Universidad ORT Uruguay

Facultad de Ingeniería Bernard Wand Polak

Programación de Redes

Documentación Obligatorio 3

Sebastian Zolkwer - 222841

Alex Piczenik - 231093

Fecha: 22.11.2021
Grupo: M6B

Índice

Link al repositorio

https://github.com/ObligatorioProgramacionRedes2.git

Introducción

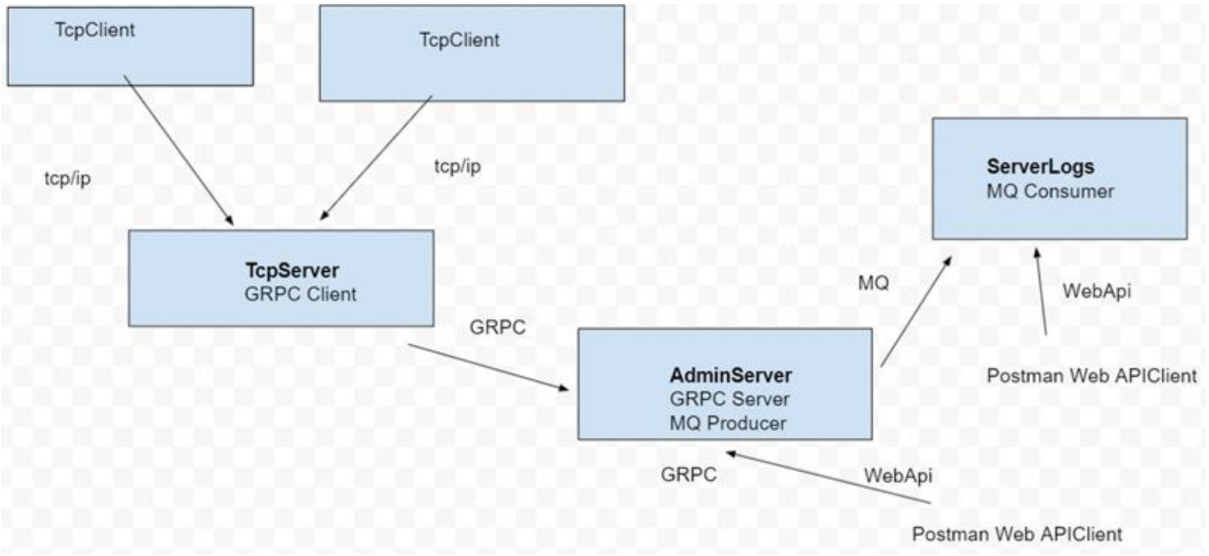
En el siguiente documento se detallan y justifican todos los aspectos tenidos en cuenta para realizar un sistema encargado de administrar Juegos y Usuarios, manteniendo un registro de logs para todas las acciones realizadas en el mismo. Se hará hincapié en la estructura de software utilizada y las justificaciones de diseño tomadas por el equipo.

Cumplimiento de requerimientos

Se implementaron todas las funcionalidades requeridas por la consigna, sin encontrar evidencia de errores. De todos modos, somos conscientes que la ausencia de evidencia no es equivalente a evidencia de ausencia y que como todo sistema, tiene margen de mejoras y aspectos a corregir que mejoren la calidad del mismo.
Para las comunicaciones entre componentes se utilizaron las 4 tecnologías requeridas: TCP, GRPC, RabbitMQ y REST API.

Arquitectura del sistema

Con respecto a la entrega anterior, analizaremos los cambios en la arquitectura del sistema, explicando todos los componentes, su responsabilidad y la comunicación entre ellos.



En el sistema contamos con los siguientes componentes:

Cliente y Servidor TCP desde donde se permite realizar todas las funcionalidades existentes. Los clientes Tcp se comunican con el Servidor Tcp y éste se comunica mediante grpc, con el server Admin.
Servidor Logs, encargado de registrar todos las acciones realizadas en el sistema recibiendo la información desde el servidor Admin mediante MQ Rabbit.
WebApi REST encargada de comunicarse directamente con el servidor logs para poder obtener los logs y filtrarlos.
Servidor Administrativo, donde se realizan todas las validaciones y métodos CRUD para cada uno de los recursos asociados al sistema. Es el encargado de centralizar la información y las listas de persistencia, siendo el único medio para acceder a las mismas.

Como ya explicamos, Server TCP se comunica con él mediante GRPC y se comunica con otra REST web api llamada Web Api Logic.
La WebApi Logic se comunica directamente mediante su businessLogic con el servidor administrativo, pudiendo acceder a las mismas funcionalidades que el TCP server y utilizando los mismos métodos para realizarlas.
A continuación explicaremos cada uno de los componentes en profundidad, mostrando ejemplos de código y dejando en claro sus respectivas responsabilidades.

Cliente y Servidor TCP

Al haber sido justificados en profundidad en las entregas anteriores, profundizaremos únicamente en las modificaciones realizadas en estos componentes para esta tercera entrega. El servidor TCP contaba con la responsabilidad de conocer y administrar las listas de recursos, accediendo directamente a la lógica para realizar las operaciones de ABM. En esta nueva instancia, el servidor se comporta únicamente como un “pasa manos” encargado de recibir las solicitudes de clientes y enviarlas mediante grpc al server admin (nuevo responsable de realizar los ABM).

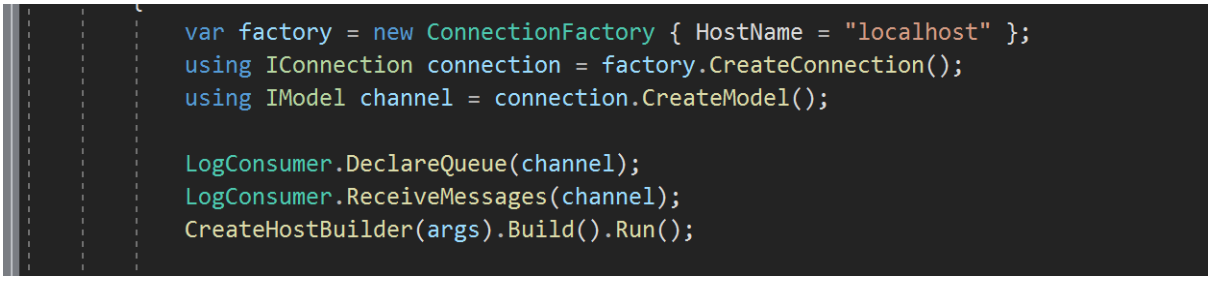
```
    case ProtocolMethods.Buy:
        request = await Protocol.Protocol.RecieveAndDecodeVariableDataAsync(networkStream, header.GetDataLength());
        response = await user.BuyGameAsync(new Request
        {
            Attributes = request,
            Name = client
        });
        await Protocol.Protocol.SendAndCodeAsync(networkStream, response.Status, response.Message, ProtocolMethods.Response);
        break;
```

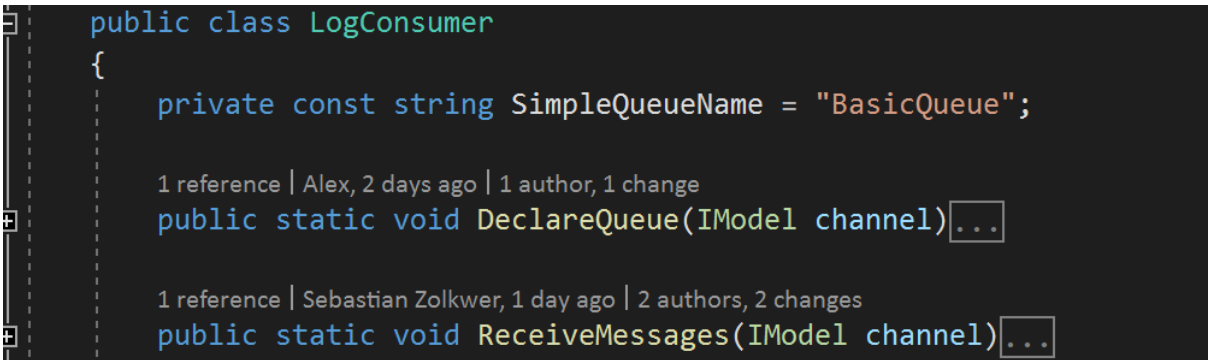
Como se puede ver en la captura, en el servidor se recibe la información encriptada mediante el protocolo ya definido, e instantáneamente se envía la request al server admin, retornando una response para continuar con el flujo de la aplicación.

Web Api Logs y Servidor Logs

El servidor Logs es el encargado de registrar todos los Logs lanzados por el sistema, logrando generar un registro de todas las acciones completadas o fallidas y su autor. Al momento de seleccionar la tecnología para implementar dicho funcionalidad, notamos que MOM era la más adecuadas basandonos en su capacidad para el almacenamiento de mensajes (strings) , la posibilidad de estar siempre a la espera para recibir los mensajes enviados, y su practicidad para enviar y recibir. Es por esto que al haberlo implementado utilizando MQ Rabbit, contamos con un productor y un consumidor. El productor se encuentra ubicado en una clase LogConnection dentro del server Admin ya que se utiliza para enviar un Log desde la lógica en cada oportunidad que así se requiera. Por otro lado, en la WebApiLogs, contamos con una clase LogConsumer, encargada de estar constantemente a la espera de recibir mensajes desde el emisor, y almacenarlos en el LogRepository.

Como se ve en la imagen, la clase LogConnection cuenta con los dos métodos necesarios para poder enviar mensajes al servidor. El método DeclareQuere es el que inicializa la cola y PublishMessage es el encargado de enviar el mensaje al receptor. Como equipo, decidimos enviar un mensaje de tipo string diagramado del siguiente modo: “UserName-TitleGame-Date-Message”.





En las capturas anteriores se observa la implementación de MQRabbit desde el consumidor, desde donde se inicializa el servidor, y se lo mantiene a la espera de cualquier mensaje recibido.
En el método ReceiveMessage, se recibe el string enviado y se lo convierte automáticamente en un objeto de tipo Log, agregando de inmediato a la lista de logs almacenada en el repositorio.
El objeto Log tiene como objetivo realizar un manejo más correcto e intuitivo de los logs en el sistema, facilitando el desarrollo de los métodos de filtrado pedidos en la consigna. Se define del siguiente modo:

```
7   public class Log
8   {
9       public string clientName { get; set; }
10      public string gameTitle { get; set; }
        2 references | Alex, 2 days ago | 1 author, 1 change
11      public string date { get; set; }
        1 reference | Sebastian Zolkwer, 1 day ago | 1 author, 1 change
12      public string message { get; set; }
13
```

Por otro lado, como indica el diagrama inicial, nuestro Servidor Logs se conecta mediante una REST WebApi con un cliente postman, permitiéndole utilizar los verbos HTTP para lanzar solicitudes al sistema. En nuestro caso, como ya mencionaremos a continuación, decidimos trabajar con capas e inyección de dependencias entre ellas, por lo que implementamos un LogController, encargado de comunicarse con la lógica de Logs. Desde la WebApi se pueden obtener todos los logs e incluso filtrar por Juego, fecha y persona que realizó la acción.

```
12  [ApiController]
13  [Route("logs")]
    1 reference | Sebastian Zolkwer, 1 day ago | 2 authors, 2 changes
14  public class LogController : ControllerBase
15  {
16
17      private ILogLogic logLogic;
18
        0 references | Sebastian Zolkwer, 1 day ago | 2 authors, 2 changes
19      public LogController(ILogLogic logLogic)
20      {
21          this.logLogic = logLogic;
22      }
23
24      [HttpGet()]
        0 references | Alex, 2 days ago | 1 author, 1 change
25      public IActionResult GetAll([FromQuery] string userName, [FromQuery] string gameTitle, [FromQuery] string date)
26      {
27          var logs = logLogic.GetAll(userName, gameTitle, date);
28          return Ok(logs);
29      }
30  }
```

De este modo, en el método GetAll, se reciben los atributos deseados para realizar el filtrado y se canalizan directamente en una llamada a la lógica. Nos esforzamos en mantener el diseño de la Api lo más conciso y claro posible, dejando todo aspecto del código o de la lógica de negocios para las capas correspondientes.

ServerAdmin

El Servidor Administrativo es el responsable directo de realizar toda la lógica necesaria para realizar cada una de las funcionalidades de ABM solicitadas en la consigna. Además, es el único encargado de llevar la persistencia del sistema, siendo el único punto de acceso a las listas de Clients y Games. Al existir la posibilidad de acceder a las listas desde distintos clientes, se mantuvo el uso de locks de forma que en cada instancia que el código accede a

la lista, esta se bloquea sin permitir que ningún otro cliente pueda modificarla hasta que se libere el recurso.

Para comunicarse con el ServerAdmin desde el ServerTcp y desde la Api para realizar el ABM mencionado previamente se utiliza la tecnología GRPC. Esta nos permite desde un cliente llamar a un método o servicio como si fuera local. En nuestro caso utilizamos un servicio Unario ya que el cliente (ServerTcp) realiza una petición y el serverAdmin responde.
Grcp es un servicio de intercambio de mensajes, donde se utilizan los mensajes definidos en un archivo. En este archivo se declaran los métodos, las entradas y las salidas. En el archivo proto a cada atributo del message se le debe asignar un número sin repetir.

```
message Request {
  string attributes = 1;
  string name = 2;
 }

message Response {
  string message = 1;
  int32 status = 2;
 }

message ResponseClient {
  string name = 1;
  string message = 2;
  int32 status = 3;
 }
```

```
service Greeter {
  rpc CreateGame (Request) returns (GameResponse);
  rpc UpdateGame (Request) returns (GameResponse);
  rpc BuyGame (Request) returns (Response);
  rpc EvaluateGame (Request) returns (Response);
  rpc Show (Request) returns (Response);
  rpc Search (Request) returns (Response);
```

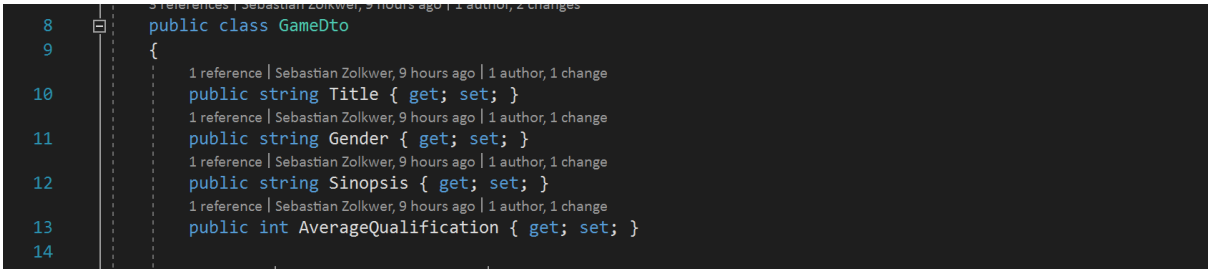
Estos mensajes son definidos en un archivo de servicios del AdminServer.
Por último para realizar el envío del mensaje y obtener respuesta se debe crear un canal, crear un cliente, crear la estructura que recibe el servicio que queremos utilizar, mandarlo y esperar la respuesta. El canal se debe conectar al puerto donde se encuentra el servidor grpc.

Con el fin de poder comunicarse directamente con el servidor administrativo, implementamos también una WebApi REST que permite utilizar las funcionalidades de ABM para clients y games y la posibilidad de que un usuario compre un juego. Al igual que lo explicado en la WebApiLogs, se utilizó el modelo de capas junto a inyección de dependencias, proponiendo el siguiente flujo.
Una vez realizada la request desde el postman, el Controller indicado comienza con la ejecución del código, llamando a la Lógica correspondiente al recurso en uso. Desde la lógica, se utiliza directamente GRPC para acceder al servidor administrativo desde donde se realiza efectivamente la acción.

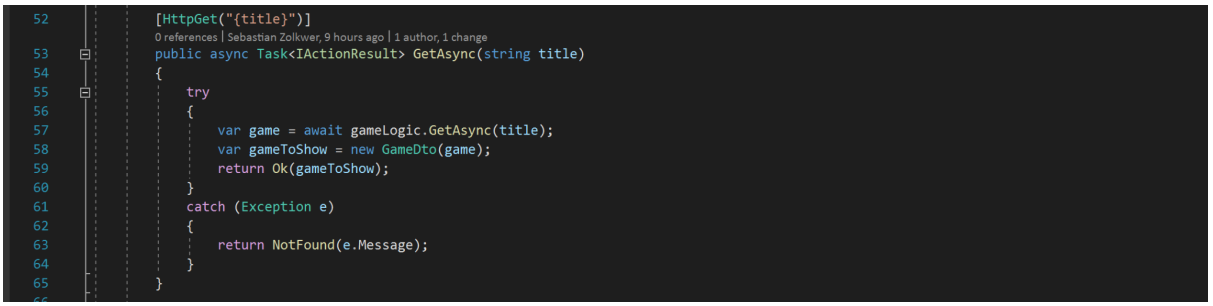
Justificaciones de Diseño

Utilización de Modelos (DTO)

Con el fin de poder desprender los objetos de dominio del conocimiento del usuario, y poder lograr que desde postman se trabaje con objetos equivalentes a los del dominio pero no los mismos decidimos utilizar DTO's. Los Modelos DTO permiten aplicar el patrón de diseño Information Hiding exponiendo al usuario únicamente la información mínima necesaria y suficiente para lograr su objetivo. En nuestro caso, decidimos crear modelos de retorno equivalentes a los objetos Game y Client del siguiente modo:



En este caso, concluimos que, por ejemplo, la lista de reviews no es un aspecto relevante a mostrar en la response, dadas las posibles request lanzadas por el usuario.



En el controller, creamos una nueva instancia del Dto correspondiente y lo retornamos dentro del ActionResult.

Como aspecto a mejorar, si bien en la gran mayoría de casos la request únicamente contiene strings equivalentes a nombre del client, título del juego, etc. en otros casos si se utilizan los objetos del dominio Game y Client.

Consideramos esto un punto a mejorar para una hipotética futura entrega, creando modelos que permitan desacoplar completamente al usuario con las clases de dominio.

Principio de inversión de dependencia

Al comenzar a pensar nuestro diseño, establecimos en pseudocodigo las distintas clases necesarias en cada una de las capas del proyecto, evidenciando que para lograr realizar un pasaje y gestión de datos desde la capa WebApi hasta el servidor correspondiente era conveniente realizar un modelo de capas adecuado, utilizando inyección de dependencias. Como respuesta,decidimos utilizar el siguiente modelo de capas:

- Capa Controladores: aquí es donde se realiza el punto de ingreso al servidor logs y uno de los posibles ingresos al servidor admin del sistema, se utilizan modelos de forma parcial para la comunicación con el usuario de forma que este no conozca directamente los objetos de dominio. El objetivo fue tener la menor cantidad de código posible en cada endpoint, dejando todo tipo de lógica de negocio para la siguiente capa.
- Capa BusinessLogic: En esta capa manejamos toda la lógica de negocio existente de acuerdo a los requerimientos del sistema. Nos encargamos de validar todos los requerimientos y prevenir problemas en la comunicación con los servidores.
- Capa DataAccess: En este caso contamos con repository únicamente para los Logs, donde almacenamos la lista de logs creados en todo el sistema.

Las clases de cada una de las capas no manejan instancias de sus clases siguientes sino que utilizamos interfaces para desprender a la lógica de una capa de la implementación de la otra. De este modo, cada capa conoce únicamente los métodos definidos y se desentiende de cómo fueron implementados, permitiendo fácilmente realizar por completo una nueva implementación.

Implementación de REST WebApi

A la hora de implementar una conexión entre cliente y servidor mediante nuestra Web Api decidimos utilizar REST como estilo de arquitectura. A continuación haremos énfasis en los distintos aspectos que tomamos como principios para el diseño de la API que a su vez representan buenas prácticas de REST.

- La API se diseñó utilizando un modelo de capas lo cual mantiene el bajo acoplamiento en el sistema además de la buena extensibilidad del mismo al separar funcionalidades por responsabilidades según la capa.
- Con respecto a las request:
    1. Se utilizan los verbos pertenecientes a la sigla CRUD (Create - Crear, Read- Leer, Update - Actualizar, Delete - Eliminar) mediante los comandos POST, GET, PUT y DELETE respectivamente.
    2. Se utilizan sustantivos (se evita usar verbos) en plural para el nombre de los recursos como /games
    3. Todos los nombres están escritos en minúscula.
    4. URLS intuitivas : Todas las urls utilizadas son intuitivas, es decir, es fácil de predecir qué acción realiza cada una y cuál será su posible respuesta.

- Se definen determinados HTTP Status Codes cómo response de los endpoints.

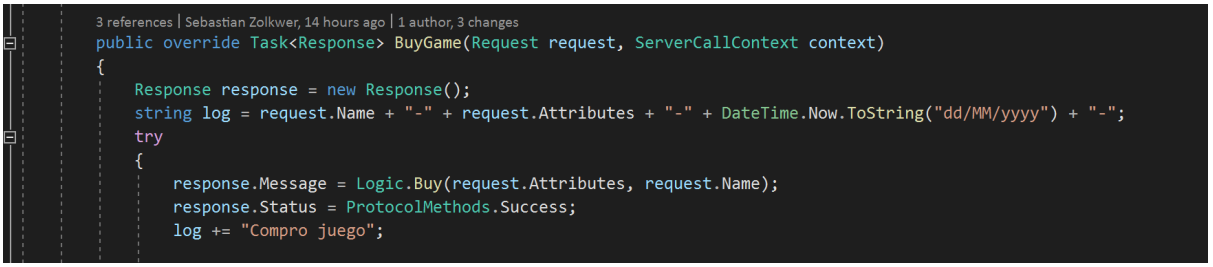
Cada uno de los errores posibles se indica con un código específico en función de lo que lo provocó. En nuestro caso, implementamos también una breve descripción del error con el fin de facilitar al usuario la comprensión y solución del mismo.

Asincronía en el sistema

Continuando con el proceso de migración de Threads al modelo asincrónico de .NET realizado en la entrega anterior, todos los cambios o nuevas implementaciones realizadas fueron desarrolladas en base a los conceptos de async y await.
Los conceptos async y await permiten convertir los métodos en asíncronos, creando una máquina de estados logrando que los métodos en lugar de bloquear la ejecución, se mantengan en espera.
Para migrar el cambio en el código definimos los métodos con async y la llamada a los mismos con await, recordando que todo método async debe tener un await.
Adjuntamos captura de métodos de las capas de WebApi y BusinesLogic y del desarrollo de la tecnología GRPC, respectivamente, que ejemplifican el concepto tratado.

```
52          [HttpGet("{title}")]
            0 references | Sebastian Zolkwer, 15 hours ago | 1 author, 1 change
53          public async Task<IActionResult> GetAsync(string title)
54          {
55              try
56              {
57                  var game = await gameLogic.GetAsync(title);
58                  var gameToShow = new GameDto(game);
59                  return Ok(gameToShow);
60              }
61              catch (Exception e)
62              {
63                  return NotFound(e.Message);
64              }
```

```
        2 references | Sebastian Zolkwer, 15 hours ago | 1 author, 1 change
        public async Task<string> ReturnGameAsync(string name, string title)
        {
            ValidateTitle(title);
            Response response = await user.DissociateGameAsync(new RequestClient
            {
                Attributes = title,
                Client = name,
                Name = "Admin"
            });
            if (response.Status == ProtocolMethods.Error)
            {
                throw new Exception(response.Message);
            }
            return response.Message;
        }
```



```
3 references | Sebastian Zolkwer, 14 hours ago | 1 author, 3 changes
public override Task<Response> BuyGame(Request request, ServerCallContext context)
{
    Response response = new Response();
    string log = request.Name + "-" + request.Attributes + "-" + DateTime.Now.ToString("dd/MM/yyyy") + "-";
    try
    {
        response.Message = Logic.Buy(request.Attributes, request.Name);
        response.Status = ProtocolMethods.Success;
        log += "Compro juego";
```

Comentarios generales

- Con el fin de poder identificar un usuario responsable para todas las acciones registradas en el sistema, y poder asociar dicho usuario al log correspondiente, se decidió utilizar el nombre “Admin” para definir todas las acciones realizadas desde el cliente postman WebApi.
- Para facilitar el filtrado de logs al usuario, se decidió reconocer la fecha del log como un string en el formato “DD/MM/YYYY”.
- Se agregaron endpoints no solicitados en la consigna para facilitar el flujo al momento de testear la aplicación. Estos son getAll y getByName entre otros.
- Se definio una coleccion de postman completa accesible con el siguiente link: https://www.getpostman.com/collections/9da2b413f8daed43212e

Conclusiones

El desarrollo del obligatorio nos permitió poner en práctica los conceptos aprendidos teóricamente durante todo el curso, enfrentándonos a errores inesperados sobre los cuales aprendimos y logramos solucionar. Si bien el uso de nuevas tecnologías siempre suele ser difícil en primera instancia, la posibilidad de contar con material del curso de apoyo, nos ayudó a dar el puntapié inicial y comprender a fondo la solución deseada.