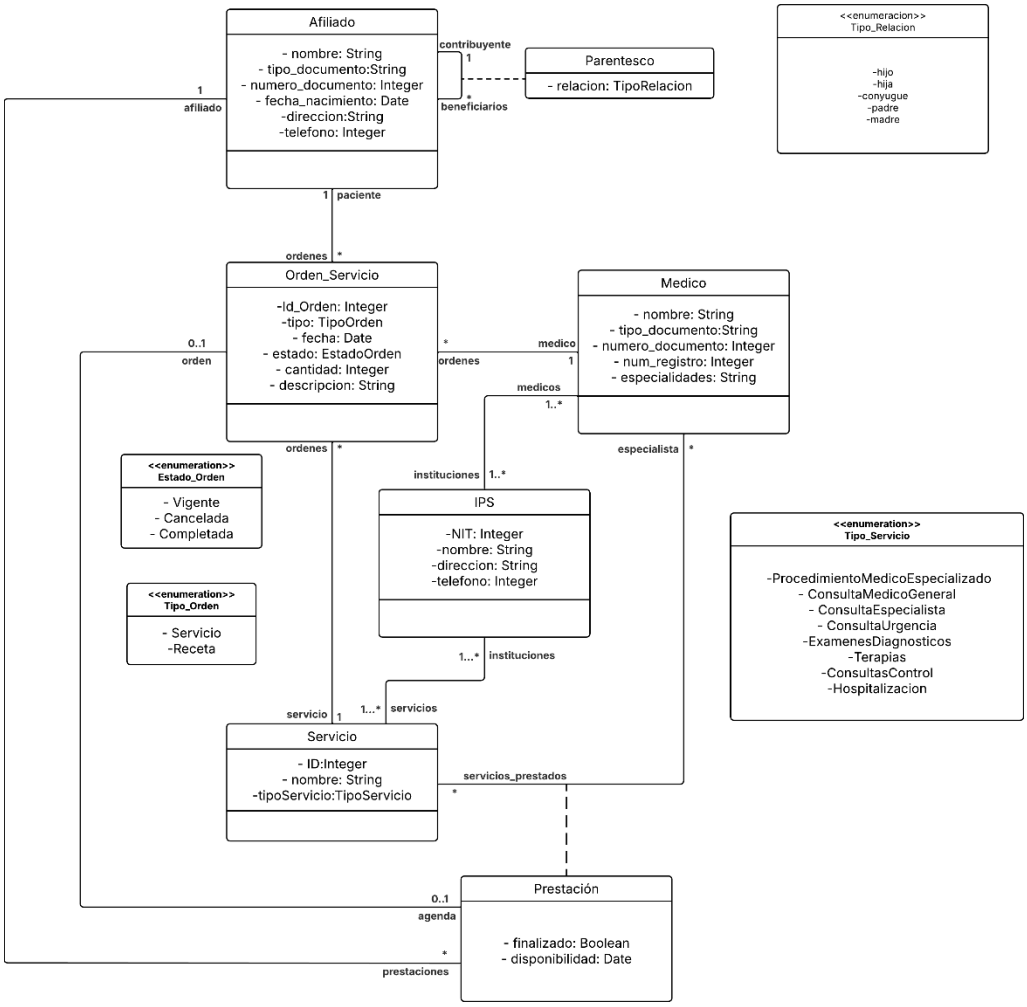
Gabriela Zambrano 202211712

María José Mantilla 202121670

María Inés Velásquez 202224325

Informe Proyecto Entrega 3

1. Modelo Conceptual



2. Diseño base de datos

a. Análisis de la carga de trabajo

i. Entidades y atributos

Las entidades que se identificaron son las siguientes:

Afiliado:

Los atributos que tiene esta clase son:

- nombre (String): Nombre completo del afiliado.

- tipo_documento (String): Tipo de documento de identidad (por ejemplo, CC, TI, CE, etc.).
- numero_documento (Integer): Número del documento de identidad del afiliado.
- fecha_nacimiento (Date): Fecha de nacimiento del afiliado.
- direccion (String): Dirección de residencia del afiliado.
- telefono (Integer): Número de teléfono de contacto del afiliado.

Parentesco :

Los atributos de esta clase son:

- Relacion (TipoRelacion): que puede tener los valores de la enumeración TipoRelacion: hijo, hija, cónyuge, padre, madre.

Oden de Servicio:

Los atributos que tiene esta clase son:

- Id Orden (Integer): Identificador único de cada orden.
- Tipo (tipoOrden): Tipo de orden (Servicio, Receta)
- Fecha (Date): Fecha en la que se creó la orden.
- Estado (EstadoOrden): Estado de la orden (Vigente, Cancelada, Completada).
- Cantidad (Integer): Si la orden corresponde a una terapia, aquí va el número de sesiones
- Descripción (String): Descripción de la orden incluyendo diferentes detalles extra.

Servicio:

Los atributos que tiene esta clase son:

- ID (Integer): Identificador unico del servicio.
- nombre (String): Nombre del servicio.
- tipoServicio(TipoServicio): Tipo del servicio (Procedimiento Médico Especializado, Consulta Medico General, Consulta Especialista, Consulta Urgencia, Exámenes Diagnósticos, Terapias, Consultas Control, Hospitalización)

Prestación:

Los atributos que tiene esta clase son:

- Finalizado (Boolean): Indica si el servicio ya fue realizado y finalizado o si no ha completado todavía.
- disponibilidad (Date): Fecha en la que está disponible para una cita, consulta o actividad programada.

Medico:

Los atributos que tiene esta clase son:

- nombre (String): nombre completo del médico.
- tipo_documento (String): tipo de documento de identidad (por ejemplo, CC, TI, CE, etc.).
- numero_documento (Integer): número de documento de identidad del médico, el cual también es el identificador único.
- num_registro (Integer): número de registro único del médico.
- especialidades (String): Lista especialidades del médico.

IPS:

- NIT (Integer): identificador numérico único por cada IPS.
- nombre (String): nombre de la IPS.
- dirección (String): dirección de la IPS.
- teléfono (Integer): número telefónico y de contacto de la IPS:

ii. Cuantificar las Entidades

Afiliado: Se crean en promedio 9000 veces cada mes, se consultan 10000 veces por día. El total de datos en la base de datos es de 900000-10000000 afiliados.

Parentesco: No se especifica cual es la carga de esta entidad.

Servicio: Se crean o modifican en promedio una vez al mes y se consultan 10.000 veces por día. El total de servicios en la base de datos es de 8.

Orden de servicio: Se crean en promedio 3000 veces cada día y se consultan en promedio 3000 veces al día. El total de ordenes de servicios en la base de datos es en promedio de unas 90000 al mes

Prestación: Se crean en promedio 2000 veces cada día y se consultan en promedio 5000 veces por día. El total de prestaciones (citas) en la base de datos son 90000 al mes, es decir, 1000000 de citas en un año.

Médico: Se crean o modifican en promedio 100 veces cada mes y se consultan 5000 veces por día. El total de datos en la base de datos es de 500000-600000 médicos.

IPS: Se crean o modifican 10 veces cada mes y se consultan 5000 veces por día. El total de datos en la base de datos es de 50000-60000 IPSs.

iii. Analicen las operaciones de lectura y escritura

Entidades	Operaciones	Información necesaria	Tipo

Afiliado	Crear/actualizar un afiliado	Datos del afiliado + si tiene beneficiario	Write
Afiliado	Consultar datos de un afiliado	Numero_documento	Read
Parentesco	Consultar que contribuyentes tiene un afiliado	Datos de afiliado a buscar + el resto de los afiliados	Read
Parentesco	Consultar si un afiliado tiene un beneficiario	Número de documento de afiliado	Read
Parentesco	Actualizar si un afiliado tiene un beneficiario	Número de documento de afiliado + si tiene beneficiario	Write
Medico	Crear/actualizar un medico	Datos del medico	Write
Medico	Consultar datos de un médico	Número de documento	Read
IPS	Crear/actualizar un afiliado	Datos de la IPS	Write
IPS	Consultar datos de un afiliado	NIT	Read
Servicio	Crear/actualizar un servicio	Nombre y tipo de servicio	Write
Servicio	Consultar datos de un servicio	Id del servicio	Read
Orden de Servicio	Crear/actualizar una orden de servicio	Datos de la orden + doctor + Id Servicio+ afiliado	Write
Orden de Servicio	Consultar datos de una orden se servicio	Id de la orden	Read
Prestación	Crear/actualizar una prestación	Id servicio+ doctor + Boolean si ya finalizo o no + fecha de la prestación	Write
Prestación	Consultar una prestación	Id Prestación	Read
Médico	Afiliar un médico a una IPS	Número documento médico + NIT	Write

iv. Cuantifiquen las operaciones de lectura y escritura

Entidades	**Operaciones**	**Información necesaria**	**Tipo**	**Rate**
Afiliado	Crear/actualizar un afiliado	Datos del afiliado + si es contribuyente o beneficiario	Write	9000/mes
Afiliado	Consultar datos de un afiliado	Numero_documento	Read	10000/día
Parentesco	Consultar que contribuyentes tiene un afiliado	Datos de afiliado a buscar + el resto de los afiliados	Read	-
Parentesco	Consultar si un afiliado tiene un beneficiario	Número de documento de afiliado	Read	-

Parentesco	Actualizar si un afiliado tiene un beneficiario	Número de documento de afiliado + si tiene beneficiario	Write	-
Medico	Crear/actualizar un medico	Datos del medico	Write	100/mes
Medico	Consultar datos de un médico	Número de documento	Read	5000/día
IPS	Crear/actualizar un afiliado	Datos de la IPS	Write	10/mes
IPS	Consultar datos de un afiliado	NIT	Read	5000/día
Servicio	Crear/actualizar un servicio	Nombre y tipo de servicio	Write	1/mes
Servicio	Consultar datos de un servicio	Id del servicio	Read	10.000/día
Orden de Servicio	Crear/actualizar una orden de servicio	Datos de la orden + doctor + Id Servicio+ afiliado	Write	3.000/mes
Orden de Servicio	Consultar datos de una orden se servicio	Id de la orden	Read	3.000/mes
Prestación	Crear/actualizar una prestación	Id servicio+ doctor + Boolean si ya finalizo o no + fecha de la prestación	Write	2.000/día
Prestación	Consultar una prestación	Id Prestación	Read	5.000/día
Médico	Afiliar un médico a una IPS	Número documento médico + NIT	Write	100/mes

b. Colecciones de datos y relaciones entre ellas NoSQL

i. Lista entidades

A partir de las consultas que se van a realizar y la carga de trabajo con la que se contara, se dejaron las siguientes entidades:

- Servicio
- Afiliado
- Medico
- IPS
- Prestación: entidad que se encontrara dentro de servicio
- OrdenServicio: entidad que se encontrara dentro de afiliado

ii. Relaciones entre entidades y cardinalidad

A continuación, se encuentran las diferentes relaciones entre entidades establecidas:

- Servicio-Medico: Esta relación es muchos a muchos y hace referencia a los servicios que realizan varios médicos. A esto se le llama una prestación.
- Afiliados-Afiliados: Esta es la relación que se usa para representar a la relación reflexiva de parentesco. Con cardinalidad de muchos a muchos.
- Afiliados-Prestación: Esta es la relación que hace referencia a las prestaciones que son ofrecidas a cada afiliado. Esta relación es de uno a muchos.
- Afiliados-Orden_Servicio: Esta es la relación que hace referencia a las ordenes creadas para cada afiliado. Esta relación es de uno a muchos.
- Prestacion-Orden_Servicio: Esta relación de uno a uno, hace referencia a la prestación del servicio mandado en la orden.
- Médico-Orden_Servicio: Esta relación hace referencia a las órdenes de servicio que prescribe cada médico. Con cardinalidad 1 a muchos.
- Médico-IPS: Esta relación representa las ips a las que están inscritos los médicos. Tiene cardinalidad muchos a muchos.
- IPS-Servicio: Esta relación hace referencia a los servicios prestados por una IPS. Tiene cardinalidad muchos a muchos.
- IPS-Médico: Esta relación hace referencia a los médicos que prestan los servicios brindados por una IPS. Cardinalidad muchos a muchos.
- Servicio-Orden_Servicio: Esta relación hace referencias a las órdenes de servicio creadas para un servicio en específico. Por ende, esta relación es uno a muchos.

iii. Análisis de selección de esquema de asociación

Nombre	Pregunta	Embeber	Referenciar
Simplicidad	Si mantener piezas de información conjuntamente me llevaría a un modelo más simple	Si	No
Ir juntos	Si la relación entre los datos es del tipo "partes de", "tiene"	Si	No
Atomicidad Cola	Si las consultas deben obtener la información conjuntamente	Si	No
Complejidad actualización	Si los datos se deben actualizar conjuntamente	Si	No
Almacenar	Si los datos se deben almacenar al mismo tiempo	Si	No
Cardinalidad	Si hay una alta cardinalidad de los hijos en la relación	No	Si
Duplicación Datos	Si la duplicación de datos sería muy compleja y difícil de maneja	No	Si
Tamaño Documento	La combinación de los tamaños de las piezas de información va a ocupar mucha memoria	No	Si
Crecimiento Documento	Si los datos que debería embeber podrían crecer sin límite	No	Si

Carga de trabajo	Si la información se escribe en diferentes momentos en procesos intensivos	No	Si
Individualidad	Si las relaciones pueden existir independientemente de la otra	No	Si

Afiliados-Afiliados:

Nombre	Respuesta
Simplicidad	No
Ir juntos	Si
Atomicidad Cola	Si
Complejidad actualización	No
Almacenar	Si
Cardinalidad	Si
Duplicación Datos	Si
Tamaño Documento	No
Crecimiento Documento	Si
Carga de trabajo	No
Individualidad	Si

Teniendo en cuenta el análisis de la tabla, lo mejor podría ser referenciar al beneficiario, para cada afiliado que lo requiera. Para esto se agregó el atributo parentesco en afiliado, el cual apunta a un ObjectId de otro afiliado.

Afiliados-Prestación:

Nombre	Respuesta
Simplicidad	No
Ir juntos	Si
Atomicidad Cola	Si
Complejidad actualización	No
Almacenar	No
Cardinalidad	Si
Duplicación Datos	Si
Tamaño Documento	No
Crecimiento Documento	Si
Carga de trabajo	Si
Individualidad	No

En este caso, es mejor referenciar la relación. Teniendo en cuenta las consultas que se realizaran, se decidió referenciar el afiliado en la prestación. Para esto se agregó un atributo de afiliado en prestación, que apunta a un afiliado.

Afiliados-Orden_Servicio:

Nombre	Respuesta
Simplicidad	Si
Ir juntos	Si
Atomicidad Cola	No
Complejidad actualización	No
Almacenar	No
Cardinalidad	No
Duplicación Datos	No
Tamaño Documento	Si
Crecimiento Documento	No
Carga de trabajo	Si
Individualidad	No

En este caso, es mejor embeber las ordenes de servicio en los afiliados. Esto debido a se suelen consultar las ordenes de servicio de un paciente y una orden debe tener un afiliado para existir. Para esto se agregó un atributo de ordenesServicio en afiliado, que embebe las ordenServicio que pertenezcan a ese afiliado.

Prestacion-Orden_Servicio:

Nombre	Respuesta
Simplicidad	No
Ir juntos	Si
Atomicidad Cola	No
Complejidad actualización	No
Almacenar	No
Cardinalidad	No
Duplicación Datos	Si
Tamaño Documento	No
Crecimiento Documento	No
Carga de trabajo	Si
Individualidad	Si

En este caso, es mejor referenciar las órdenes en la prestación. Esta decisión se toma de acuerdo con la tabla anterior y a que las prestaciones pueden o no necesitar de una orden para ser agendadas. Para esto se agregó un atributo de orden en prestación, que apunta a una orden.

Médico-Orden_Servicio:

Nombre	Respuesta
Simplicidad	No
Ir juntos	No
Atomicidad Cola	No

Complejidad actualización	No
Almacenar	No
Cardinalidad	Si
Duplicación Datos	Si
Tamaño Documento	Si
Crecimiento Documento	Si
Carga de trabajo	No
Individualidad	No

En este caso es mejor referenciar las ordenes de servicio, ya que no es común que al consultar un médico sea necesario consultar las órdenes creadas por este. Para esto se agregó un atributo de medico en ordenSerivicio, que apunta a un médico, para saber quién la creo.

IPS-Médico:

Nombre	Respuesta
Simplicidad	No
Ir juntos	Si
Atomicidad Cola	Si
Complejidad actualización	No
Almacenar	No
Cardinalidad	Si
Duplicación Datos	Si
Tamaño Documento	No
Crecimiento Documento	Si
Carga de trabajo	No
Individualidad	Si

En este caso es mejor referenciar los médicos a las IPS, ya que, una IPS puede llegar a tener muchos médicos y expandirse mucho. Para esto se agregó un atributo de médicos en IPS, que apunta a una lista de médicos, que atienden con esa IPS.

IPS-Servicio:

Nombre	Respuesta
Simplicidad	Si
Ir juntos	Si
Atomicidad Cola	Si
Complejidad actualización	No
Almacenar	No
Cardinalidad	Si
Duplicación Datos	No
Tamaño Documento	Si
Crecimiento Documento	Si
Carga de trabajo	Si

Individualidad	Si

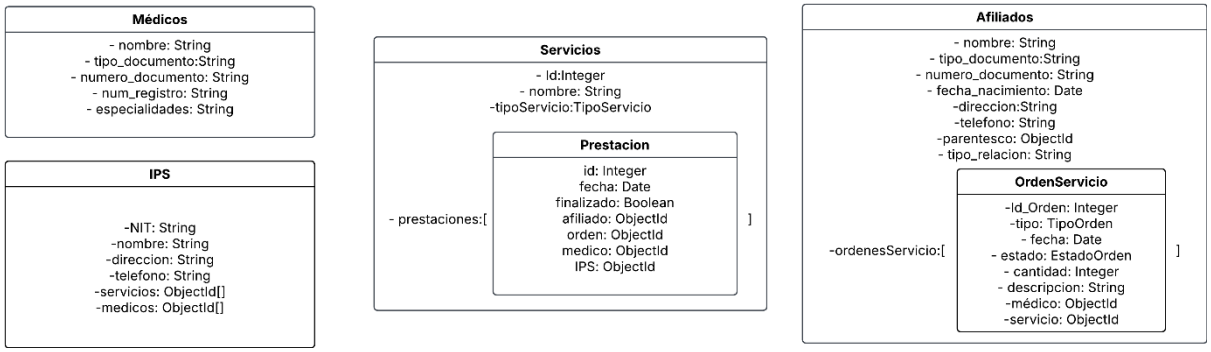
En este caso es mejor referenciar los servicios, ya que, estos pueden pertenecer a varias IPS y en si van a crecer mucho por la cantidad de prestaciones que pueda haber. Para esto se agregó un atributo de servicios en IPS, que contiene una lista que apunta a cada uno de los servicios que se prestan en esa IPS.

Orden_Servicio-Servicio:

Nombre	Respuesta
Simplicidad	Si
Ir juntos	No
Atomicidad Cola	Si
Complejidad actualización	No
Almacenar	Si
Cardinalidad	No
Duplicación Datos	No
Tamaño Documento	No
Crecimiento Documento	No
Carga de trabajo	Si
Individualidad	Si

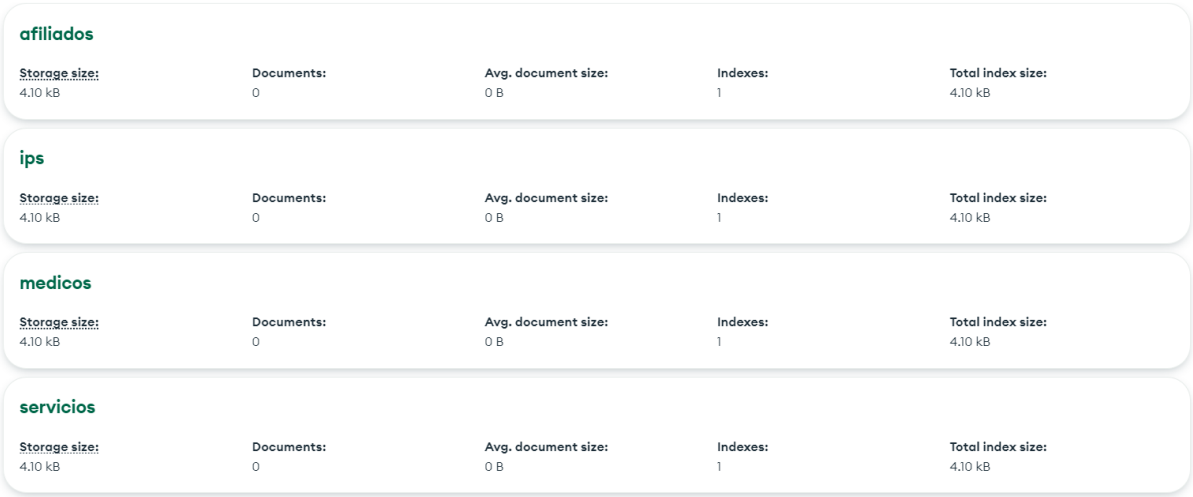
En este caso es mejor referenciar los servicios en las órdenes. Para esto se agregó un atributo de servicio en ordenServicio, que apunta al servicio al que hace referencia la orden.

iv. Descripción gráfica usando Json



En la imagen anterior, se puede ver la representación gráfica de la base de datos NoSQL. En esta se pueden observar las 4 colecciones principales que van a ser creadas, siendo estas: Medicos, IPS, Servicios y Afiliados. Adicionalmente, las referencias de los documentos se observan como arreglos de tipo ObjectId, o simplemente el campo de tipo ObjectId. Por otra parte, las dos entidades que fueron embebidas se representan como otro objeto dentro de la colección, en este caso sería Prestacion y OrdenServicio.

c. Creación MongoDB de colecciones principales



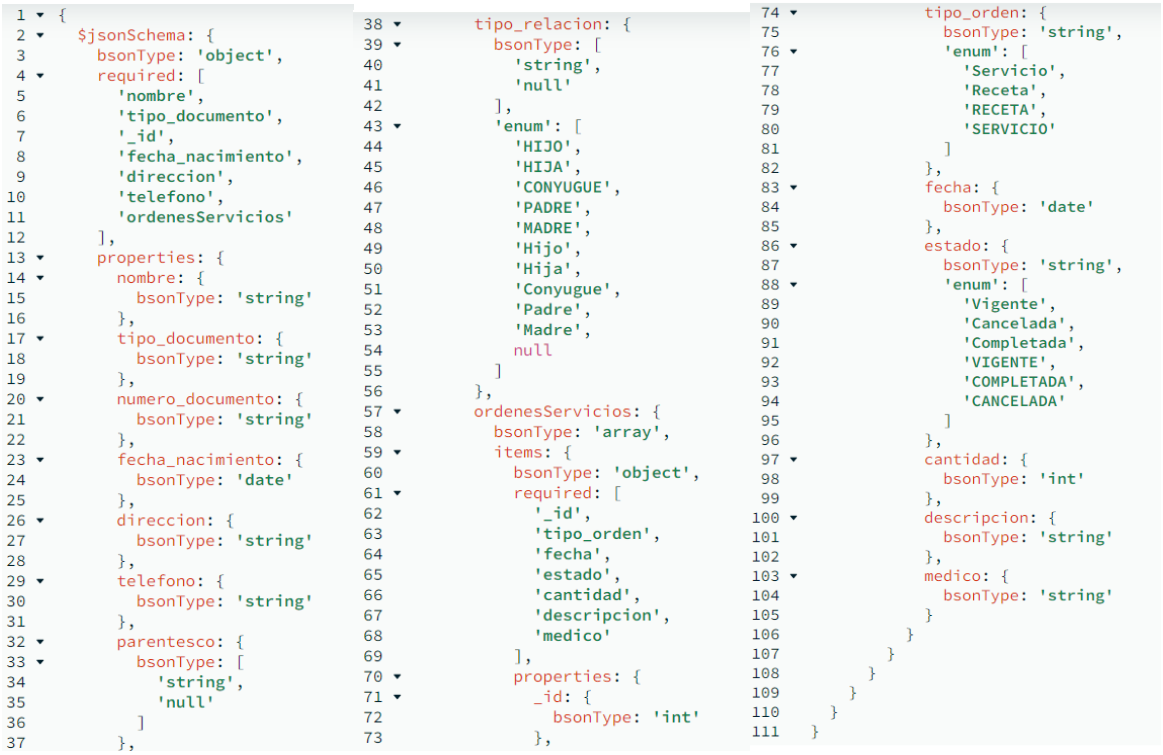
d. Creación esquemas de validación

```
{
  $jsonSchema: {
    bsonType: 'object',
    required: [
      '_id',
      'nombre',
      'direccion',
      'telefono',
      'medicos'
    ],
    properties: {
      NIT: {
        bsonType: 'string'
      },
      nombre: {
        bsonType: 'string'
      },
      direccion: {
        bsonType: 'string'
      },
      telefono: {
        bsonType: 'string'
      },
      medicos: {
        bsonType: 'array',
        items: {
          bsonType: 'string'
        }
      }
    }
  }
}
```

Para la colección **IPS** se planteó el siguiendo esquema de validación, este indica que se requiere un identificador, nombre, dirección, teléfono y médicos registrados para que exista una IPS. Además, indica que todos estos atributos deben ser de tipo String y los médicos deben ser una lista de strings , que son los números de documentos de los médicos (id's) al estar referenciando a la colección médicos.

Para la colección **médicos** se planteó el siguiente esquema de validación, en el cual se requiere nombre, tipo_documento, id (numero_documento), num_registro y especialidades. Además, se aclara que el tipo de todos los atributos es string y especialidades es una lista de strings.

```
{
  $jsonSchema: {
    bsonType: 'object',
    required: [
      'nombre',
      'tipo_documento',
      '_id',
      'num_registro',
      'especialidades'
    ],
    properties: {
      nombre: {
        bsonType: 'string'
      },
      tipo_documento: {
        bsonType: 'string'
      },
      numero_documento: {
        bsonType: 'string'
      },
      num_registro: {
        bsonType: 'string'
      },
      especialidades: {
        bsonType: 'array',
        items: {
          bsonType: 'string'
        }
      }
    }
  }
}
```



```
{
  $jsonSchema: {
    bsonType: 'object',
    required: [
      'nombre',
      'tipo_documento',
      '_id',
      'fecha_nacimiento',
      'direccion',
      'telefono',
      'ordenesServicios'
    ],
    properties: {
      nombre: {
        bsonType: 'string'
      },
      tipo_documento: {
        bsonType: 'string'
      },
      numero_documento: {
        bsonType: 'string'
      },
      fecha_nacimiento: {
        bsonType: 'date'
      },
      direccion: {
        bsonType: 'string'
      },
      telefono: {
        bsonType: 'string'
      },
      parentesco: {
        bsonType: [
          'string',
          'null'
        ]
      },
      tipo_relacion: {
        bsonType: [
          'string',
          'null'
        ],
        'enum': [
          'HIJO',
          'HIJA',
          'CONYUGUE',
          'PADRE',
          'MADRE',
          'Hijo',
          'Hija',
          'Conyugue',
          'Padre',
          'Madre',
          null
        ]
      },
      ordenesServicios: {
        bsonType: 'array',
        items: {
          bsonType: 'object',
          required: [
            '_id',
            'tipo_orden',
            'fecha',
            'estado',
            'cantidad',
            'descripcion',
            'medico'
          ],
          properties: {
            _id: {
              bsonType: 'int'
            },
            tipo_orden: {
              bsonType: 'string',
              'enum': [
                'Servicio',
                'Receta',
                'RECETA',
                'SERVICIO'
              ]
            },
            fecha: {
              bsonType: 'date'
            },
            estado: {
              bsonType: 'string',
              'enum': [
                'Vigente',
                'Cancelada',
                'Completada',
                'VIGENTE',
                'COMPLETADA',
                'CANCELADA'
              ]
            },
            cantidad: {
              bsonType: 'int'
            },
            descripcion: {
              bsonType: 'string'
            },
            medico: {
              bsonType: 'string'
            }
          }
        }
      }
    }
  }
}
```

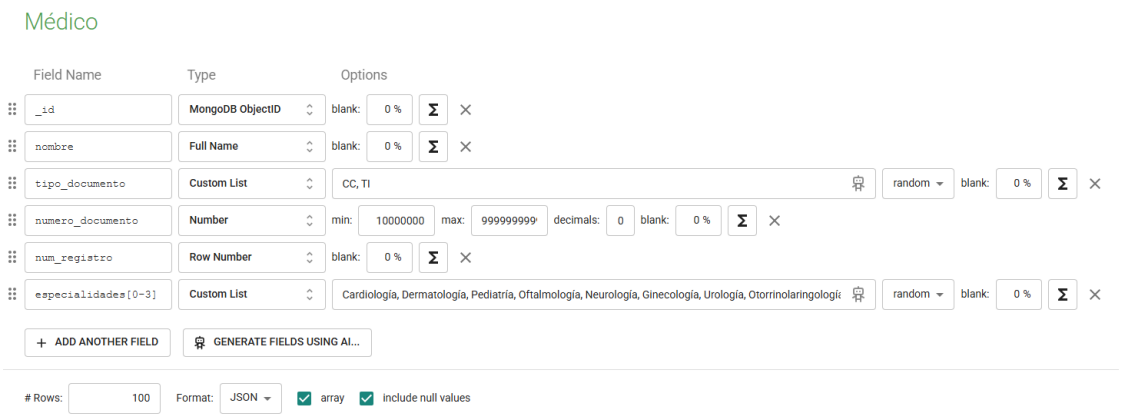
Para la colección de **afiliados** se planteó el siguiente esquema de validación, en el cual se indica que se requiere que tenga nombre, tipo de documento, id (número de documento), fecha de nacimiento, dirección, teléfono y ordenes de servicio. Al establecer el tipo de cada atributo, cabe resaltar que parentesco y tipo de relación pueden ser null si el afiliado es un contribuyente y no un beneficiario. Además, se indicó los enum para los atributos que tienen un tipo como tipo relación, tipo orden y estado. Por último, se indicó los atributos necesarios para cada orden de servicio que se encuentra embebido en afiliados.

```
{
  $jsonSchema: {
    bsonType: 'object',
    required: [
      '_id',
      'nombre',
      'tipo_servicio',
      'prestaciones'
    ],
    properties: {
      nombre: {
        bsonType: 'string'
      },
      tipo_servicio: {
        bsonType: 'string',
        'enum': [
          'ProcedimientoMedicoEspecializado',
          'ConsultaMedicoGeneral',
          'ConsultaEspecialista',
          'ConsultaUrgencia',
          'ExamenesDiagnosticos',
          'Terapias',
          'ConsultasControl',
          'Hospitalizacion'
        ]
      },
      prestaciones: {
        bsonType: 'array',
        items: {
          bsonType: 'object',
          required: [
            'fecha',
            'finalizado',
            'medico',
            'ips'
          ],
          properties: {
            fecha: {
              bsonType: 'date'
            },
            finalizado: {
              bsonType: 'bool'
            },
            afiliado: {
              bsonType: [
                'string',
                'null'
              ]
            },
            orden: {
              bsonType: [
                'int',
                'null'
              ]
            },
            medico: {
              bsonType: 'string'
            },
            ips: {
              bsonType: 'string'
            }
          }
        }
      }
    }
  }
}
```

Para la colección **servicios**, se estableció el anterior esquema de validación, donde se requiere tener un identificador, nombre, tipo servicio y prestaciones. Los primeros dos atributos deben ser de tipo string, tipo servicio también es string pero con un enum con los tipos de servicios permitidos. Por último, prestaciones es un array de string y se indican los atributos de esta, ya que esta embebida en servicios, en esta los únicos atributos que pueden ser null son afiliado por si nadie ha reservado esa prestación y orden por si el servicio no requiere una.

3. **Población BD**

   Para poblar las colecciones se utilizó Mockaroo. En este se crearon los siguientes esquemas:



IPSs

Field Name	Type	Options
_id	MongoDB ObjectID	blank: 0 %
NIT	Number	min: 100000000 max: 999999999 decimals: 0 blank: 0 %
nombre	Fake Company Name	blank: 0 %
direccion	Street Address	blank: 0 %
telefono	Phone	format: (###) ###-#### blank: 0 %
medicos[10-25]	Custom List	68338479fc13ae442a28550a, 68338479fc13ae442a28550b, 68338479fc13ae442a28550c, 68338479fc · random blank: 0 %

ADD ANOTHER FIELD | GENERATE FIELDS USING AI...

Rows: 25 Format: JSON array include null values

Afiliado

Field Name	Type	Options
_id	MongoDB ObjectID	blank: 0 %
nombre	Full Name	blank: 0 %
tipo_documento	Custom List	CC, TI · random blank: 0 %
numero_documento	Number	min: 100000000 max: 999999999 decimals: 0 blank: 0 %
fecha_nacimiento	Datetime	01/01/1960 to 12/31/2020 format: yyyy-mm-dd blank: 0 %
direccion	Street Address	blank: 0 %
telefono	Phone	format: (###) ###-#### blank: 0 %
parentesco	MongoDB ObjectID	blank: 0 %
tipo_relacion	Custom List	Hijo, Hija, Padre, Madre, Conyugue · random blank: 0 %
ordenesServicios	JSON Array	min elements: 0 max elements: 3 blank: 0 %
ordenesServicios._id	Number	min: 1 max: 500 decimals: 0 blank: 0 %
ordenesServicios.tip	Custom List	Servicio, Receta · random blank: 0 %
ordenesServicios.fec	Datetime	01/01/2020 to 01/01/2030 format: yyyy-mm-dd blank: 0 %
ordenesServicios.est	Custom List	Vigente, Cancelada, Completada · random blank: 0 %
ordenesServicios.car	Number	min: 1 max: 3 decimals: 0 blank: 0 %
ordenesServicios.des	Sentences	at least 1 but no more than 3 blank: 0 %
ordenesServicios.med	Custom List	68338479fc13ae442a28550a, 68338479fc13ae442a28550b, 68338479fc13ae442a28550c, 68338479fc · random blank: 0 %

ADD ANOTHER FIELD | GENERATE FIELDS USING AI...

Rows: 300 Format: JSON array include null values

Servicio

Field Name	Type	Options
_id	MongoDB ObjectID	blank: 0 %
nombre	Custom List	Consulta de Medicina General, Consulta de Pediatría, Radiografía de Tórax, Ecografía Abdominal, Resona · random blank: 0 %
tipo_servicio	Custom List	ProcedimientoMedicoEspecializado, ConsultaMedicoGeneral, ConsultaEspecialista, ConsultaUrgencia, E · random blank: 0 %
prestaciones	JSON Array	min elements: 1 max elements: 5 blank: 0 %
prestaciones.fecha	Datetime	01/01/2020 to 01/01/2030 format: yyyy-mm-dd blank: 0 %
prestaciones.finali	Boolean	blank: 0 %
prestaciones.afiliac	Custom List	683385e0fc13ae449b2856ea, 683385e0fc13ae449b2856ec, 683385e0fc13ae449b2856ee, 683385e0fc1 · random blank: 50 %
prestaciones.orden	Custom List	390, 469, 230, 272, 356, 200, 454, 79, 291, 257, 312, 474, 334, 109, 66, 353, 352, 11, 198, 199, 125, 462, 42 · random blank: 50 %
prestaciones.medico	Custom List	68338479fc13ae442a28550a, 68338479fc13ae442a28550b, 68338479fc13ae442a28550c, 68338479fc · random blank: 0 %
prestaciones.ips	Custom List	68338556fc13ae4303285146, 68338556fc13ae4303285147, 68338556fc13ae4303285148, 68338556fc · random blank: 0 %

ADD ANOTHER FIELD | GENERATE FIELDS USING AI...

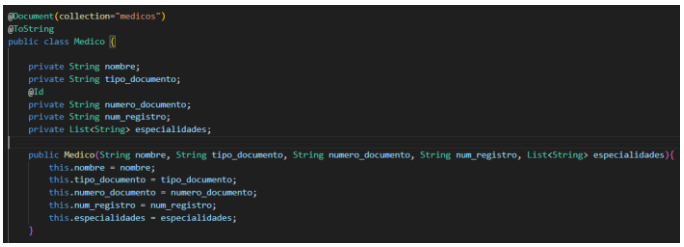
Rows: 100 Format: JSON array include null values

Para poder garantizar las relaciones entre las colecciones se implementó un archivo de Python que en primer lugar corrige los tipos de datos como en el caso de las fechas para que tengan el formato de ISOdate que se permite insertar en MongoDB Compass, corrige errores de lógica en cuanto a si una fecha ya pasó la prestación en esa fecha debe haber finalizado y viceversa. Por último, se imprimen los ids de cada colección para poder ser utilizados en la creación de las demás creaciones. El archivo correspondiente se encuentra adjunto como "correccionesColecciones.py". Cabe aclarar que es necesario que primero se cree la colección de médico, después IPS, Afiliado y finalmente Servicio. Los archivos de población correctos tambien se encuentran adjuntos ("Servicios.json", "Medicos.json", "Afiliados.json" y "IPS.json")

4. Implementación RF4-7 y RFC1-2

a. RF4

Para poder registrar un médico, se creó la clase médico, cuyas instancias se van a almacenar en la colección médicos y tiene como atributos nombre, tipo de documento, numero de documento (id), numero de registro y una lista de especialidades.



```
@Document(collection="medicos")
@ToString
public class Medico {

    private String nombre;
    private String tipo_documento;
    @Id
    private String numero_documento;
    private String num_registro;
    private List<String> especialidades;

    public Medico(String nombre, String tipo_documento, String numero_documento, String num_registro, List<String> especialidades){
        this.nombre = nombre;
        this.tipo_documento = tipo_documento;
        this.numero_documento = numero_documento;
        this.num_registro = num_registro;
        this.especialidades = especialidades;
    }
```

Teniendo esta información, en el controlador de médicos se planteó el método crear Médico el cual recibe la información del médico, luego se llama al repositorio de médico que a su vez usa save(medico) para registrarlo. Cuando ya se encuentran creados los médicos en la colección cada uno de estos es referenciado con su número de documento, en la IPS en la cual se encuentra registrado.

```
@PostMapping("/new/save")
public ResponseEntity<Map<String, String>>  crearMedico (@RequestBody Medico medico) {
    try {

        medicoRepository.insertarMedicos(medico);;
        Map<String, String> response = new HashMap<>();
        response.put(key:"mensaje", value:"Medico creado exitosamente");
        response.put(key:"numero_documento", medico.getNumero_documento());
        return new ResponseEntity<>(response, HttpStatus.CREATED);
    } catch (Exception e) {
        Map<String, String> response = new HashMap<>();
        response.put(key:"mensaje", "Error al crear el medico: " + e.getMessage());
        response.put(key:"numero_documento", value:null);
        return new ResponseEntity<>(response, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

b. RF5

Para poder registrar un afiliado creamos la clase afiliado, cuyas instancias van a hacer parte de la colección afiliados. Entre sus atributos están nombre, tipo de documento, numero de documento (id), fecha de nacimiento, dirección, teléfono, parentesco y relación (por si es un beneficiario) y ordenes de servicio que corresponde a ordenes de servicio que este embebido en la colección.

```java
@Document(collection = "afiliados")
@ToString
public class Afiliado {

    @Id
    private String numero_documento;

    private String nombre;
    private String tipo_documento;

    @JsonFormat(pattern = "dd-MM-yyyy")
    private LocalDate  fecha_nacimiento;
    private String direccion;
    private String  telefono;

    //Parentesco
    private String parentesco;

    private TipoRelacion relacion;

    //Orden Servicio
    private List<OrdenServicio> ordenesServicios;

    public Afiliado(String nombre, String tipo_documento, String numero_documento, LocalDate  fecha_nacim
        this.nombre = nombre;
        this.tipo_documento = tipo_documento;
        this.numero_documento = numero_documento;
        this.fecha_nacimiento = fecha_nacimiento;
        this.direccion = direccion;
        this.telefono = telefono;
        this.parentesco = parentesco_numero_documento;
        this.relacion = relacion;
    }
```

Ahora bien, en el controlador de afiliados hicimos un método llamado crear afiliado en cual se recibe la información del afiliado y utilizando el repositorio con la función save(afiliado) se registra.

```java
@PostMapping("/new/save")
public ResponseEntity<Map<String, String>>  crearAfiliado (@RequestBody Afiliado afiliado) {
    try {

        afiliadoRepository.insertarAfiliado(afiliado);
        Map<String, String> response = new HashMap<>();
        response.put(key:"mensaje", value:"Afiliado creado exitosamente");
        response.put(key:"numero_documento", afiliado.getNumero_documento());
        return new ResponseEntity<>(response, HttpStatus.CREATED);
    } catch (Exception e) {;
        Map<String, String> response = new HashMap<>();
        response.put(key:"mensaje", "Error al crear el afiliado: " + e.getMessage());
        response.put(key:"numero_documento", value:null);
        return new ResponseEntity<>(response, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

c. RF6

Como ya se mencionó, tenemos la colección afiliados que tiene embebida las ordenes de servicio mediante un array de órdenes. Por ende, para poder registrar una orden de servicio a un afiliado por parte de un médico se tiene que agregar esta nueva orden a la lista ordenes de servicio del afiliado. Además, uno de los atributos de la clase orden de servicio es médico, el cual referencia mediante el número de documento al médico que genero la orden.

```java
@PostMapping("/{id}/ordenes")
public ResponseEntity<Map<String, Object>> agregarOrden( @PathVariable String id,@RequestBody OrdenServicio nuevaOrden) {
    try {
        List<Afiliado> afiliadolist = afiliadoRepository.buscarAfiliado(id);
        if (afiliadolist == null || afiliadolist.isEmpty()) {
            Map<String, Object> response = new HashMap<>();
        response.put(key:"error", "Afiliado no encontrado con ID: " + id);
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(response);
        }
        Afiliado afiliado = afiliadolist.get(index:0);
        if (afiliado.getOrdenesServicios() == null) {
        afiliado.setOrdenesServicios(new ArrayList<>());
        }

        int maxId = afiliado.getOrdenesServicios().stream()
        .mapToInt(orden -> orden.getId_Orden() == null ? 0 : orden.getId_Orden())
        .max()
        .orElse(other:0);

        nuevaOrden.setId_Orden(maxId + 1);

        afiliado.getOrdenesServicios().add(nuevaOrden);
        afiliadoRepository.save(afiliado);
        Map<String, Object> response = new HashMap<>();
        response.put(key:"mensaje", value:"Orden registrada con éxito");
        response.put(key:"id_Orden", nuevaOrden.getId_Orden());

        return ResponseEntity.status(HttpStatus.CREATED).body(response);


    } catch (Exception e) {
        Map<String, Object> errorResponse = new HashMap<>();
    errorResponse.put(key:"error", "Error al registrar la orden de servicio: " + e.getMessage());
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(errorResponse);
    }


}
```

Para realizar esto, en el controlador de afiliado está el método agregar orden, en este encontramos el afiliado por el id, agregamos la orden al array o si no ha tenido ninguna se crea la lista con la primera orden y finalmente se guardan los cambios en el afiliado con save(afiliado).

d. RF7

- **Disponibilidad próximas 4 semanas**

Como ya mencionamos, tenemos la colección servicios en la cual se encuentra embebidas las prestaciones de este servicio, en la que se referencia el medico que lo va a prestar, la ips, el afiliado que quiere agendar esa prestación y la orden del servicio si se necesita. Además, cuenta con la fecha y hora de la prestación, así como el atributo finalizado (boolean) que indica si ya fue realizado el servicio que se agendo en esa prestación.

```java
@GetMapping("/{id}/prestaciones")
public ResponseEntity<List<Prestacion>> obtenerDisponibilidad(@PathVariable("id") String id) {
    try {
        List<Servicio> servicios = servicioRepository.buscarServicios(id);
        if (servicios == null || servicios.isEmpty()) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body(Collections.emptyList());
        }
        Servicio servicio = servicios.get(index:0);
        LocalDateTime actual = LocalDateTime.now();
        LocalDateTime enCuatroSemanas = actual.plusWeeks(weeks:4);


        List<Prestacion> disponibles = new ArrayList<>();
        for (Prestacion p : servicio.getPrestaciones()) {
        if (!p.getFecha().isBefore(actual) && !p.getFecha().isAfter(enCuatroSemanas) &&
            (p.getAfiliado() == null || p.getAfiliado().isEmpty())) {
        disponibles.add(p);
        }
    }
        return ResponseEntity.ok(disponibles);

    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(body:null);
    }

}
```

Para poder ver la disponibilidad se usó un método en el controlador de servicios, en el cual se identifica el servicio con el id y luego se define le rango de tiempo teniendo la fecha actual y la fecha futura (dentro de 4 semanas). Con esta información, se busca entre las prestaciones del servicio las que se encuentran disponibles entre esas 2 fechas y afiliado es null indicando que no han sido agendadas. Finalmente, se retorna esta nueva lista con las prestaciones del servicio que están disponibles en las próximas 4 semanas.

- **Agendar sin orden**

Para poder agendar un servicio debemos asignarle el número de documento del afiliado a la prestación, si es un servicio que no necesita orden como consulta con medico general o consulta de urgencia se hace de la siguiente manera:

Verifica el servicio con el id correspondiente luego se verifica que efectivamente no se necesite una orden para el servicio dependiendo del tipo. Acto seguido, iteramos sobre las prestaciones de ese servicio y filtramos por la fecha-hora ingresada por parámetro y que no tenga un afiliado ya asignado. Apenas encontremos una prestacion que cumpla con estos requisitos se asigna el afiliado a la prestación y se guarda el servicio con save(servicio). De esta forma ya queda agendada la prestación de ese servicio por parte de un afiliado.

```java
@PostMapping("/{id}/prestaciones/{fecha_hora}/agendar/{afiliado}")
public ResponseEntity<String> AgendarSinOrden(@PathVariable("id") String id, @PathVariable("fecha_hora") @DateTimeFormat(pattern = "dd-MM-yyyy HH:m
    try{
        List<Servicio> servicios = servicioRepository.buscarServicios(id);
        if (servicios == null || servicios.isEmpty()) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body("Servicio no encontrado con ID: " + id);
        }
        Servicio servicio = servicios.get(index:0);
        String tipo = servicio.getTipo_servicio().name();
        if (!tipo.equalsIgnoreCase(anotherString:"CONSULTAMEDICOGENERAL") &&
        !tipo.equalsIgnoreCase(anotherString:"CONSULTAURGENCIA")) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(body:"El servicio necesita una orden de servicio");
        }

        List<Prestacion> prestaciones = servicio.getPrestaciones();
        for (Prestacion p : prestaciones) {
            if (fecha_hora.equals(p.getFecha()) &&
                !p.getFinalizado() &&
                (p.getAfiliado() == null || p.getAfiliado().isEmpty())) {

                p.setAfiliado(afiliado);
                servicioRepository.save(servicio);

                return ResponseEntity.ok("Prestación agendada correctamente: " + p.getId_prestacion());
            }
        }

    } catch (Exception e) {
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                        .body("Error al agendar la prestación: " + e.getMessage());
}
    return null;

}
```

- **Agendar con orden**

Para poder agendar un servicio debemos asignarle el número de documento del afiliado a la prestación y el id de la orden, si es un servicio que necesita orden, esto se hace de la siguiente manera:

Verifica el servicio con el id correspondiente luego se verifica que efectivamente se necesite una orden para el servicio dependiendo del tipo. Acto seguido, iteramos sobre las prestaciones de ese servicio y filtramos por la fecha-hora ingresada por parámetro y que no tenga un afiliado ya asignado ni una orden. Apenas encontremos una prestación que cumpla con estos requisitos se asigna el afiliado y la orden a la prestacion y se guarda el servicio con save(servicio). De esta forma ya queda agendada la prestación de ese servicio por parte de un afiliado.

```java
@PostMapping("/{id}/prestaciones/{fecha_hora}/agendar/{afiliado}/{id_Orden}")
public ResponseEntity<String> AgendarConOrden(@PathVariable("id") String id, @PathVariable("fecha_hora") @DateTimeFormat(pattern = "dd-MM-yyyy HH:mm"
    try{
        List<Servicio> servicios = servicioRepository.buscarServicios(id);
        if (servicios == null || servicios.isEmpty()) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body("Servicio no encontrado con ID: " + id);
        }
        Servicio servicio = servicios.get(index:0);
        String tipo = servicio.getTipo_servicio().name();
        if (tipo.equalsIgnoreCase(anotherString:"CONSULTAMEDICOGENERAL") &&
        tipo.equalsIgnoreCase(anotherString:"CONSULTAURGENCIA")) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(body:"El servicio no necesita una orden de servicio");
        }

        List<Prestacion> prestaciones = servicio.getPrestaciones();
        for (Prestacion p : prestaciones) {
            if (fecha_hora.equals(p.getFecha()) &&
                !p.getFinalizado() &&
                (p.getAfiliado() == null || p.getAfiliado().isEmpty()) && (p.getOrden() == null || p.getOrden().isEmpty())) {

                p.setAfiliado(afiliado);
                p.setOrden(id_Orden);
                servicioRepository.save(servicio);

                return ResponseEntity.ok("Prestación agendada correctamente: " + p.getId_prestacion());
            }
        }

    } catch (Exception e) {
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                        .body("Error al agendar la prestación: " + e.getMessage());
}
    return null;

}
```

e. RFC1

Para realizar este requerimiento, se implementaron las clases necesarias en Spring. En primer lugar, se creó la clase CitasDisponibles.java que va a representar la respuesta de la consulta.

```java
package uniandes.edu.co.demo.modelo;

import java.util.Date;

public class CitasDisponibles {

    private Date fecha;
    private String medico;
    private String ips;
    private String servicio;

    public CitasDisponibles(Date fecha, String medico, String ips, String servicio) {
        this.fecha = fecha;
        this.medico = medico;
        this.ips = ips;
        this.servicio = servicio;
    }

    public Date getFecha() {
        return fecha;
    }
    public void setFecha(Date fecha) {
        this.fecha = fecha;
    }
    public String getMedico() {
        return medico;
    }
    public void setMedico(String medico) {
        this.medico = medico;
    }

    public String getIps() {
        return ips;
    }

    public void setIps(String ips) {
        this.ips = ips;
    }
```

```java
    public String getServicio() {
        return servicio;
    }

    public void setServicio(String servicio) {
        this.servicio = servicio;
    }

    @Override
    public String toString() {
        return "CitasDisponibles [fecha=" + fecha + ", medico=" + medico + ", ips=" + ips + ", servicio=" + servicio
                + "]";
    }

}
```

En este caso, la respuesta solo representará el nombre del servicio, del médico, de la IPS y la fecha de las citas disponibles junto con su hora.
Después se creó el repositorio en donde se realiza la consulta para consultar la agenda de disponibilidad que tiene un servicio de salud ingresado por el usuario en las siguientes 4 semanas.

```java
@Repository
public class CitasDisponiblesRepository {

    private final MongoTemplate mongoTemplate;

    public CitasDisponiblesRepository(MongoTemplate mongoTemplate) {
        this.mongoTemplate = mongoTemplate;
    }

    public List<Document> findCitasDisponibles(String id) {


        ObjectId objectId = new ObjectId(id);

        List<Document> pipeline = List.of(
            new Document(key:"$match", new Document(key:"_id", objectId)),

            new Document(key:"$unwind", value:"$prestaciones"),

            new Document(key:"$match", new Document(key:"$expr", new Document(key:"$and", List.of(
                new Document(key:"$gte", List.of(e1:"$prestaciones.fecha", e2:"$$NOW")),
                new Document(key:"$lte", List.of(
                    e1:"$prestaciones.fecha",
                    new Document(key:"$dateAdd", new Document()
                        .append(key:"startDate", value:"$$NOW")
                        .append(key:"unit", value:"day")
                        .append(key:"amount", value:28)
                    )
                ))
            )))),

            new Document(key:"$match", new Document(key:"prestaciones.afiliado", new Document(key:"$eq", value:null))),

            new Document(key:"$lookup", new Document()
                .append(key:"from", value:"medicos")
                .append(key:"let", new Document(key:"medicoIdStr", value:"$prestaciones.medico"))
                .append(key:"pipeline", List.of(
                    new Document(key:"$match", new Document(key:"$expr",
                        new Document(key:"$eq", List.of(
                            e1:"$_id",
                            new Document(key:"$toObjectId", value:"$$medicoIdStr")
                        ))
                    ))
                ))
                .append(key:"as", value:"nomMedico")
            ),

            new Document(key:"$unwind", value:"$nomMedico"),

            new Document(key:"$lookup", new Document()
                .append(key:"from", value:"ips")
                .append(key:"let", new Document(key:"ipsIdStr", value:"$prestaciones.ips"))
                .append(key:"pipeline", List.of(
                    new Document(key:"$match", new Document(key:"$expr",
                        new Document(key:"$eq", List.of(
                            e1:"$_id",
                            new Document(key:"$toObjectId", value:"$$ipsIdStr")
                        ))
                    ))
                ))
                .append(key:"as", value:"nomips")
            ),

            new Document(key:"$unwind", value:"$nomips"),

            new Document(key:"$project", new Document()
                .append(key:"_id", value:0)
                .append(key:"Servicio", value:"$nombre")
                .append(key:"Fecha", value:"$prestaciones.fecha")
                .append(key:"Medico", value:"$nomMedico.nombre")
                .append(key:"IPS", value:"$nomips.nombre")
            )
        );

        return mongoTemplate.getCollection(collectionName:"servicios").aggregate(pipeline).into(new java.util.ArrayList<>());
    }

}
```

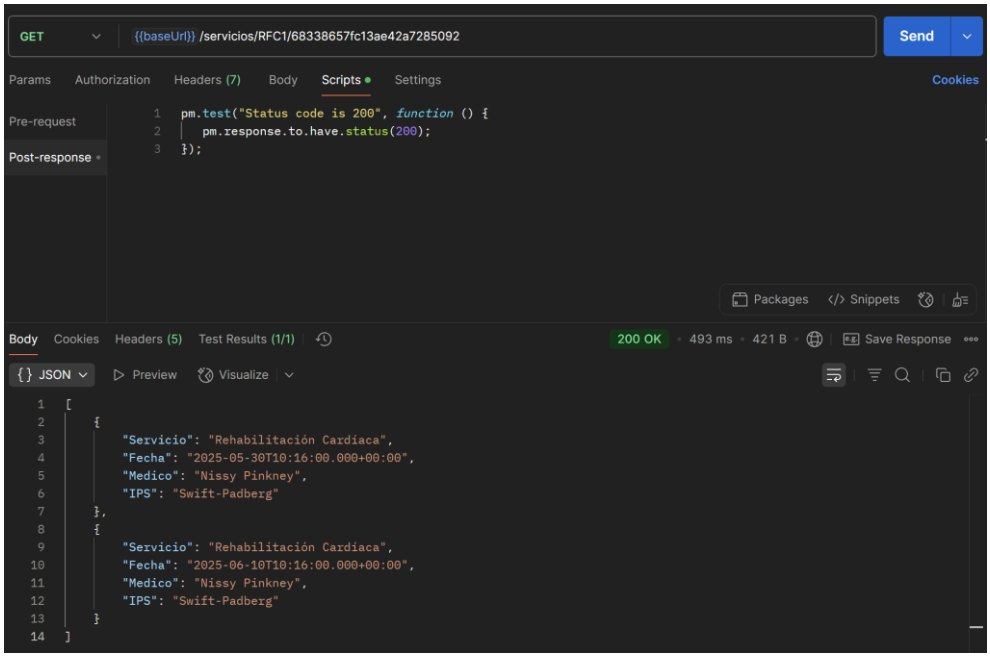
Primero, se obtiene el servicio ingresado por id. Después se descompone la lista de prestaciones para poder analizar una por una con el método $unwind. Y se filtraron las prestaciones por su fecha, comparándolas con la fecha de ese momento con la funcion $$NOW y con la fecha dentro de cuatro semanas al hacer una operación matemática con $dateAdd y sumarle 28 días a la fecha del momento.
En este momento ya se cuenta con las citas en el rango indicado. Después, se verifica que la prestación no tenga un afiliado asociado para saber si la prestación esta disponible.
Ahora bien, para obtener los datos del médico que prestará la cita se utilizó la funcion $lookup para obtener los datos del medico y se guardaron en nomMedico. Como esta función crea un array, para acceder fácilmente a la información, se descompuso la lista. Se realizó lo mismo con la IPS para obtener la información de esta. Esta información se guardó en nomips y se descompuso esta lista para obtener los datos.
Finalmente, se utilizó un $project para mostrar solo la información necesaria para la respuesta, en el que se desactivó el id y se mostraron los campos solicitados.

```
@Autowired
private CitasDisponiblesRepository CitasDisponiblesRepository;

@GetMapping("/RFC1/{id}")
public ResponseEntity<List<Document>> obtenerCitasDisponibles(@PathVariable("id") String id) {
    try {
        List<Document> citas = CitasDisponiblesRepository.findCitasDisponibles(id);
        if (citas != null && !citas.isEmpty()) {
            return ResponseEntity.ok(citas);
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(body:null);
    }
}
```

Después se implementó el endpoint en el controller. En donde se instanció el repositorio implementado en donde se recibe el id del servicio solicitado.

Esta función se probó en Postman y se obtuvieron los siguientes resultados.



f. RFC2

Para realizar este requerimiento, se implementaron las clases necesarias en Spring. En primer lugar, se creó la clase ServiciosSolicitados.java que va a representar la respuesta de la consulta.

```java
package uniandes.edu.co.demo.modelo;

public class ServiciosSolicitados {

    private String servicio;
    private int solicitudes;

    public ServiciosSolicitados(String servicio, int solicitudes) {
        this.servicio = servicio;
        this.solicitudes = solicitudes;
    }

    public String getServicio() {
        return servicio;
    }

    public void setServicio(String servicio) {
        this.servicio = servicio;
    }

    public int getSolicitudes() {
        return solicitudes;
    }

    public void setSolicitudes(int solicitudes) {
        this.solicitudes = solicitudes;
    }

    @Override
    public String toString() {
        return "ServiciosSolicitados [Servicio=" + servicio + ", Solicitudes=" + solicitudes + "]";
    }
}
```

En este caso, la respuesta solo representará el nombre del servicio y la cantidad de solicitudes.

Después se creó el repositorio en donde se realiza la consulta para consultar los 20 servicios más solicitados.

```java
package uniandes.edu.co.demo.repository;

import java.util.Date;
import java.util.List;

import org.bson.Document;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Repository;

@Repository
public class ServiciosSolicitadosRepository {

    private final MongoTemplate mongoTemplate;

    public ServiciosSolicitadosRepository(MongoTemplate mongoTemplate) {
        this.mongoTemplate = mongoTemplate;
    }

    public List<Document> findServiciosSolicitados(Date fechaInicio, Date fechaFin) {

        List<Document> pipeline = List.of(
            new Document(key:"$unwind", value:"$prestaciones"),

            new Document(key:"$match", new Document(key:"$expr",
                new Document(key:"$and", List.of(
                    new Document(key:"$lte", List.of(e1:"$prestaciones.fecha", fechaFin)),
                    new Document(key:"$gte", List.of(e1:"$prestaciones.fecha", fechaInicio))
                ))
            ).append(key:"prestaciones.afiliado", new Document(key:"$ne", value:null))),

            new Document(key:"$group", new Document()
                .append(key:"_id", value:"$nombre")
                .append(key:"solicitudes", new Document(key:"$sum", value:1))
            ),

            new Document(key:"$sort", new Document(key:"solicitudes", -1)),

            new Document(key:"$limit", value:20),

            new Document(key:"$project", new Document()
                .append(key:"_id", value:0)
                .append(key:"Servicio", value:"$_id")
                .append(key:"Solicitudes", value:"$solicitudes")
            )
        );

        return mongoTemplate.getCollection(collectionName:"servicios").aggregate(pipeline).into(new java.util.ArrayList<>());
    }
}
```

Primero, se descompone la lista de prestaciones para poder analizar una por una con el método $unwind. Y se filtraron las prestaciones por su fecha, comparándolas con las fechas ingresadas por parámetro.

En este momento ya se cuenta con las citas en el rango indicado. Después, se verifica que la prestación tenga un afiliado asociado para saber si la prestación fue solicitada.

Ahora bien, se agruparon las prestaciones por el nombre del servicio y se contaron las prestaciones por servicio (cantidad de solicitudes). Con las solicitudes se ordenan los datos de mayor a menor y se obtienen solo los primeros 20 con $limit.

Finalmente, se utilizó un $project para mostrar solo la información necesaria para la respuesta, en el que se desactivó el id y se mostraron los campos solicitados.

```
    @Autowired
    private CitasDisponiblesRepository CitasDisponiblesRepository;

    @GetMapping("/RFC1/{id}")
    public ResponseEntity<List<Document>> obtenerCitasDisponibles(@PathVariable("id") String id) {
        try {
            List<Document> citas = CitasDisponiblesRepository.findCitasDisponibles(id);
            if (citas != null && !citas.isEmpty()) {
                return ResponseEntity.ok(citas);
            } else {
                return new ResponseEntity<>(HttpStatus.NOT_FOUND);
            }
        } catch (Exception e) {
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(body:null);
        }
    }
```

Después se implementó el endpoint en el controller. En donde se instanció el repositorio implementado en donde se recibe las fechas solicitadas y las castea correctamente para que puedan ser utilizadas en la consulta, además de revisra que si haya un rango de que el inicio sea antes del final.

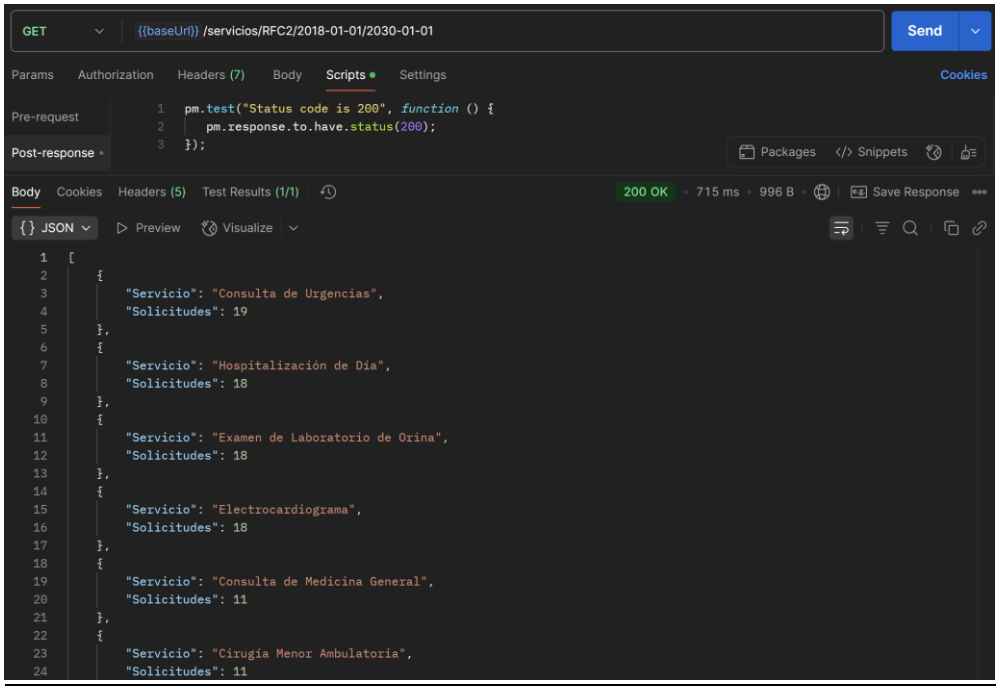
```
@Autowired
private ServiciosSolicitadosRepository serviciosSolicitadosRepository;

@GetMapping("/RFC2/{fechaInicio}/{fechaFin}")
public ResponseEntity<List<Document>> obtenerServiciosSolicitados(@PathVariable("fechaInicio") String fechaInicio,
                                                                  @PathVariable("fechaFin") String fechaFin) {
    SimpleDateFormat sdf = new SimpleDateFormat(pattern:"yyyy-MM-dd");
    Date fechaI = null;
    Date fechaF = null;
    try {
        fechaI = sdf.parse(fechaInicio);
        fechaF = sdf.parse(fechaFin);
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(body:null);
    }

    if (fechaI == null || fechaF == null || fechaI.after(fechaF)) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(body:null);
    }

    try {
        List<Document> servicios = serviciosSolicitadosRepository.findServiciosSolicitados(fechaI, fechaF);
        if (servicios != null && !servicios.isEmpty()) {
            return ResponseEntity.ok(servicios);
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(body:null);
    }
}
```

Esta función se probó en Postman y se obtuvieron los siguientes resultados.



5. Escenarios Prueba

RF4:

```
>_MONGOSH
> use ISIS2304E05202510
< switched to db ISIS2304E05202510
> db.medicos.insertOne({
    "_id": 12345678,
    "nombre": "Médico Inválido",
    "especialidad": "Neurología"
  });
▸ MongoServerError: Document failed validation
ISIS2304E05202510 >
```

RF5:

```
> db.afiliados.insertOne({
    nombre: "María González",
    tipo_documento: "CC",
    numero_documento: 11223344,
    fecha_nacimiento: ISODate("1990-03-15T00:00:00Z"),
    direccion: "Calle 123 #45-67",
    telefono: "3001234567",
    parentesco: null,
    tipo_relacion: null,
    ordenesServicios: []
  })
MongoServerError: Document failed validation
```

RF6:

```
> db.afiliados.updateOne(
    { numero_documento: "368699136" },  // criterio de búsqueda
    {
      $push: {
        ordenesServicios: {
          _id: 2,
          tipo_orden: "Servicio",
          fecha: ISODate("2024-06-01T10:00:00Z"),
          estado: "Pendiente",
          cantidad: 1,
          descripcion: "Terapia respiratoria"
        }
      }
    }
  )
MongoServerError: Document failed validation
```

RF7:


db.servicios.updateOne(
{ _id: ObjectId("68338658fc13ae42a7285095") },
{
$set: {
"prestaciones.0": {
fecha: "mañana",
finalizado: "sí",
medico: null,
ips: 123,
afiliado: 99999,
orden: "primero"
}
}
}
);
MongoServerError: Document failed validation