C#)
- Microsoft SQL Server Express 2014
- Postman 6.2.4
- NET Core SDK 2.1.4 / ASP.NET Core 2
- Entity Framework Core
- Astah o cualquier otra herramienta UML 2

Instalación
El costo de instalación de la aplicación debe de ser mínimo y documentado adecuadamente.

NOTA: La totalidad y detalle de los requisitos serán relevados a partir de consultas en el foro correspondiente en aulas. Para evitar complejidades innecesarias se realizaron simplificaciones al dominio del problema real.

Independencia de librerías
Se debe diseñar la solución que al modificar el código fuente minimice el impacto del cambio en los componentes físicos de la solución. Debe documentar explícitamente como su solución cumple con este punto. Cada paquete lógico debe ser implementado en un *assembly* independiente, documentando cuáles de los elementos internos al paquete son públicos y cuáles privados, o sea cuáles son las interfaces de cada *assembly*.

Persistencia de los datos
La empresa requiere que todos los datos del sistema sean persistidos en una base de datos. De esta manera, la siguiente vez que se ejecute la aplicación se comenzará con dichos datos cargados con el último estado guardado antes de cerrar la aplicación.

Persistencia en base de datos

Toda la información contenida en el sistema debe ser persistida en una base de datos. El diseño debe contemplar el modelado de una solución de persistencia adecuada para el problema utilizando Entity Framework (*Code First*).

Se espera que como parte de la entrega se incluya dos respaldos de la base de datos: uno vacío y otro con datos de prueba. Se debe entregar el archivo *.bak* y también el script *.sql* para ambas base de datos.

Es condición necesaria para obtener el puntaje mínimo del obligatorio que al menos una entidad del sistema pueda ser persistida.

Mantenibilidad

La propia empresa eventualmente hará cambios sobre el sistema, por lo que se requiere un alto grado de mantenibilidad, flexibilidad, calidad, claridad del código y documentación adecuada.

Por lo que el desarrollo de todo el obligatorio debe cumplir:

- Estar en un repositorio **Git**.
- Haber sido escrito utilizando **TDD** (desarrollo guiado por pruebas) lo que involucra otras dos prácticas: escribir las pruebas primero (Test First Development) y Refactoring. De esta forma se utilizan las pruebas unitarias para dirigir el diseño.
  Es necesario utilizar **TDD únicamente** para el *back-end* y la *API REST*, no para el desarrollo del *front-end*.
  Se debe utilizar un framework de Mocking (como Moq, https://www.nuget.org/packages/moq/) para poder realizar pruebas unitarias sobre la lógica de negocio. En caso de necesitar hacer un test double del acceso a datos, podrán hacerlo utilizando el mismo framework de Mocking anterior o de lo contrario con el paquete EF Core InMemory (https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.InMemory).
- Cumplir los lineamientos de **Clean Code** (capítulos 1 al 10, y el 12), utilizando las técnicas y metodologías ágiles presentadas para crear código limpio.

Control de versiones

La gestión del código del obligatorio debe realizarse utilizando UN ÚNICO repositorio Git de **Github**, apoyándose en el flujo de trabajo recomendado por **GitFlow** (nvie.com/posts/a-successful-git-branching-model). Dicho repositorio debe pertenecer a la organización de GitHub "ORT-DA2" (www.github.com/ORT-DA2), en la cual deben estar todos los miembros del equipo. **Al realizar la entrega se debe realizar un *release* en el repositorio y la rama *master* no debe ser afectada luego del hito que corresponde a la entrega del obligatorio.**

Documentación

La documentación entregada debe ser en **un solo** documento digital en formato PDF, que contenga la siguiente información ordenada e indexada:

- La documentación no debe superar las 25 páginas pudiéndose complementar mediante la utilización de anexos, por ejemplo, utilizando anexos para el informe de clean code, informe de pruebas y mayor detalle o diagramas para casos puntuales.

- Descripción general del trabajo (1/2 carilla): si alguna funcionalidad no fue implementada o si hay algún error conocido (*bug*) deben ser descritos aquí.
- Descripción del diseño propuesto, incluyendo los diagramas de paquetes, clases (al menos uno por paquete), interacción para especificar los principales comportamientos del sistema, de componentes y de entrega.
- Justificación de las decisiones de diseño tomadas, explicando los mecanismos generales, aplicación de principios, patrones, mecanismo de persistencia, etc.
- Modelo de tablas de la estructura de la base de datos.
- Justificación de Clean Code. Explicación de por qué entiende que su código se puede considerar limpio, y en caso de existir, aclaración de oportunidades de mejora o aspectos en los que cree que no cumple con el estándar.
- Resultado de la ejecución de las pruebas. Evidencia del código de pruebas automáticas (unitarias y de integración), reporte de la herramienta de cobertura y análisis del resultado. Como parte de la evaluación se va a revisar el nivel de cobertura de los *tests* sobre el código entregado, por lo que se debe entregar un reporte y un análisis de la cobertura de las pruebas.
- El documento debe cumplir con los siguientes elementos del Documento 302 de la facultad (http://www.ort.edu.uy/fi/pdf/documento302facultaddeingenieria.pdf)
    a. Capítulo 3, secciones 3.1 (sin la leyenda), 3.2, 3.5, 3.7, 3.8 y 3.9
    b. Capítulos 4 (salvo 4.1) y 5

Las condiciones de entrega serán evaluadas como si se le estuviese entregando a un cliente real: prolijidad, claridad, profesionalismo, etc.

La entrega debe ser la documentación en formato PDF (incluyendo modelado UML) y acceso a los docentes al repositorio Git utilizado por el grupo. En el repositorio Git se debe incluir:

- Una carpeta con la aplicación compilada en *release* para realizar la instalación de la misma.
- Código fuente de la aplicación, incluyendo todo lo necesario que permita compilar y ejecutar la aplicación.
- Una carpeta Documentación en la que se incluya el documento en formato PDF (incluyendo modelado UML, tener especial cuidado que los diagramas queden legibles en el documento).
- Base de datos: entregar una base de datos vacía y otra con datos de prueba.

Evaluación (20 puntos)

	Evaluación de	**Puntos**
Demo al cliente	Cada equipo deberá realizar una demostración al cliente de su trabajo, **en las computadoras de los laboratorios de ORT**. Dentro de los aspectos que un cliente espera se encuentran: ● **Ver la demo cuando él esté listo y no tener que esperar a que el equipo se apronte.** ● **Los datos y la funcionalidad a mostrar debe estar preparada. Y a la hora de mostrar la interacción con la API, todas las llamadas deben venir preparadas y**	2

	parametrizadas, siendo realmente simple probar el correcto funcionamiento. El equipo no debe escribir las llamadas a mano o tener que copiar y pegarlas. Cada una de las interacciones debe venir preparada de antemano. De no cumplir con esto, el cliente puede decidir no corregir la funcionalidad. ● Probar la solución y que la misma funcione sin problemas o con problemas mínimos, y de buena calidad de interfaz de usuario. ● Conocer las capacidades de cada uno de los integrantes del equipo, pudiendo preguntar a cualquier integrantes sobre la solución, su diseño, el código y sobre cómo fue construida, y así apreciar que fue un trabajo en equipo. Todos los integrantes deben conocer toda la solución. ● Verificar el aporte individual al trabajo por parte de cada uno de los integrantes del equipo y en función de los resultados, se podrán otorgar distintas notas a los integrantes del grupo. Se espera que cada uno de los integrantes haya participado en la codificación de parte significativa del obligatorio. Esta demostración al cliente hará las veces de **defensa del trabajo**. NOTA: El incorrecto funcionamiento de la instalación puede significar la no corrección de la funcionalidad. En el caso de defensa en el laboratorio, durante la defensa cada grupo contará con 15 minutos para la instalación de la aplicación. Luego de transcurridos los mismos se restan puntos al trabajo.	
Funcionalidad	La asignación de los puntos de este ítem se basará en los siguientes criterios de corrección: • Se implementaron sin errores todos los requerimientos funcionales propuestos. • Se implementaron todos los requerimientos funcionales propuestos, pero se detectan errores menores que no afectan el uso normal del sistema. • Se implementaron los principales requerimientos funcionales, aunque no todos, pero se detectan errores menores que no afectan el uso normal del sistema. • Se implementaron los principales requerimientos funcionales, aunque no todos, pero se detectan errores que afectan el uso normal del sistema. • Los requerimientos funcionales implementados son básicos y/o se detectan errores que afectan el uso normal del sistema.	6
Diseño y documentación	La documentación cumple con lo detallado en la sección documentación. Se considera un diseño y documentación como aceptable si: • Los diagramas presentan el uso adecuado de la notación UML. • La estructura de la solución representa la descomposición lógica (módulos) y de proyectos de la aplicación. • Las vistas de módulos, de componentes, modelo de datos y los comportamientos documentados sirven como guía para la comprensión del código implementado. • La taxonomía de paquetes y clases en los diagramas respeta las convenciones de nombres de C# utilizada en la implementación. • Se justifica y explica el diseño en base al uso de principios y patrones de diseño. Los mismos se implementan correctamente a partir de su objetivo y teniendo en cuenta sus ventajas y desventajas para favorecer o inhibir la calidad de la solución. • El diseño de la API REST se basa en buenas prácticas de la industria (por ejemplo https://docs.microsoft.com/en-us/azure/architecture/best-	6

	practices/api-design y Web API Design: The Missing Link by apigee). • El informe de Clean Code justifica por qué su código se puede considerar limpio. • El informe de las pruebas sirve como evidencia de la correcta aplicación de las mismas. • Se describen claramente los errores conocidos. • La documentación se encuentra bien organizada, es fácil de leer y su formato corresponde con los ítems indicados del documento 302.	
Implementación	Correcta aplicación de desarrollo guiado por las pruebas (TDD) y técnicas de refactoreo de código. Utilización de buenas prácticas de estilo y codificación y su impacto en la mantenibilidad (Clean Code). Correcto uso de las tecnologías. Claridad del código respetando las guías de estilo de C#. Concordancia con el diseño documentado. Correcto manejo de excepciones. Implementación de acceso a base de datos.	6

Información importante

Lectura de obligatorio: 03-09-2018 Plazo máximo de entrega: 10-10-2018

Defensa: A definir por el docente Puntaje mínimo / máximo: 10 / 20 puntos

Los grupos de obligatorio se forman como máximo por 2 estudiantes.

Todas las entregas se realizan mediante las reglas de la entrega electrónica de obligatorios.