Nombres: Carlos Rodríguez

Alberto Zapata

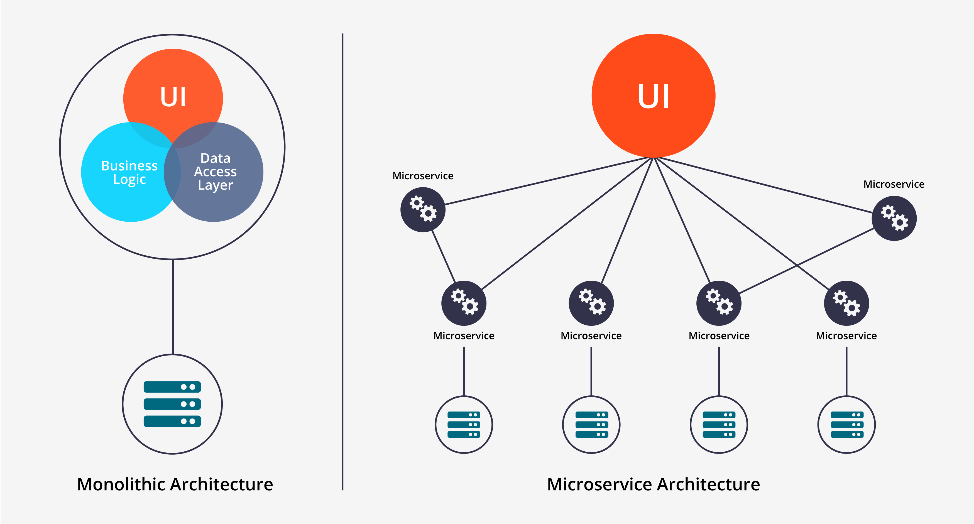
Reynaldo Blanco

Juan Villanueva

Taller de modelado y validación de arquitectura

1. **Patrón nuclear de la implementación:** **Microservicios**

Como patrón nuclear de la arquitectura elegimos microservicios. Este nos permitirá un desacoplamiento de los sistemas que interactúan reduciendo la complejidad e incrementando la portabilidad y la escalabilidad del sistema. Esto es muy importante para nuestro diseño ya que nuestra idea requiere exponer una serie de servicios REST para la gestión de los convenios de pago que este disponga dentro de nuestra plataforma.



Dentro de las ventajas por las cuales lo seleccionamos como patrón nuclear están:

* Escalabilidad
* Funcionalidad modular, módulos independientes.
* Libertad del desarrollador de desarrollar y desplegar servicios de forma independiente
* Uso de contenedores permitiendo el despliegue y el desarrollo de la aplicación
* rápidamente

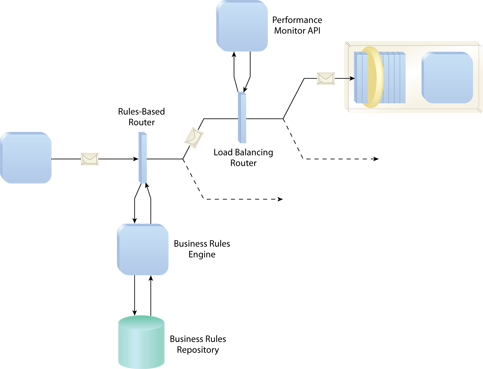
De la misma forma sabemos que encontramos desventajas en este enfoque como lo es la gestión de estos servicios y la complejidad que lleva realizar pruebas end to end.

1. Otros patrones implementados en la solución

* Cliente – servidor: La arquitectura cliente-servidor es un patrón de arquitectura de software en el que las tareas se reparten entre los proveedores de recursos o servicios (servidores), y los que solicitan estos servicios o recursos (clientes).



* Orquestación de servicios: Se realiza una orquestación de servicios (composición de servicios) en el cual un servicio es el encargado de realizar de la interacción entre los diferentes servicios creados para cumplir con la funcionalidad.
* MVC: Patrón de arquitectura de software que utilizando 3 componentes separados (Vista, Modelo y Controlador) separa la lógica de la aplicación de la lógica de la vista en una aplicación donde el controlador permite la comunicación entre la vista y el modelo y comunica entre ellos los cambios de estado de cada uno.
* Dispatcher: Un dispatcher es el responsable de la ejecución de los servicios, controlando su ejecución de acuerdo con los parámetros enviados por el routing controller. Un dispatcher se puede encapsular dentro de un controlador (en este caso lo hemos implementado así) o puede ser un componente independiente que trabaja en coordinación con el controlador.
* Intermediate routing: Es un patrón que se usa para el enrutamiento de la petición al servicio correcto de acuerdo con los parámetros que se le ingresan.



* Registry service: Servicio para la ejecución de los pagos de acuerdo con el número de factura y el convenio al que pertenece la misma.

**TradeOffs**

Al tener una arquitectura orientada a microservicios, garantizamos el desacoplamiento de la capa lógica, lo cual nos garantiza Escalabilidad en cuanto a la evolución de funcionalidades. Esto beneficia otros atributos de calidad sobre el tiempo, como la eficiencia, disponibilidad y rendimiento.

Si bien desagregar la capa lógica permite la reutilización de componentes y funcionalidades al ser expuestas a través de un protocolo de comunicación, Por otro lado, podemos estar expuestos a tiempos de latencia elevados al momento de realizar y consumir servicios.

La arquitectura orientada a microservicios implementa la federación, lo que nos asegura que cada uno de sus componentes hace parte de un todo y a su vez se regula a sí mismo. Esto por otro lado agrega cierto grado de complejidad al momento de realizar nuevas funcionalidades y/o despliegues, pues todo los interesados o responsables de los servicios deben mantenerse en contacto para evitar desfases en cuanto a la integración de dichos componentes.

1. Diagrama de componentes

A close up of a piece of paper

Description automatically generated

1. Herramientas utilizadas

* Lenguaje de programación: Java
* Servidor de aplicaciones: Apache Tomcat.
* Frameworks: Spring boot, JAX-RS, Jersey.
* Base de datos: MySQL 8.
* Entorno de Desarrollo: Eclipse STS
* Herramienta para pruebas de servicios: SOAP UI

#%RAML 1.0

title: PagoFacturas

version: 1.0

baseUri: api/resources

types:

RESTData:

properties:

metodo: string

funcion: string

recurso: string

pathParamMapping: string

paramsMapping: string

headers: string

payloadMapping: string

SOAPData:

properties:

soapAction: string

xsltDefinition: string

funcion: string

responseElement: string

Convenio:

properties:

id: number

idConvenio: number

nombre: string

tipoServicio: string

host: string

puerto: number

urlServicio: string

tipo: string

soapData: SOAPData[]

restData: RESTData[]

Pago:

properties:

numeroFactura: number

numeroConvenio: number

numeroIdCliente: string

tipoIdCliente: string

totalPago: number

/api/v1/convenio:

post:

description: Permite crear un convenio a partir del objeto dado

body:

application/json:

type: Convenio

/{id}:

delete:

description: Permite borrar un convenio registrado

responses:

200:

description: Borra un convenio

patch:

description: Permite modificar la información almacenada de un convenio

body:

application/json:

type: Convenio

get:

description: Permite devolver el convenio asociado a un número de convenio dado

responses:

200:

body:

application/json:

type: Convenio

/pago:

post:

description: Permite efectuar un pago

body:

application/json:

type: Pago