inside sound
monolithic_backend

Etapas

El proyecto se dividió en las siguientes partes:

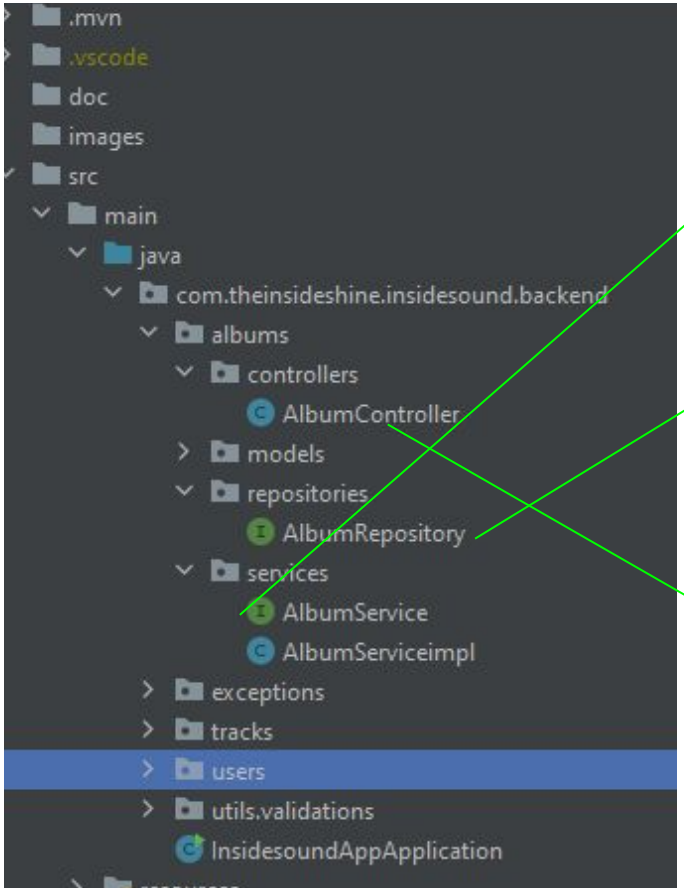
- Evaluación de la interfaz gráfica y ecosistemas de uS
  - Modelo de datos - Modelo de errores
- Evaluación del despliegue en Kubernetes
- Migración de ecosistemas a un servicio monolítico que cumpla con las recomendaciones de diseño

Lineamiento de diseño

- Utilizar arquitectura hexagonal.
- Principios Solid Alid
- Objeto DTO tiene que ser completamente inmutable.
- Evitar malas prácticas como concatenación de Strings con +

Estructura hexagonal

inside sound
monolithic_backend



1. Capa del Dominio(service):

En esta capa, defines tu lógica de negocio y las entidades del dominio.

2. Capa de Puertos(repository):

Define interfaces que serán implementadas por adaptadores. Estas interfaces son puertos que definen cómo se interactúa con el dominio.

3. Adaptadores:

Implementa las interfaces definidas en los puertos. Estos adaptadores interactúan con el dominio y se encargan de la comunicación con el exterior.

4. Configuración y Punto de Entrada:

En esta capa, configuras la inyección de dependencias y defines los controladores o adaptadores de entrada.

inside sound
monolithic_backend

Principios SOLID:

1. **S - Principio de Responsabilidad Única (Single Responsibility Principle):**
   Cada clase debe tener una única razón para cambiar. Asegúrate de que cada clase se ocupe de una sola responsabilidad.
2. **O - Principio de Abierto/Cerrado (Open/Closed Principle):**
   Las clases deben estar abiertas para la extensión pero cerradas para la modificación Esto significa que puedes agregar nuevas funcionalidades sin cambiar el código existente.
3. **L - Principio de Sustitución de Liskov (Liskov Substitution Principle):**
   Las instancias de una clase base deben poder ser sustituidas por instancias de sus clases derivadas sin afectar la funcionalidad correcta del programa.
4. **I - Principio de Segregación de Interfaces (Interface Segregation Principle):**
   Es mejor tener muchas interfaces específicas que una interfaz general. Esto evita qu las clases implementen métodos que no necesitan.
5. **D - Principio de Inversión de Dependencia (Dependency Inversion Principle):**
   Depende de abstracciones, no de implementaciones concretas. Las clases de alto nivel no deben depender de clases de baj vel, ambas deben depender de abstracciones.

Principios ALID (Atomicidad, Localidad, Independencia, Duración):

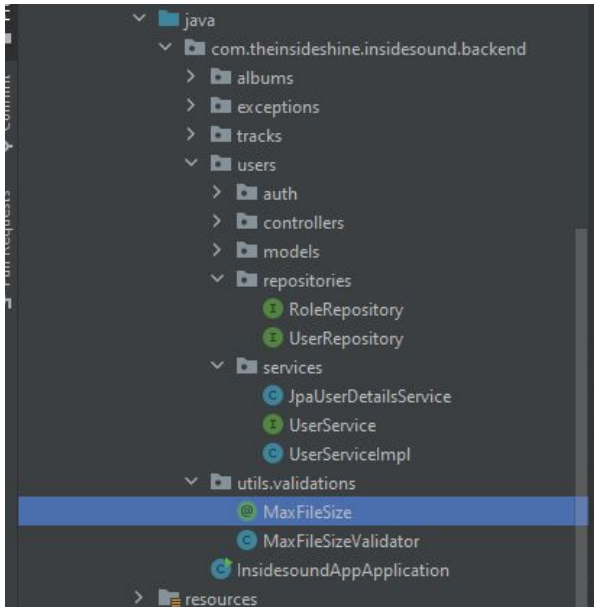
1. **Atomicidad:**
   Asegúrate de que las operaciones sean atómicas, es decir, se ejecuten como una unidad indivisible. Esto es crucial para evitar inconsistencias en el estado de la aplicación.
2. **Localidad:**
   Mantén las operaciones y los datos locales cuando sea posible. Evitar la globalidad en las variables y operaciones contribuye a una mayor cohesión y facilidad de mantenimiento.
3. **Independencia:**
   Las partes del sistema deben ser independientes entre sí. La modificación de una parte no debería afectar a otras, siempre que la interfaz entre ellas se mantenga constante.
4. **Duración:**
   Controla la duración de las operaciones para minimizar el tiempo de bloqueo y mejorar la eficiencia. Esto es especialmente relevante en operaciones de concurrencia.

inside sound
monolithic_backend

Liskov

El principio de Liskov Substitution establece que los objetos de una clase base deben poder ser sustituidos por objetos de una clase derivada sin afectar la corrección del programa. Dado que estás trabajando con interfaces y validadores de Spring, la idea de sustitución es más relevante en el **contexto de herencia de clases.**

Isp



puedes decir que si tienes una sola implementación de todos los métodos de una interfaz, estás cumpliendo con el principio de segregación de interfaces (ISP). Este principio sugiere que una clase no debería verse obligada a implementar métodos que no necesita. Si tienes una única implementación de la interfaz y todos los métodos de esa interfaz son relevantes para esa implementación, entonces estás siguiendo el principio de ISP.

D- Inversión de dependencias

inside sound
monolithic_backend

Es una práctica común en el diseño de software utilizar interfaces en la capa de servicios. Este enfoque permite la flexibilidad para tener múltiples implementaciones del servicio y facilita la sustitución de una implementación por otra sin afectar el resto del sistema. Al utilizar interfaces en la capa de servicios, estás aplicando el principio de inversión de dependencias (Dependency Inversion Principle - DIP) de los principios SOLID.

La inversión de dependencias, como se describe en el principio de inversión de dependencias (Dependency Inversion Principle - DIP), se refiere a la idea de que las clases de alto nivel no deben depender de clases de bajo nivel, sino que ambas deben depender de abstracciones. Además, las abstracciones no deben depender de los detalles, sino que los detalles deben depender de las abstracciones.

En el contexto de interfaces y clases de implementación, la inversión de dependencias implica que las clases de alto nivel (por ejemplo, las clases en capas superiores de tu aplicación, como las capas de presentación o lógica de negocio) deben depender de abstracciones (interfaces), en lugar de depender directamente de las implementaciones concretas.

Sistema de excepciones

inside sound
monolithic_backend

handlerGlobal

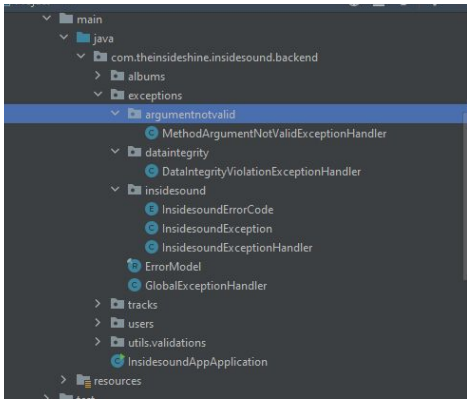
ErrorModel

Handler

Valid

Data

InsideSound



Inside sound Excepción

Inside sound error code

Sistema de excepciones

inside sound
monolithic_backend

Validations

Service

Dto

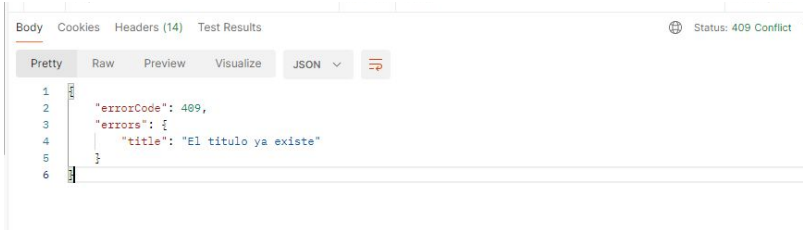
Entity

Inside Sound Excepción

Argument not valid

Data integrity Violation

Inside sound error code

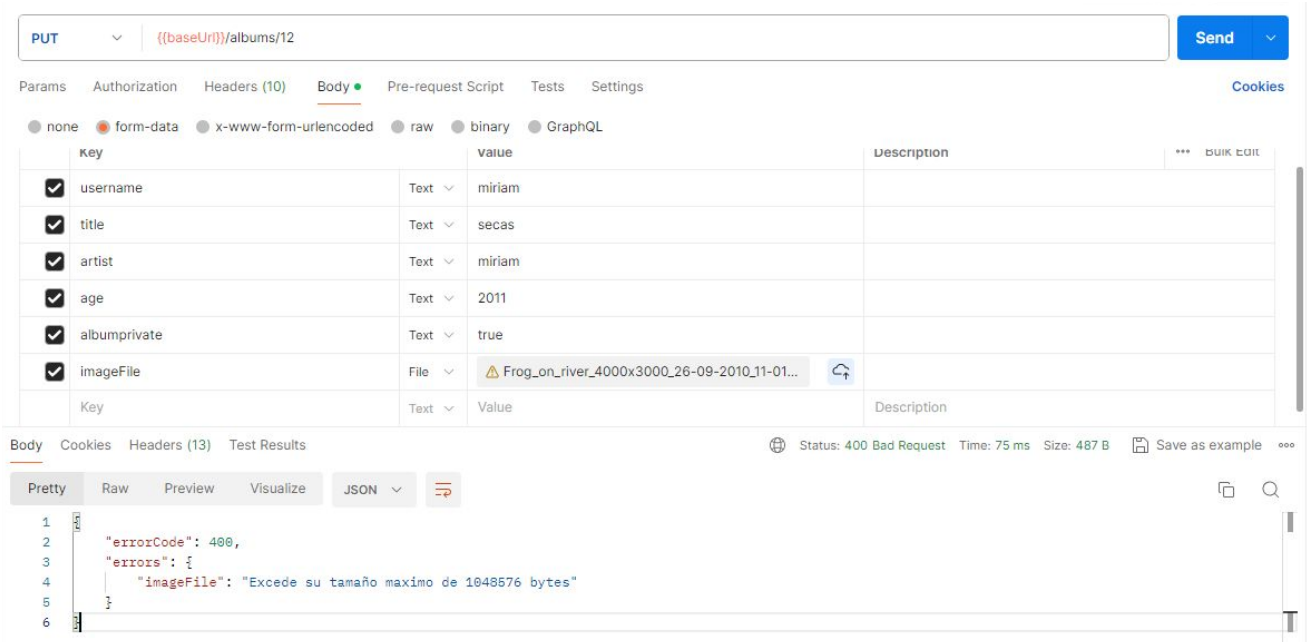


Application.ymal

Sistema de excepciones

inside sound
monolithic_backend

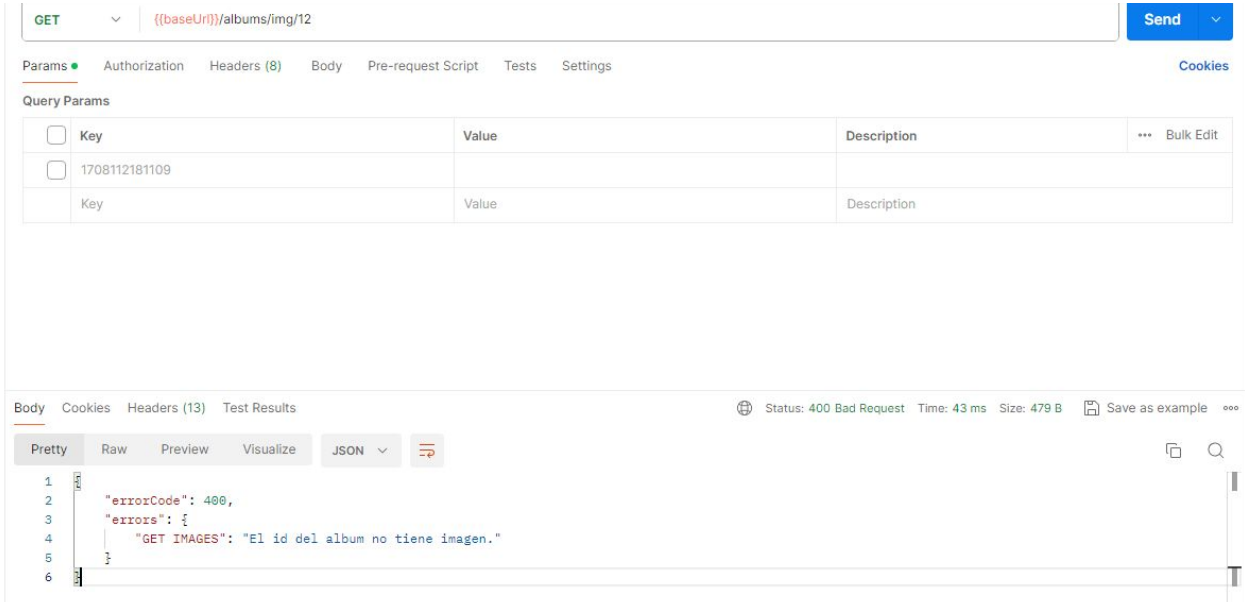
Validations



Sistema de excepciones

inside sound
monolithic_backend

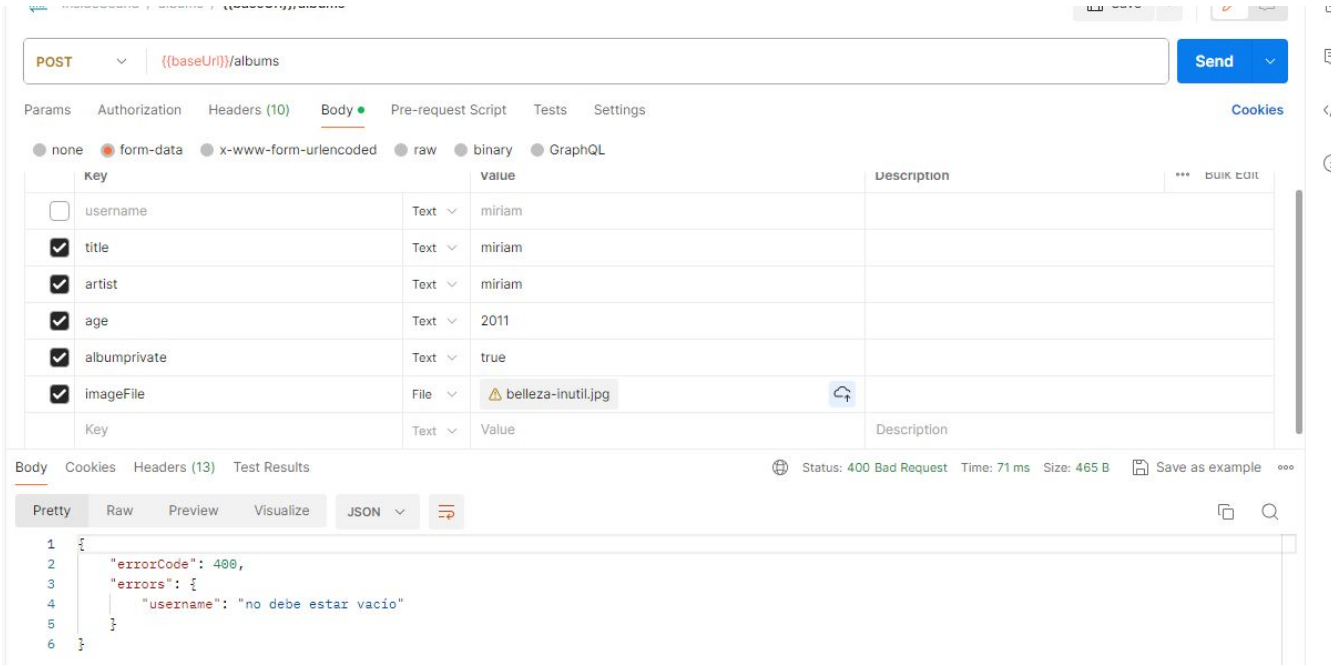
Service



Sistema de excepciones

inside sound
monolithic_backend

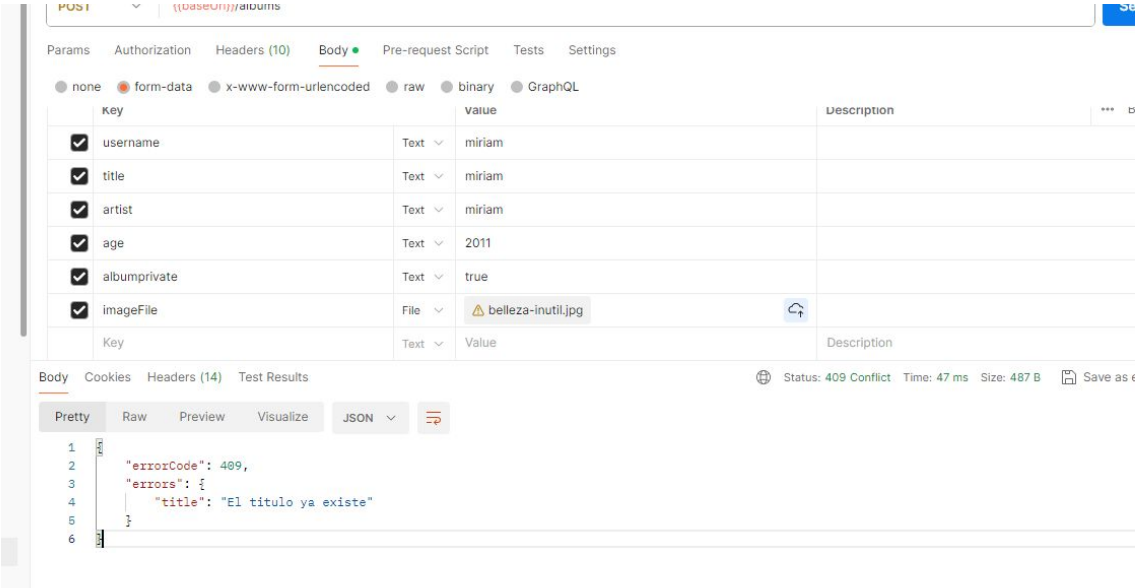
Dto



Sistema de excepciones

inside sound
monolithic_backend

Entity



Sistema de excepciones

inside sound
monolithic_backend

useUsers.js | useAlbums.js | useTracks.js | RegisterAlbumPage.jsx | AlbumForm.jsx | TrackFor

src > hooks > useAlbums.js > useAlbums

```
9   export const useAlbums = () => {
19      const handlerAddAlbum=async(formData)=>{
38                      'El Album ha sido creado con exito!' :
39                      'El Album ha sido actualizado con exito!',
40                  'success'
41              );
42              handlerCloseAlbum(); //Borra errores
43              navigate('/albums');
44              } catch (error) {
45
46                  if (error.response && error.response.status == 400) {
47
48                      dispatch(loadingAlbumError(error.response.data.errors));
49
50                  } else if (error.response && error.response.status == 409){
51
52                      console.log(error.response.data.errors);
53                      dispatch(loadingAlbumError(error.response.data.errors));
54                  } else if (error.response && error.response.status == 500){
55                      console.log('error: ',error);
56                  }else {
57                      console.log(error);
58                      throw error;
59                  }
60              }
61      }
```