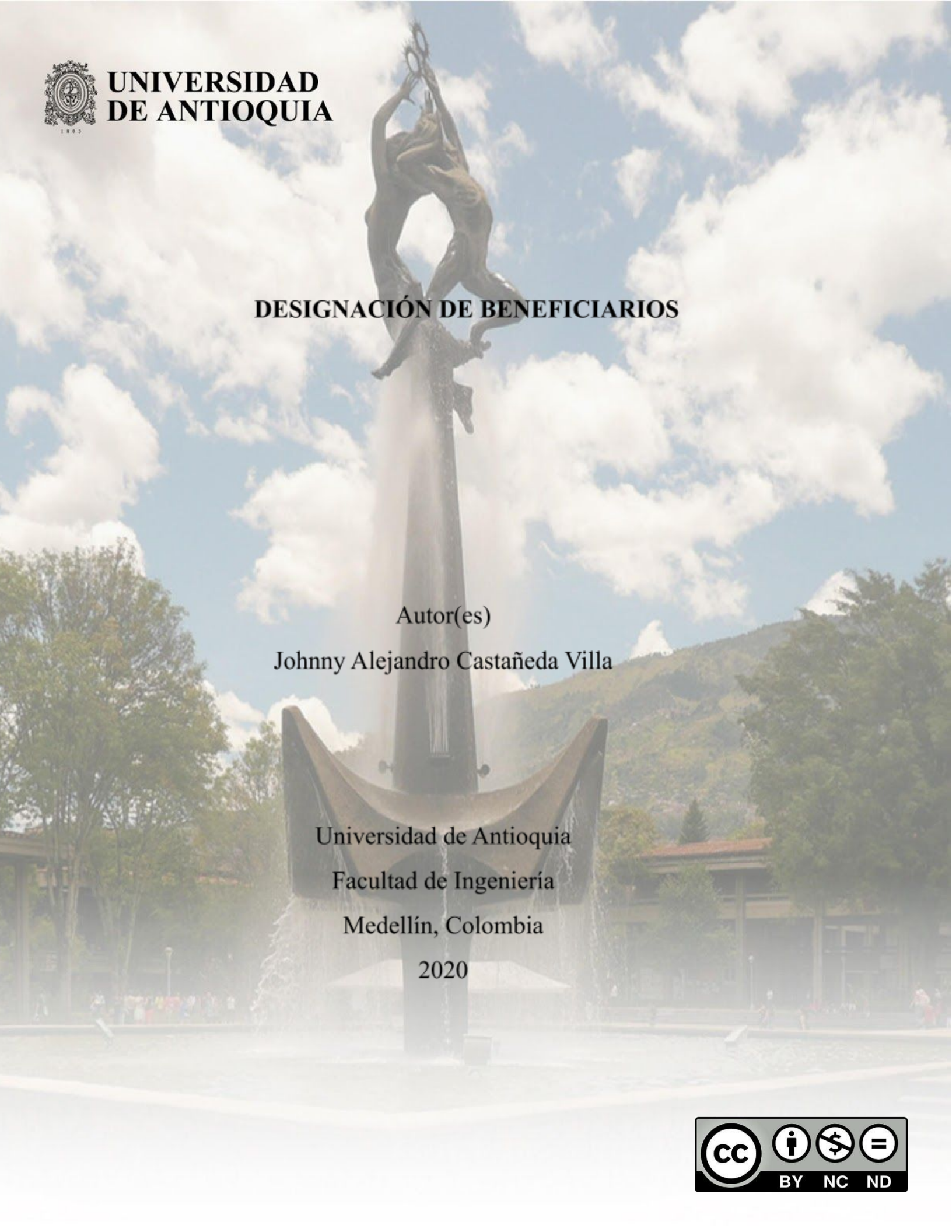


UNIVERSIDAD DE ANTIOQUIA

DESIGNACIÓN DE BENEFICIARIOS

Autor(es)

Johnny Alejandro Castañeda Villa

Universidad de Antioquia

Facultad de Ingeniería

Medellín, Colombia

2020

Designación de Beneficiarios

Johnny Alejandro Castañeda Villa

Informe de práctica
como requisito para optar al título de:
Ingeniero de Sistemas.

Asesores

Gustavo Andres Marin Lopera - Ingeniero de Sistemas
Sebastián Ochoa Isaza - Ingeniero de Sistemas

Universidad de Antioquia
Facultad de Ingeniería.
Medellín, Colombia
2020

Designación de Beneficiarios

Resumen

Antes de la realización de este proyecto, no se tenía desarrollado ningún microservicio que permitiera por medio del número de identificación de un asegurado, obtener las pólizas adquiridas con la compañía y todos sus beneficiarios. Por medio de este trabajo que se realizó en el periodo de prácticas académicas se logró desarrollar un microservicio que puede ser consumido por cualquier aplicativo como una unidad independiente, autónoma, modular, auto-contenida y con una gestión centralizada en el lenguaje de programación SCALA haciendo uso del framework de desarrollo Play.

Introducción

El sector asegurador juega un papel muy importante en nuestra economía, ya que a medida que el país avanza en su proceso de desarrollo, los mayores niveles de inversión y de producto involucran mayores riesgos. Sura como compañía de seguros proporciona seguridad a las personas y a las empresas en la medida en que compensa o reduce las pérdidas debidas a sucesos imprevistos. La importancia como sector canalizador de ahorro e inversión y su especial característica de relacionarse con tomadores, asegurados y beneficiarios mediante contratos de adhesión, determinan que el acceso, ejercicio y control de las actividades sean cada vez más flexibles.

Sura como entidad aseguradora establece una serie de condiciones para sus asegurados que le permiten ofrecer su servicio de manera transparente y confiable. Entre los derechos indelegables del asegurado a una póliza se encuentra hacer y revocar la designación de sus beneficiarios. Es decir, definir quiénes son los acreedores (mediante el pago de una prima) de la prestación que satisface la empresa en el caso de que se produzca un hecho indemnizable previsto al contrato suscrito.

Actualmente, este proceso de designación y actualización de beneficiarios por parte del asegurado se realiza por medio de un formato físico, que permite notificar a la aseguradora información relevante para sus procesos internos. Surge

entonces la necesidad de un componente tecnológico que facilite esta tarea y que le brinde a todos los clientes un canal de autogestión de beneficiarios. Este componente será un microservicio ejecutándose en su propio proceso con una interfaz de programación de aplicaciones (API) sobre protocolo de transferencia de hipertexto (HTTP) como mecanismos ligeros de comunicación. Este servicio se construirá alrededor de las capacidades del negocio y con independencia de despliegue e implementación totalmente automatizada. Reduciendo la operatividad de los asesores y los reprocesos, lo cual se va a traducir en mayor productividad por parte de todos los colaboradores de la compañía que participan en el proceso de recolección y gestión de los documentos que se usan actualmente en dicha actividad; también se va a mejorar la experiencia de los usuarios, debido a que el servicio se va a llevar a cabo en un tiempo más corto y de forma más flexible determinando así la capacidad para realizar o cumplir adecuadamente el proceso; y la calidad de los datos también va a tener una mejora sustancial lo cual es sumamente importante ya que estos constituyen un recurso muy importante para la organización al ser utilizados fundamentalmente para la toma de decisiones y realización de procesos. La mala calidad de los mismos influye de manera significativa y profunda en la efectividad y eficiencia de la empresa, así como en todo el negocio.

Objetivos

Objetivo general

Permitir a los clientes de la compañía adquirir un canal de autogestión de los beneficiarios asociados a su póliza mejorando la experiencia en el uso de los servicios de la compañía y disminuyendo la operatividad de los asesores.

Objetivos específicos

- Realizar análisis de requerimientos y viabilidad del proyecto
- Realizar definición de actividades y planes de ejecución
- Desarrollar microservicio web que permita a los asegurados de la compañía gestionar los beneficiarios asociados a su póliza.
- Revisar correcta funcionalidad, integración y superación de pruebas

Marco teórico

- Seguro

Es un contrato mediante el cual el TOMADOR se compromete a pagar una prima y el ASEGURADOR (SURA) a indemnizar al ASEGURADO, en caso de siniestro.

- Tomador

Es la persona natural o jurídica que solicita el seguro y con la que la Compañía se entiende en el manejo del seguro: expedición, modificación, renovación, cancelación, cobros y otros eventos. No necesariamente es quien recibe la indemnización en caso de siniestro.

- Siniestro

Acontecimiento que, por originar daños concretos previstos en la póliza, motiva la aparición del principio indemnizatorio, obligando a la entidad aseguradora SURA a satisfacer, total o parcialmente, al asegurado o a sus beneficiarios, el capital garantizado en el contrato.

- Asegurado

Es quien recibe la indemnización en caso de siniestro, si no se ha designado a otro beneficiario.

- Beneficiario

Es la persona natural o jurídica que recibe la indemnización en caso de un siniestro.

- Scala

Es un lenguaje de programación multi-paradigma diseñado para expresar patrones comunes de programación en forma concisa, elegante y con tipos seguros. Integra sutilmente características de lenguajes funcionales y orientados a objetos. La implementación actual corre en la máquina virtual de Java y es compatible con las aplicaciones Java existentes Es un lenguaje de programación orientado a objetos puro, en el sentido de que cada valor es un objeto. El tipo y comportamiento de los objetos se describe por medio de clases y traits. La abstracción de clases se realiza extendiendo otras clases y usando un mecanismo

de composición basado en mixins como un reemplazo limpio de la herencia múltiple. Scala también posee características propias de los lenguajes funcionales. En Scala las funciones son valores de primera clase, soportando funciones anónimas, orden superior, funciones anidadas y currificación. Scala viene integrado de fábrica con la técnica de pattern matching para modelar tipos algebraicos usados en muchos lenguajes funcionales.Scala está equipado con un sistema de tipos expresivo que refuerza a que las abstracciones de tipos se usen en forma coherente y segura.

- Framework Play

Play es un marco web moderno y refrescante para scala y java, sigue el patrón arquitectónico Model-View-Controller (MVC). Está inspirado en Ruby on Rails (RoR) y prefiere la convención sobre el enfoque de configuración en la creación de aplicaciones web. Play está diseñado desde cero hasta un alto rendimiento con un consumo mínimo de recursos (CPU, memoria, hilos), siendo su naturaleza central asincrónica y sin bloqueo. Por lo tanto, es más fácil escalar aplicaciones Play tanto horizontalmente (agregando instancias de aplicaciones paralelas debido a que no tiene estado) como verticalmente (agregando más CPU y memoria), proporcionando un conjunto de herramientas muy robusto para escalar un proyecto.

Metodología

La metodología Scrum para el desarrollo ágil de software es un marco de trabajo que utiliza un enfoque incremental que se fundamenta en transparencia, control y adaptación; la transparencia, garantizó la visibilidad en el proceso de todas aquellas cosas que podrían afectar el resultado de los esfuerzos; el control, nos ayudó a detectar cambios o hechos inesperados en el proceso; y la adaptación, nos permitió realizar los ajustes pertinentes para reducir el impacto de estos cambios.

El ciclo de vida de este marco de trabajo estuvo compuesto de cuatro fases: planeación, diseño, desarrollo y entrega. En la planeación se estableció la visión, se fijaron las expectativas y se aseguró el financiamiento. En la puesta en escena se identificaron más requerimientos y se priorizaron para la primera iteración. En la implementación se desarrolló el microservicio, y en la entrega se hizo el despliegue operativo.

La entrega del producto se hizo en sprints (iteraciones); cada iteración añadió nuevas funcionalidades o mejoras al servicio requerido, y se compuso de ceremonias o reuniones que ayudaron a cumplir los objetivos, las cuales fueron:

planeación del Sprint, Daily Scrum, trabajo de desarrollo, revisión del Sprint y retrospectiva del Sprint.

En la reunión de planeación del Sprint se definió plan de trabajo: que se iba a entregar y cómo se lograría. Es decir, el diseño de la solución y la estimación de cantidad de trabajo. La Daily es un evento de quince minutos, que se realizó cada día con el fin de explicar lo que se había alcanzado desde la última reunión; lo que se haría antes de la siguiente; y los obstáculos que se habían presentado. La Revisión del Sprint ocurrió al final del Sprint y su duración era de 2 horas. En esta etapa: se revisó lo que se hizo, se identificó lo que no se hizo y se discutió acerca del Backlog; allí se manifestaron los problemas que se encontraban y la manera en que fueron resueltos, también se mostraban los avances y funcionamiento. Esta reunión fue de gran importancia para los siguientes Sprints.

La retrospectiva del Sprint se realizó en una reunión de 2 horas en la que se analizó cómo fue la comunicación, el proceso y las herramientas; que estuvo bien, que no, y se creó un plan de mejoras para el siguiente Sprint.

Algunos objetos importantes de esta metodología fueron el Product Backlog y el Sprint Backlog. El Product Backlog es una lista −ordenada por valor, riesgo, prioridad y necesidad− de los requerimientos que el P.O definió, actualizó y ordenó. El Sprint Backlog es un subconjunto de ítems del Product Backlog y el plan que se estableció para realizar el Incremento del desarrollo. Debido a que el Product backlog estaba organizado por prioridad, el Sprint backlog era construido con los requerimientos más prioritarios del Product backlog y con aquellos que quedaron por resolver en el Sprint anterior. Requerimientos adicionales fueron incluidos en el Product backlog y desarrollados en el siguiente Sprint, si su prioridad así lo indicaba.

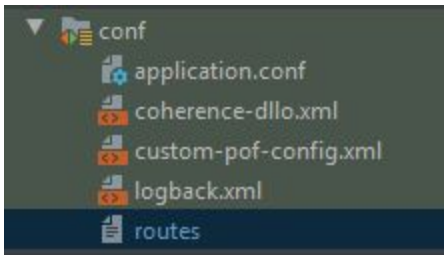
Resultados y análisis

Los resultados del trabajo realizado en las prácticas académicas fueron satisfactorios, puesto que se logró por medio de distintas reuniones una mejor planeación del proyecto. Se definieron planes de trabajo que permitieron crear y probar un servicio REST que posibilita a un asegurado identificar sus pólizas y beneficiarios, sin embargo, el microservicio realizado cuenta con oportunidades de mejora en donde se pueden adicionar dos servicios más que permitan una mejor validación y persistencia de cambios en la base de datos de la organización, que por falta de definición no pudieron ser implementados en este alcance. Además del desarrollo del microservicio se trabajó la integración

continua a través de Jenkins y el aprovisionamiento de la API en la que será montado el microservicio, la cual hará uso de la infraestructura ofrecida por el proveedor de nube Azure y que permitirá que los clientes de la compañía tengan acceso a este nuevo servicio en donde autogestionarán sus beneficiarios.

Implementación del microservicio

conf/routes es el archivo de configuración utilizado por el enrutador. Este archivo enumera todas las rutas que necesita la aplicación. Cada ruta consta de un método HTTP y un patrón URI, ambos asociados con una llamada a un Actiongenerador.



Para el proyecto beneficiarios se tienen definidas las siguientes rutas.

```
# Routes
# This file defines all application routes (Higher priority routes first)
# ~~~~

# An example controller showing a sample home page
POST    /secureCommand/:nombrecomando   co.com.sura.apivida.controladores.ControladorComandos.command(nombrecomando)
GET     /last/version                   co.com.sura.apivida.controladores.ControladorConsultas.index
```

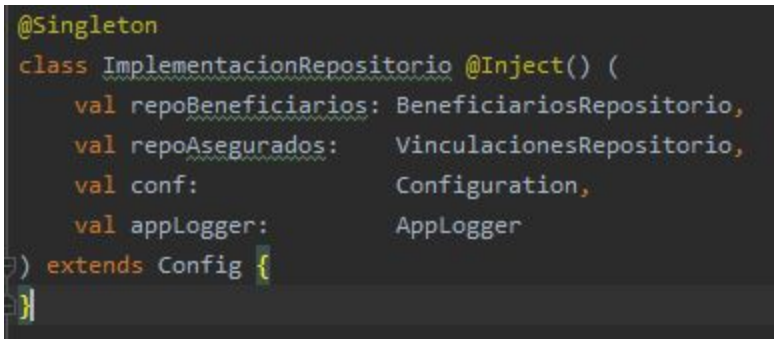
En donde la ruta asociada al método GET corresponde a la versión y la ruta asociada al método POST permite la implementación del patrón de diseño Command, ya que permite acceder al controlador de comandos.
En el proyecto se usará el anterior patrón, por lo cual todos métodos (GET, PUT, DELETE) serán invocados bajo el método POST del endpoint command.

Controladores

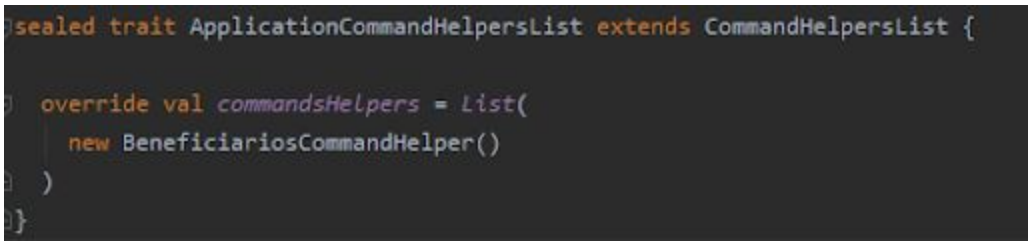
```
@Singleton
class ControladorComandos @Inject() ( override val confComando: ImplementacionRepositorio ) extends ControllerBaseSimple with ApplicationCommandHelpersList {
  implicit val executionContext = co.com.sura.apivida.appExecutionContext
}
```

La clase ControladorComandos recibe como parámetro una configuración que está dada por los repositorios de datos y las clases predefinidas Configuration y AppLogger. Como este controlador requiere el componente ImplementacionRepositorio como dependencia, se declara la acción con la anotación @inject. La cual es usada en el constructor, y debe estar después del nombre de la clase pero antes de los parámetros. La clase cuenta con la anotación @Singleton, para asegurar que si se llega a inyectar el controlador en otro lugar, no se obtendrá una nueva instancia. También, es importante mencionar que el parámetro confComando se define como override, para poder sobreescribir la variable con igual nombre de la clase heredada ControllerBaseSimple.

La variable confComando tendrá como valor la instancia de ImplementaciónRepositorio.



Finalmente, la clase hace uso de la interfaz ApplicationCommandHelpersList, la cual hereda de la clase CommandHelpersList. La interfaz sobreescribe la variable commandHelpers, y como valor le asigna una lista con las instancias de los CommandHelpers necesarios, que para el proyecto solo es uno llamado BeneficiariosCommandHelper.



Repositorios

- BeneficiariosRepositorio

Esta clase hace uso de la inyección de dependencias para obtener una instancia de DatabaseConfigProvider, para la base de datos nombrada default en el

archivo "application.conf". Esto lo hace internamente el framework Play a través de Slick.

Slick es una consulta moderna de base de datos y biblioteca de acceso para Scala. Le permite trabajar con datos almacenados casi como si estuviera usando colecciones de Scala y al mismo tiempo le da control total sobre cuándo ocurre un acceso a la base de datos y qué datos se transfieren. Puede escribir las consultas de su base de datos en Scala en lugar de SQL, aprovechando así la comprobación estática, la seguridad en tiempo de compilación y la composición de Scala. Slick presenta un compilador de consultas extensible que puede generar código para diferentes backends.

```
class BeneficiariosRepositorio @Inject() ( protected val dbConfigProvider: DatabaseConfigProvider ) {
```

Luego por medio del DatabaseConfigProvider accedemos a la API jdbc de Play y obtenemos el JdbcProfile, que a su vez nos va a servir para crear una nueva instancia de la base de datos. Con las interfaces y clases estándar de la API JDBC, podemos escribir aplicaciones que conecten con bases de datos, envíen consultas escritas en el lenguaje de consulta estructurada (SQL) y procesar los resultados.

```
val dbConfig = dbConfigProvider.get[JdbcProfile]
val db = dbConfig.db
```

Después en la variable implícita rsBeneficiarios se guarda el GetResult[Beneficiario], que se encarga de convertir cada registro de la consulta sql en un objeto de tipo Beneficiario. Finalmente, esta clase define el método obtenerBeneficiariosPorPolizaCertificado, que recibe como parámetros una lista con las pólizas-certificados del asegurado e implícitamente un contexto de ejecución. Internamente la función, convierte la lista a
tuplas que luego inserta en el query de consulta.

El query relaciona por medio del DNI las tablas BENEFICIARIOS y PERSONAS, en donde selecciona los campos PRIMER_NOMBRE, SEGUNDO_NOMBRE, PRIMER_APELLIDO, SEGUNDO_APELLIDO, TIPO_IDENTIFICACION, DNI, DSE_,MAIL, DSCELULAR de la tabla PERSONAS y los campos NCERTIFICADO, CDPARENTESCO, PODISTRIBUCION, CDTIPO_BENEFI y NPOLIZA de la tabla BENEFICIARIOS. Finalmente, filtra los resultados solo teniendo en cuenta las tuplas póliza-certificado recibidas en la función.

Ya teniendo la consulta definida, se accede a la base de datos, se consultan todos los beneficiarios con la instruccion db.run(consulta.as[Beneficiario]) y con .map, se genera una lista.

```
db.run( consulta.as[Beneficiario] ).map {
  _.toList
}
```

Nota: Para lograr convertir un objeto de scala en json y viceversa, es necesario declarar una objeto implícito tipo format[tipo], donde el "tipo" corresponde al objeto a convertir.
El Format[T] tiene implícitamente la combinación de los métodos Reads and Writes.

- VinculacionesRepositorio

La clase es implementada de manera similar a BeneficiariosRepositorio, las únicas diferencias que vale la pena mencionar son que el GetResult de esta clase construye a partir de cada registro de la consulta objetos de tipo Póliza y la construcción del query de consulta en base de datos.

El query relaciona las tablas PERSONAS_PROD, CUERPOLIZA, PERSONAS, COBERTURAS y DIC_ALIAS_RS.
Las relaciones se hacen de la siguiente manera:

- PERSONAS_PROD y CUERPOLIZA por el campo NPOLIZA, con FECSUS is null para garantizar polizas vigentes.
- PERSONAS_PROD y PERSONAS por DNI
- PERSONAS_PROD y COBERTURAS por los campos NPOLIZA y NCERTIFICADO, con CDGARANTIA igual VID(Poliza de vida) y CDESTADO igual a A (Activa)
- DIC_ALIAS_RS y CUERPOLIZA por los campos CDRAMO y CDSUBRAMO.

De la tabla PERSONAS, selecciona el PRIMER NOMBRE, SEGUNDO_NOMBRE, PRIMER_APELLIDO, SEGUNDO_APELLIDO, DNI, TIPO_IDENTIFICACION, EMAIL y CELULAR; de la tabla PERSONAS_PROD se obtienen POLIZA y NUMERO_CERTIFICADO, de la tabla COBERTURAS se selecciona el VALOR_ACODISEGURADO, CDGARANTIA y CDSUBGARANTIA, finalmente de la tabla DIC_aLIAS_RS se obtiene CDSUBRAMO y NOMBRE_PLAN. Toda esta información la trae solo para las pólizas cuyo CDRAMO sea 081,083 o 181, tambien se filtra por CODIGO teniendo en cuenta solo el 120 y el 020.

Resumiendo, el query nos trae los datos personales del asegurado y las pólizas de vida a las que está asociado.

BeneficiariosCommandHelper

```
class BeneficiariosCommandHelper extends CommandHelper {

  import play.api.libs.functional.syntax._
  import play.api.libs.json.Reads._
  import play.api.libs.json._

  override val name: String = "beneficiarios"

  override val commandReads: Reads[Command] = (
    ( JsPath \ "tipoDocumento" ).read[String] and
    ( JsPath \ "numeroDocumento" ).read[String]
  )( BeneficiariosCommand.apply _ )

  override val jsonPathsNames: Map[String, String] = Map(
    "/tipoDocumento" -> "tipoDocumento",
    "/numeroDocumento" -> "numeroDocumento"
  )
}
```

La clase BeneficiariosCommandHelper hereda de la clase generica CommandHelper, la cual tiene funciones de uso común para cada uno de los comandos que se definan en el proyecto. Aquí, se define el comando, asignándole el nombre "beneficiarios" en la variable name, especificando también los parámetros de entrada necesitados para la ejecución del comando.

JsPath es un componente básico para crear Reads/ Writes. JsPath representa la ubicación de los datos en una estructura JsValue y con .read[String], se leen los parametros que luego van a ser usados por el constructor de la clase BeneficiariosCommand. Finalmente, en la variable jsonPathsNames se almacena un mapa cuya clave es "/tipoDocumento" y valor "/numeroDocumento".

BeneficiariosCommand

```scala
case class BeneficiariosCommand( tipoDocumento: String, numeroDocumento: String ) extends SyncCommand {

  override def execute( request: Option[Request[AnyContent]] = None, authInfo: Option[Authorized] = None )
                       ( implicit executor: ExecutionContext ): Reader[Config, Future[( Mensaje, List[Event] )]] =
    Reader {
      case configuration: ImplementacionRepositorio =>
        println( "Ejecutando beneficiarios SyncCommand" )
        val dni = s"$tipoDocumento$numeroDocumento"
        val beneficiariosAsegurado = AseguradoPolizasServicio.polizasAsegurado( dni, configuration.repoBeneficiarios,
          configuration.repoAsegurados )
        beneficiariosAsegurado
          .map( lista => {
            ( MensajeExito( Json.toJson( lista ).toString() ), List() )
          } )
          .recover {
            case error =>
              val codigoError = configuration.appLogger.loggerSplunk(  mensaje = s"Error consultando beneficiarios",
                Some( error ) )
              ( MensajeExito( Json.toJson( s"Error beneficiarios: $codigoError ${error.getMessage}" ).toString() ), List() )
          }
    }

}
```

BeneficiariosCommand es un case class que recibe como parámetros el tipo de documento y el número de documento del asegurado enviados desde la clase BeneficiariosCommandHelper. Esta clase hereda de SyncCommand, lo que permite hacer uso del método execute de forma síncrona, lo cual asegura que el método se llame solo una vez en cualquier momento, para que dos hilos no intenten acceder simultáneamente y, por lo tanto, interrumpan algún tipo de estado que el método mantiene.

Servicios

AseguradoPolizaServicio

Esta clase es de tipo trait, lo cual significa que actúa como una interfaz que contiene métodos necesarios para la consulta y manejo de los datos del asegurado que se envio como parámetro al momento de consumir el servicio. Los métodos definidos en la interfaz son los siguientes:

```scala
def polizasBeneficiarios( poliza: Poliza, beneficiarios: List[Beneficiario] ): PolizaBeneficiariosDTO = {
  val listaBeneficiarios = BeneficiariosPolizaServicio.beneficiariosPolizaCertificadoFilter( poliza, beneficiarios )
  val polizaDTO = PolizaDTO( poliza.numeroPoliza, poliza.cdSubramo, poliza.nombrePlan, listaBeneficiarios )
  PolizaBeneficiariosDTO( polizaDTO )
}
```

Este método recibe una póliza y una lista de beneficiarios. Por medio de la función beneficiariosPolizaCertificadoFilter crea una lista de beneficiarios, los

asocia a la póliza, y crea una nueva PolizaDTO que es enviada como parámetro al constructor de una nueva PolizaBeneficiariosDTO.

```
def polizasAsegurado( dni: String, beneficiariosRepositorio: BeneficiariosRepositorio, vinculacionesRepositorio: VinculacionesRepositorio )
                ( implicit ec: ExecutionContext ): Future[PolizaAseguradoBeneficiariosDTO] = {

  for {
    vinculaciones <- obtenerVinculacionesAsegurado( dni )( ec )( vinculacionesRepositorio )
    polizasConCertificado <- obtenerPolizasCertificados( vinculaciones )
    listaBeneficiarios <- obtenerBeneficiariosPolizaCertificado( polizasConCertificado )( ec )( beneficiariosRepositorio )
    polizasBeneficiarios <- obtenerAsociacionBeneficiariosPoliza( vinculaciones, listaBeneficiarios )
    beneficiariosPolizaAsegurado <- obtenerPolizasBeneficiariosByAsegurado( Option( vinculaciones.head.asegurado.persona ), polizasBeneficiarios )

  } yield beneficiariosPolizaAsegurado

}
```

Este método recibe el dni del asegurado y los repositorios de datos, de manera interna por medio de la función obtenerVinculacionesAsegurado consulta las pólizas que tiene y las guarda en una lista la cual es enviada como parámetro a la función obtenerPolizasCertificados, que se encarga de crear una lista de tuplas póliza-certificado. Con estas tuplas se buscan todos los beneficiarios con el método obtenerBeneficiariosPolizaCertificado. Ya teniendo todos los beneficiarios se asocian con cada una de las tuplas póliza-certificado que correspondan por medio de la función obtenerAsociacionBeneficiariosPoliza para finalmente enviar esta lista al método obtenerPolizasBeneficiariosByAsegurado para que construya una nueva PolizaAseguradoBeneficiariosDTO.

La palabra clave yield usada en el método devolverá un resultado después de completar las iteraciones del bucle. El bucle for utiliza el búfer internamente para almacenar el resultado iterado y al finalizar todas las iteraciones produce el resultado final de ese búfer. No funciona como bucle imperativo, es decir, no se ejecuta de manera secuencial. El tipo de la colección que se devuelve es el mismo tipo en el que tendemos a iterar, por lo tanto, retorna el objeto beneficiariosPolizaAsegurado.

```
def obtenerVinculacionesAsegurado( dni: String )( implicit ec: ExecutionContext ): ( VinculacionesRepositorio ) => Future[List[Poliza]] =
  ( repo: VinculacionesRepositorio ) => {
    repo.obtenerVinculacionesAsegurado( dni ).recoverWith {
      case e =>
        Future.failed( new ErrorConsultaDBException( s"Error consultando las vinculaciones para el $dni", e ) )
    }
  }
```

Este método recibe el DNI de un asegurado y con base a este, va y consulta sus datos personales como también todas las pólizas de vida a las que está asociado, por medio de la función obtenerVinculacionesAsegurado de la clase VinculacionesRepositorio. El tratamiento de posibles excepciones, se hace por medio de la función .recoverWith, que es un método que permite intercambiar un Failure por un mensaje de error más claro y diciente para el programador.

```
def obtenerBeneficiariosPolizaCertificado( polizasCertificado: List[( String, String )] )
                                        ( implicit ec: ExecutionContext ): ( BeneficiariosRepositorio ) => Future[List[Beneficiario]] =
  ( repo: BeneficiariosRepositorio ) => {
    repo.obtenerBeneficiariosPorPolizaCertificado( polizasCertificado ).recoverWith {
      case e =>
        Future.failed( new ErrorConsultaDBException( s"Error en la consulta de beneficiarios por póliza y certificado", e ) )
    }
  }
```

Este método recibe una lista de tuplas póliza-certificado, con esta información va y consulta a la base de datos todos los beneficiarios haciendo uso del método obtenerBeneficiariosPorPolizaCertificado, trayendo los datos personales de la tabla PERSONAS y la información de parentesco, porcentaje de distribución, y tipo de beneficiario de la tabla BENEFICIARIOS.

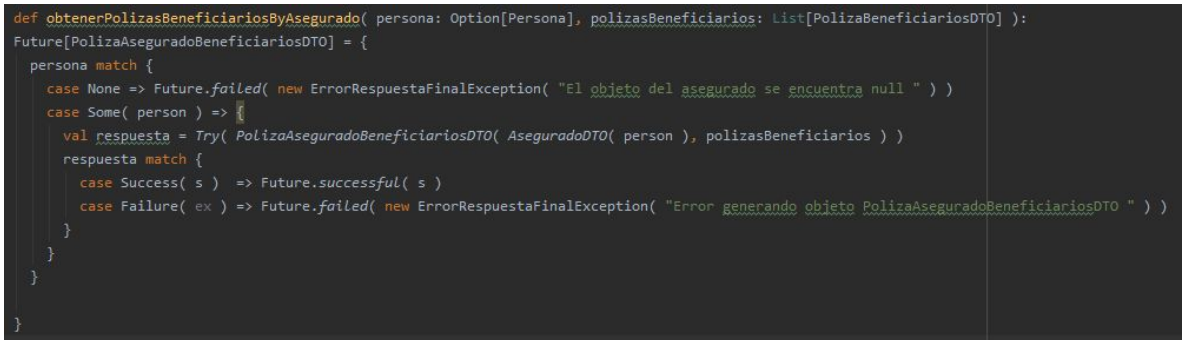
```
def obtenerPolizasCertificados( vinculaciones: List[Poliza] ): Future[List[( String, String )]] = {
  if ( vinculaciones.isEmpty )
    Future.failed( new ParametroNoEncontradoException( "No se encontraron polizas asociadas con la información del asegurado recibida" ) )
  else Future.successful( vinculaciones.map( a => ( a.numeroPoliza, a.asegurado.numeroCertificado ) ) )
}
```

Este método recibe una lista de pólizas, la cual en caso de estar vacía va a retornar un mensaje indicando "No se encontraron pólizas asociadas con la información del asegurado recibida", en caso de tener elementos va a crear una nueva lista de tuplas que guardan el número de póliza y el certificado. En la implementación, por medio del condicional IF, se valida si la lista de pólizas recibida como parámetro está vacía con la instrucción vinculaciones.isEmpty, en caso de que la respuesta sea TRUE, se crea una nueva excepción de tipo ParametroNoEncontradoException, en caso de que la lista tenga elementos por medio del método .map() se aplica a cada elemento una reducción de campos, guardando solo numeroPoliza y numeroCertificado.

```
def obtenerAsociacionBeneficiariosPoliza( vinculaciones: List[Poliza], listaBeneficiarios: List[Beneficiario] ): Future[List[PolizaBeneficiariosDTO]] = {
  if ( listaBeneficiarios.isEmpty ) Future.failed( new ParametroNoEncontradoException( "No se encontraron beneficiarios asociados a ninguna póliza del asegurado" ) )
  else {
    val response = Try( vinculaciones.map( vinculacion => {
      polizasBeneficiarios( vinculacion, listaBeneficiarios )
    } ) )

    response match {
      case Success( s )  => Future.successful( s )
      case Failure( ex ) => Future.failed( new ErrorCalculoException( "Error generando objeto PolizaBeneficiariosDTO" ) )
    }
  }
}
```

Este método recibe una lista de pólizas y una lista de beneficiarios, la cual en caso de estar vacía va a retornar un mensaje indicando "No se encontraron beneficiarios asociados a ninguna póliza del asegurado", en caso de que la lista de beneficiarios tengan elementos, internamente el método va a hacer uso por cada póliza de la función polizasBeneficiarios, la cual va a construir una nueva PolizaBeneficiariosDTO en donde relaciona póliza y beneficiarios. En la implementación del método, por medio del condicional IF se valida con la función isEmpty() que la lista de beneficiarios recibida como parámetro no esté

vacía, en caso de estarlo se crea una nueva excepción, sino para cada póliza por medio de .map() de la lista de vinculaciones, se relacionan los beneficiarios asociados. la variable response almacenará el resultado del Try implementado para manejo de excepciones. y finalmente, con la palabra reservada match se establece la respuesta del servicio para el caso correspondiente al valor de la variable response.



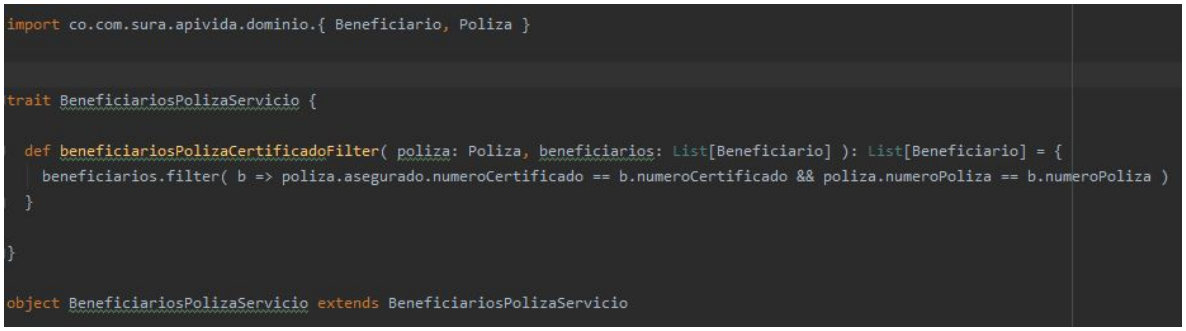
```
def obtenerPolizasBeneficiariosByAsegurado( persona: Option[Persona], polizasBeneficiarios: List[PolizaBeneficiariosDTO] ):
Future[PolizaAseguradoBeneficiariosDTO] = {
  persona match {
    case None => Future.failed( new ErrorRespuestaFinalException( "El objeto del asegurado se encuentra null " ) )
    case Some( person ) => {
      val respuesta = Try( PolizaAseguradoBeneficiariosDTO( AseguradoDTO( person ), polizasBeneficiarios ) )
      respuesta match {
        case Success( s )  => Future.successful( s )
        case Failure( ex ) => Future.failed( new ErrorRespuestaFinalException( "Error generando objeto PolizaAseguradoBeneficiariosDTO " ) )
      }
    }
  }

}
```

Este método recibe un elemento de tipo Persona y una lista de PolizaBeneficariosDTO, en caso de que Persona tenga como valor None, el método debe retornar un mensaje indicando: "El objeto del asegurado se encuentra null ", en caso de que tenga un valor, se va a crear una nueva PolizaAseguradoBeneficiariosDTO, en donde se relacionan los datos de la persona y su lista de PolizaBeneficiariosDTO. En la implementación del método se valida el valor del parámetro persona recibido, el cual es de tipo Option. Un Optión[T] en Scala es un contenedor para cero o un elemento de un tipo dado. Un Option [T] puede ser un objeto Some [T] o None , que representa un valor faltante.El tipo Option se usa con frecuencia en los programas Scala y puede compararlo con el valor nulo disponible en Java que indica que no hay ningún valor.

En caso de que el parámetro persona recibido tenga un valor de None, se crea una nueva Excepción, si el valor es Some, se procede a hacer uso de la función PolizaAseguradoBeneficiariosDTO dentro de un Try que capture algún posible error. La respuesta del Try se compara con los casos Success y Failure, usando la palabra reservada match,en el caso exitoso se retorna el objeto representado por la variable s y en el caso fallido, se crea una excepción que indica que el objeto no pudo ser generado.

Finalmente, para que la clase pueda ser accedida por otras que hacen parte de la estructura del proyecto se crea un nuevo objeto, que herede de la misma clase, para implementar todos los métodos anteriormente explicados.

BeneficiariosPolizaServicio



```
import co.com.sura.apivida.dominio.{ Beneficiario, Poliza }

trait BeneficiariosPolizaServicio {

  def beneficiariosPolizaCertificadoFilter( poliza: Poliza, beneficiarios: List[Beneficiario] ): List[Beneficiario] = {
    beneficiarios.filter( b => poliza.asegurado.numeroCertificado == b.numeroCertificado && poliza.numeroPoliza == b.numeroPoliza )
  }

}

object BeneficiariosPolizaServicio extends BeneficiariosPolizaServicio
```

Esta clase es de tipo trait, lo cual significa que actúa como una interfaz que contiene el método necesario para el manejo de los datos de los beneficiarios asociados a una de las pólizas que tiene el asegurado que se recibe como parámetro al momento de consumir el servicio.

El método beneficiariosPolizaCertificadoFilter recibe la informaciòn de una póliza en específico y una lista con todos los beneficiarios que se encontraron con la consulta a la base de datos. Con estos datos realiza un filtro, en donde recorre la lista de beneficiarios uno a uno, comparando el número de póliza y el certificado con los datos de la Póliza recibida. Finalmente retorna todos los beneficiarios que encontró relacionados a esa póliza. En la implementación se hace uso del método Filter(), el cual permite filtrar los elementos de una colección para crear una nueva colección que contenga solo los elementos que coincidan con los criterios de filtrado, en este método en particular los criterios utilizados es que el beneficiario debe tener un número de póliza y un número de certificado que coincidan con los valores que tiene el objeto Póliza recibido como parámetro

Finalmente se crea un nuevo objeto que hereda de la misma clase, para que el método pueda ser implementado por otras clases del proyecto..

Pruebas

TestKit: Clase de prueba

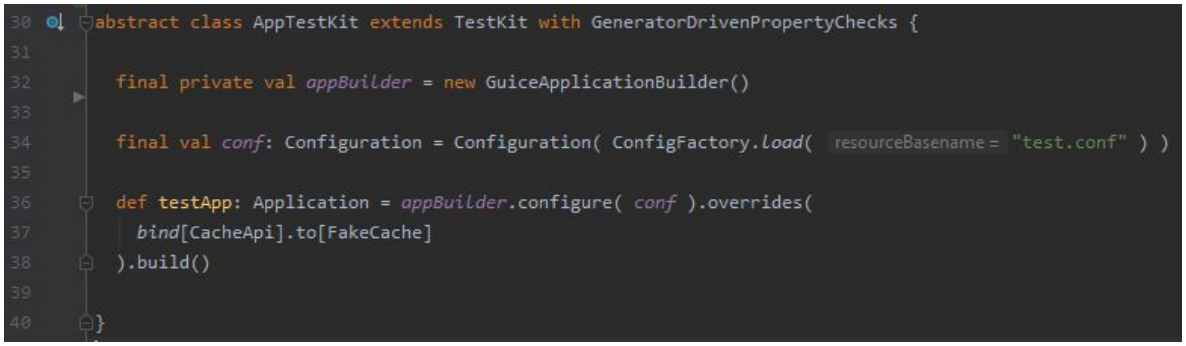
En ScalaTest + Play , usted define clases de prueba extendiendo de PlaySpec

```
16  abstract class TestKit extends PlaySpec
```

FakeCache:

Almacenamiento en caché en una web La aplicación es el proceso de almacenar elementos generados dinámicamente, ya sean objetos de datos, páginas o partes de una página, en la memoria en el momento inicial en que se solicitan. Esto se puede reutilizar más tarde si se realizan solicitudes posteriores para los mismos datos, lo que reduce el tiempo de respuesta y mejora la experiencia del usuario.

AppTestKit:



Al utilizar Guice para la inyección de dependencias , podemos configurar directamente cómo se crean los componentes y las aplicaciones para las pruebas. Esto incluye agregar enlaces adicionales o anular enlaces existentes.

GuiceApplicationBuilder provee una builder API para configurar la inyección de dependencias y creación de una aplicación.

El framework Play con frecuencia requiere una aplicación en ejecución como contexto. Cómo estamos utilizando la inyección de dependencia de Guice predeterminada, hacemos uso de la clase GuiceApplicationBuilder instanciandola en la variable privada appBuilder configurandola por medio del archivo “test.conf”.

Finalmente la función testApp nos permite crear la aplicación que nos servirá para las pruebas, haciendo uso de la instancia configurada de GuiceApplicationBuilder y la memoria caché establecida.

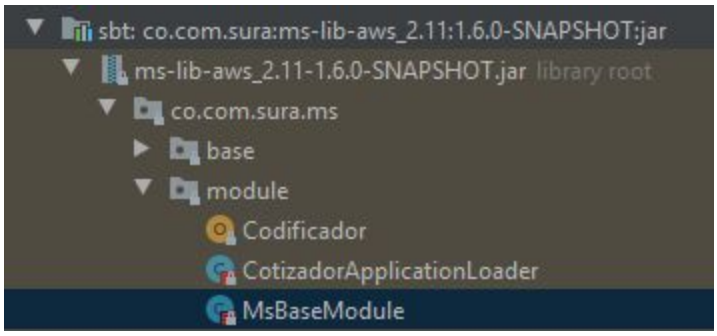
Archivo test.conf

Este archivo es el encargado de habilitar la librería ms-lib-aws_2.11-1.6.0-SNAPSHOT.jar, la cual contiene la configuración de autenticación y seguridad establecida por la compañía para el desarrollo de microservicios.

test.conf

```
enabled += "co.com.sura.ms.module.MsBaseModule"
```

Librería ms-lib-aws



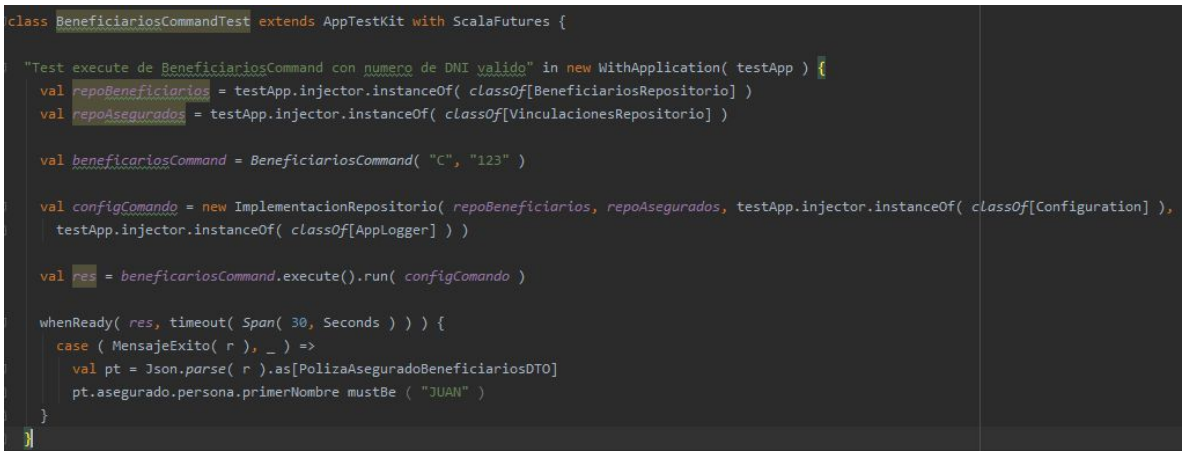
Clase MsBaseModule

```scala
package co.com.sura.ms.module

import co.com.sura.api.aggregate.authentication.UserAuthentication.AuthorizationAttribute
import co.com.sura.api.aggregate.authentication.filtro.play.{ UserAction, UserAuthentication }
import co.com.sura.ms.base.seguridad.{ CustomUserAuthenticationBuilder, SeusActionBuilder, SeusPropsLoader }
import com.google.inject.{ AbstractModule, Injector, Provides }
import net.codingwell.scalaguice.ScalaModule
import play.api.cache.CacheApi
import play.api.libs.ws.WSClient
import play.api.mvc.ActionBuilder

import scala.concurrent.ExecutionContext

/**
 * Created by roger on 18/07/16.
 */
class MsBaseModule extends AbstractModule with ScalaModule {

  override def configure(): Unit = {
    bind[SeusPropsLoader].asEagerSingleton()
  }

  @Provides
  def providesUserAuthentication( injector: Injector ): ( Option[List[AuthorizationAttribute]], Option[String] ) => ActionBuilder[UserAction] = {
    val wsClient = injector.getInstance( classOf[WSClient] )
    val cacheApi = injector.getInstance( classOf[CacheApi] )
    implicit val ec = injector.getInstance( classOf[ExecutionContext] )
    ( authAttribute, resource ) => {
      CustomUserAuthenticationBuilder( authAttribute, resource, wsClient, cacheApi )
    }
  }
}
```

La clase está conformada por 2 funciones, una de configuración y otra de autenticación, la compañía establece por defecto esta clase en la construcción de microservicios para garantizar un correcto uso de la infraestructura y seguridad en la implementación.

BeneficiariosCommandTest

AppTestKit: Contiene la función testApp, que permite hacer uso de la aplicación construida para pruebas.
ScalaFutures: Contiene la función whenReady, que permite convertir un resultado futuro a uno esperado.



Guardamos en la variable configComando, la configuración necesaria para hacer uso del método execute de la clase BeneficiariosCommand, adicional a eso se crea una instancia de la clase, enviando como parámetros al constructor el tipo de documento y el número de documento. Por medio de la instancia, ejecutamos la función execute, la cual retornará un futuro de Mensaje que guardaremos en la variable res. Con whenReady esperamos el resultado del futuro, dicho valor contendrá en caso de ser exitoso un string que representa el JSON con la respuesta que envía el servicio, con Json.parse(r) se convierte a JSON la respuesta y con .as[PolizaAseguradoBeneficiariosDTO] se crea una nueva instancia del dominio a la cual podemos acceder.

AseguradosRepositorioSpec

```
"Test obtenerVinculacionesAsegurado de DNI con polizas de vida asociadas" in new WithApplication( testApp ) {
  val repoVinculaciones = testApp.injector.instanceOf( classOf[VinculacionesRepositorio] )
  val res = Await.result( repoVinculaciones.obtenerVinculacionesAsegurado( dni = "C123" ), 15 seconds )
  res.isEmpty mustBe false
}
```

En la variable repoVinculaciones guardamos una instancia de la clase VinculacionesRepositorio, de tal modo que podamos hacer uso en los tests de la base de datos. Luego en la variable res guardamos el resultado de la función obtenerVinculacionesAsegurado que recibe como parámetro el dni del asegurado. Awair.result, nos permite convertir un resultado futuro al resultado esperado.

BeneficiariosRepositorioSpec

```
class BeneficiariosRepositorioSpec extends AppTestKit {

 "Test obtenerBeneficiariosPorPolizaCertificado con 2 beneficiarios en 2 polizas" in new WithApplication( testApp ) {
  val poliza = List( ( "083000450694", "13" ), ( "083001391876", "2" ) )
  val repoBeneficiarios = testApp.injector.instanceOf( classOf[BeneficiariosRepositorio] )
  val res = Await.result( repoBeneficiarios.obtenerBeneficiariosPorPolizaCertificado( poliza ), 15 seconds )
  res.length mustBe 2
 }
```

En la variable repoBeneficiarios guardamos una instancia de la clase BeneficiaopsRepositorio, de tal modo que podamos hacer uso en los tests de la base de datos. Luego en la variable res guardamos el resultado de la función obtenerBeneficiariosPorPolizaCertificado que recibe como parámetro la lista poliza, la cual contiene las tuplas poliza-certificado que se buscarán en la bd. Await.result, nos permite convertir un resultado futuro al resultado esperado.

Diagramas

Diagrama de dominio

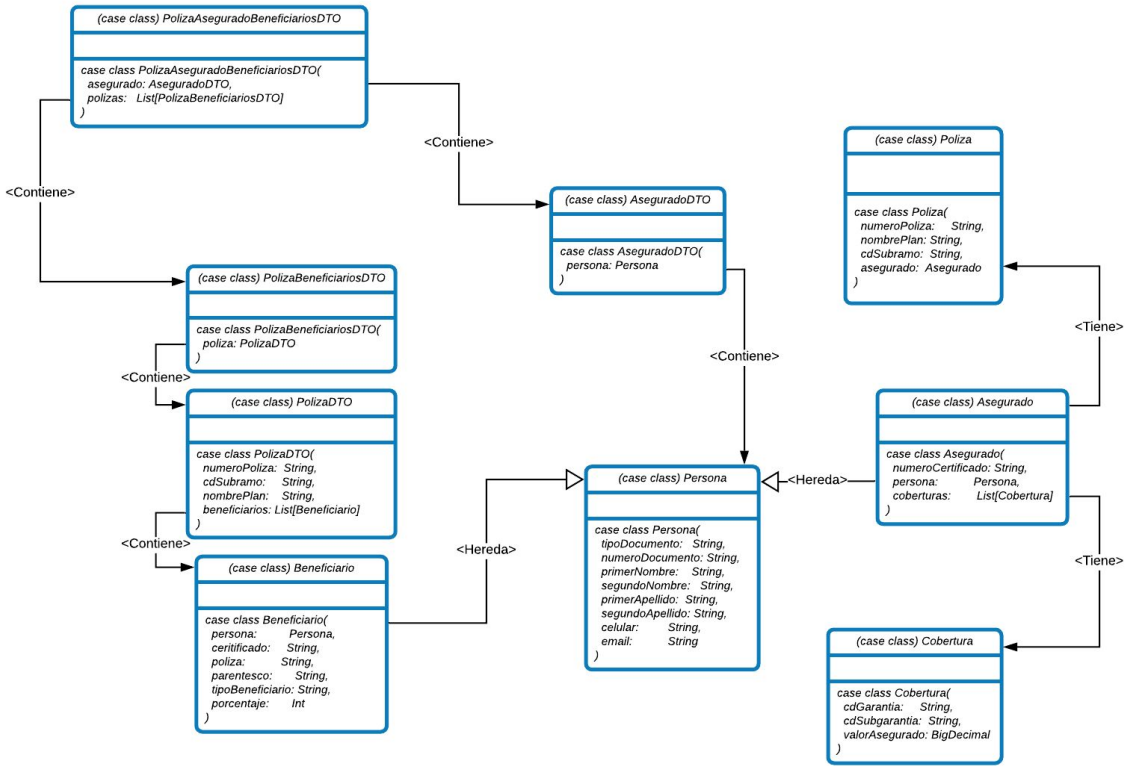
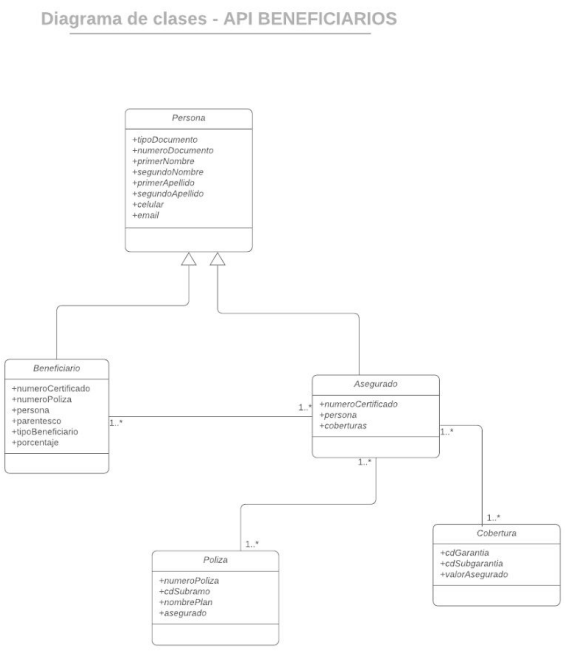
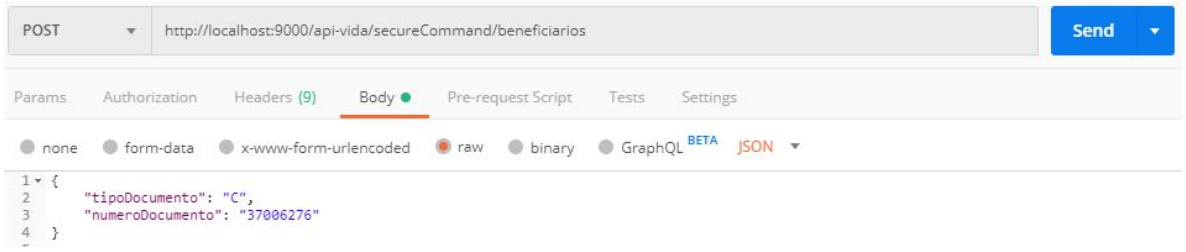


Diagrama de clases



Microservicio

Para hacer uso del servicio, se hace un request con el tipo de documento y número de documento del asegurado a consultar en la base de datos.



La respuesta del servicio es un archivo JSON que contiene los datos personales del asegurado, junto con una lista de pólizas que a su vez entre sus datos retorna la lista de los beneficiarios.

POST http://localhost:9000/api-vida/secureCommand/beneficiarios

Pretty Raw Preview Visualize BETA JSON

```
{
    "asegurado": {
        "persona": {
            "tipoDocumento": "C",
            "numeroDocumento": "37006276",
            "primerNombre": "ANA",
            "segundoNombre": "PAULA",
            "primerApellido": "BLANCO",
            "segundoApellido": "TORRES CQLII",
            "celular": "3148207738",
            "email": "laura.victoriau@gmail.com"
        }
    },
    "polizas": [
        {
            "poliza": {
                "numeroPoliza": "081003522479",
                "cdSubramo": "VO3",
                "nombrePlan": "Plan inversión protegida",
                "beneficiarios": [
                    {
                        "persona": {
                            "tipoDocumento": "C",
                            "numeroDocumento": "3753936",
                            "primerNombre": "PEDRO",
                            "segundoNombre": "FERNEY",
                            "primerApellido": "AGUDELO",
                            "segundoApellido": "PEÑUELA CQLII",
                            "celular": "3149874563",
                            "email": "PERMANENCIA@HOTMAIL.ES"
                        },
                        "parentesco": "AM",
                        "tipoBeneficiario": "GR",
                        "porcentaje": 100,
                        "numeroCertificado": "1"
```

Conclusiones

- Al hacer uso de la metodología SCRUM durante el desarrollo del microservicio se logró adquirir una dinámica de trabajo adecuada y ágil.

En donde se programaron reuniones que permitieron establecer el alcance del desarrollo y el plan de trabajo a seguir.

- Se logró crear servicio REST que al recibir como parámetros el tipo de identificación y número de un asegurado, retornará un archivo JSON con toda la información de las pólizas adquiridas con Sura y sus beneficiarios.
- Durante el desarrollo del microservicio fue sumamente importante la implementación de pruebas, las cuales permitieron detectar errores a tiempo y darles una pronta solución, evitando así deuda técnica y reprocesos.
- Respecto a los objetivos que se plantearon inicialmente y teniendo en cuenta los resultados obtenidos durante el desarrollo de la práctica académica, se puede evidenciar que se cumplieron satisfactoriamente.

Referencias Bibliográficas

● Selvam Palanimalai, J. P. (2017, septiembre). Scala microservices. Recuperado de https://scholar.google.com

● López, D. (2017b, 17 julio). Arquitectura de Software basada en Microservicios para Desarrollo de Aplicaciones Web. Recuperado de http://dspace.redclara.net/bitstream/10786/1277/1/93%20Arquitectura%20de%20Software%20basada%20en%20Microservicios%20para%20Desarrollo%20de%20Aplicaciones%20Web.pdf

● Navarro Cadavid, A. (2013, 4 junio). Revisión de metodologías ágiles para el desarrollo de software. Recuperado de https://www.redalyc.org/pdf/4962/496250736004.pdf