

Informe Trabajo Final



Integrante

Nombre

Correo

Legajo

Dammiano, Agustín

adammiano@itba.edu.ar

57702

Fecha: 24/01/2022

Índice

1. Introducción	2
2. Solución propuesta	3
2.1. Arquitectura	3
2.2. Backend	3
2.2.1. Estructura	3
2.2.2. Endpoints	4
2.2.3. Código	5
2.2.3.1 Manejo de errores	5
2.2.3.2 Conexión a la base de datos	6
2.2.3.3 Interactuar con la base de datos	7
2.2.3.4 Separacion de capas	7
2.3. Frontend	8
2.3.1. Página principal	8
2.3.2. Lista de entidad	9
3. Trabajo futuro	10
4. Conclusión	11

1. Introducción

Se decidió realizar una API REST que provea la información de la vacunación covid de Argentina. La motivación de este proyecto es ofrecer una mejor manera de acceder a esta información.

Actualmente el gobierno permite descargar dichos datos desde una página web en formato CSV. Lo cual agrega datos redundantes que en una base de datos estarían en otra tabla y serían referenciados por foreign keys. Además, si uno quiere transformar los datos o realizar una consulta es poco eficiente. Por estas razones es incómodo realizar una visualización de dichos datos.

El proyecto proveerá distintos endpoints para crear, consultar y eliminar las distintas entidades del mismo. También se podrá realizar una sincronización con el CSV del gobierno y consultar cuándo fue la última sincronización. Además se realizará un frontend donde se muestren todas estas funcionalidades.

2. Solución propuesta

2.1. Arquitectura

Se decidió desarrollar un backend en haskell y un frontend en React. La API se comunica con el servidor del gobierno por medio de HTTP para realizar la sincronización con la base de datos. También se decidió utilizar Postgresql como motor de base de datos ya que existía un conocimiento previo del mismo.

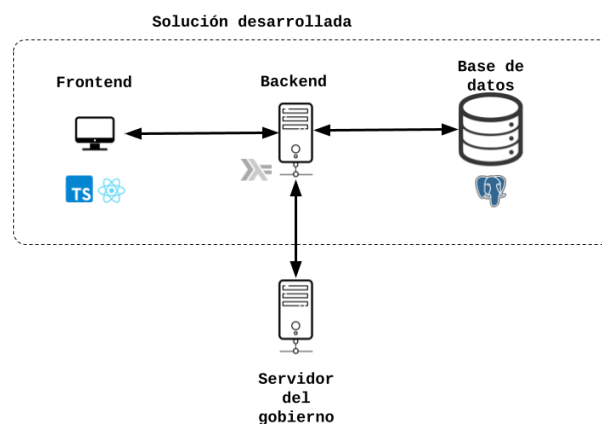


Figura 1: Arquitectura de la solución propuesta.

2.2. Backend

2.2.1. Estructura

A continuación se detalla los principales directorios del módulo backend:

- **Feature:** La carpeta Common contiene funciones que se comparten a través de las distintas entidades (como transformar un objeto a JSON). Las otras carpetas representan las distintas entidades (Jurisdiction, etc). Las cuales están divididos en capas:
 - **Controller:** Encargado de la comunicación HTTP.
 - **Service:** Contiene lógica de negocio.

- **Dao y Repository:** Encargados de manejar fuentes de datos. El primero se relaciona con la base de datos y el segundo con fuentes externas al proyecto.
- **Types:** Contiene el modelado.
- **Platform:** Contiene archivos de configuración general de la base de datos y el framework Scotty.
- **Postgresql:** Contiene los scripts para crear y migrar la base de datos.

Se va a enumerar distintas las entidades y lo que representan, cabe destacar que no tiene que existir correspondencia con las tablas de la base de datos:

- **Date:** fecha de aplicación de la dosis.
- **Department:** departamento.
- **Dose:** dosis aplicada.
- **Jurisdiction:** provincia.
- **Update:** sincronización de la información.
- **Vaccine:** vacuna aplicada

2.2.2. Endpoints

La API expone los siguientes endpoints para Department, Dose, Jurisdiction y Vaccine:

- GET /api/:entity
Permite acceder a la lista de dicha entidad.
- POST /api/:entity
Permite crear una instancia de esta entidad.
- DELETE /api/:entity
Permite borrar todas las instancias.
- DELETE /api/:entity/:id
Permite borrar una instancia.

También se exponen los siguientes endpoints para Jurisdiction y Date:

- GET /api/:entity/dose
Permite acceder a la lista de dicha entidad con información agregada de las dosis aplicadas. Por ejemplo se podría saber la cantidad de vacunas aplicadas en la provincia de Buenos Aires.

Los endpoint relacionados con la sincronización son los siguientes:

- GET /api/update
Permite obtener las fechas de las últimas actualizaciones.
- POST /api/update
Realiza la sincronización de los datos.

2.2.3. Código

2.2.3.1 Manejo de errores

Se utilizó la monada ExceptT para correr una secuencia de acciones que se interrumpe si se tira una excepción. En particular en la capa de servicio se utilizó las funciones *throwError* y *runExceptT*. A continuación se muestra un ejemplo:

```
getJurisdiction :: (Dao m) => Text ->
  m(Either JurisdictionError Jurisdiction)
getJurisdiction name = runExceptT $ do
  result <- lift $ listJurisdictions
    (JurisdictionFilter $ Just name) (Pagination 1 0)
  case result of
    [jurisdiction] -> return jurisdiction
    _ -> throwError $ JurisdictionNameNotFound name

createJurisdictionHandle :: (Dao m) =>
  (m(Either JurisdictionError Bool) ->
    m(Either JurisdictionError Bool)) ->
  CreateJurisdiction -> m(Either JurisdictionError Jurisdiction)
createJurisdictionHandle handler param = runExceptT $ do
  _ <- mapExceptT handler $ ExceptT $
    createJurisdictionFromDB param
  ExceptT $ getJurisdiction $ createJurisdictionName param
```

En el archivo de tipos están definidos los siguientes errores:

```
data JurisdictionError = JurisdictionNameNotFound Text |  
    JurisdictionAlreadyExist Text | UnknownError
```

Finalmente los controllers utilizan una función handler que traduce cada error con el código HTTP correspondiente. En el caso de que la operación sea un éxito devuelve el resultado.

2.2.3.2 Conexión a la base de datos

Se creó una función de alto orden para manejar las conexiones a la base de datos. Esta recibe una función que mapea los errores de la base a los errores definidos en los archivos de tipos. También recibe una función que describe la operación que se realizará sobre la base. A continuación se muestra su implementación:

```
withConn :: Database r m => (SqlError -> ConstraintViolation ->  
    IO (Either e a)) -> (Connection -> IO a) -> m (Either e a)  
withConn catcher action = do  
    pool <- asks getter  
    liftIO $ catchViolation catcher (fmap Right  
        (withResource pool action))
```

Finalmente se utiliza de la siguiente manera en los Daos (cabe destacar que los tipos de errores son los mismo que se mostraron en la sección anterior):

```
deleteJurisdictionFromDB :: Database r m => Integer ->  
    m (Either JurisdictionError Bool)  
deleteJurisdictionFromDB i = do  
    result <- withConn handler $ \conn -> execute conn qry (Only i)  
    return $ second (\_ -> True) result  
    where qry = "delete from jurisdiction where id = ?"  
  
handler :: (SqlError -> ConstraintViolation ->  
    IO (Either JurisdictionError a))  
handler _ (UniqueViolation x) = return $ Left $  
    JurisdictionAlreadyExist $ decodeUtf8 x  
handler _ _ = return $ Left UnknownError
```

2.2.3.3 Interactuar con la base de datos

Se crearon instancias de las clases FromRow y ToRow. De esta forma se logra que dependiendo del contexto se ejecuten distintas implementaciones. Se realizó de la siguiente manera:

```
data JurisdictionError = JurisdictionNameNotFound Text |
    JurisdictionAlreadyExist Text | UnknownError
instance FromRow Dose where
    fromRow = Dose <$> field <*> field <*> field <*> field
    <*> field <*> field <*> field <*> field <*> field
    <*> field <*> field <*> field

instance ToRow CreateDose where
    toRow param =
        [toField $ createDoseSex param,
         toField $ createDoseAge param,
         toField $ createDoseCondition param,
         toField $ createDoseLot param,
         toField $ createDoseDate param,
         toField $ createDoseSerie param,
         toField $ createDoseVaccineId param,
         toField $ createDoseResidenceJurisdictionId param,
         toField $ createDoseResidenceDepartmentId param,
         toField $ createDoseApplicationJurisdictionId param,
         toField $ createDoseApplicationDepartmentId param]
```

2.2.3.4 Separacion de capas

Con el fin de separar los controllers, los servicios y Daos se hizo que en cada archivo se declaren clases que actúan como dependencia. Las cuales quedan definidas cuando se instancian para esta aplicación. De esta manera es sencillo reemplazar una capa o reutilizarla en otro lado. A continuación se muestra como se declara en un controller y su instancia:

```
class Monad m => Service m where
    listJurisdictions :: JurisdictionFilter -> Pagination
        -> m [Jurisdiction]
    getJurisdiction :: Text
        -> m(Either JurisdictionError Jurisdiction)
    createJurisdiction :: CreateJurisdiction
        -> m(Either JurisdictionError Jurisdiction)
    deleteJurisdiction :: Integer -> m (Either JurisdictionError Bool)
    deleteJurisdictions :: m (Either JurisdictionError Bool)
    listJurisdictionsDose :: Pagination -> m [JurisdictionDose]
```

```
instance JurisdictionController.Service AppT where
  listJurisdictions = JurisdictionService.listJurisdictions
  getJurisdiction = JurisdictionService.getJurisdiction
  createJurisdiction = JurisdictionService.createJurisdiction
  deleteJurisdiction = JurisdictionService.deleteJurisdiction
  deleteJurisdictions = JurisdictionService.deleteJurisdictions
  listJurisdictionsDose = JurisdictionService.listJurisdictionsDose
```

2.3. Frontend

El frontend se desarrolló en React. Como el trabajo está enfocado en Haskell no se hablara de la implementación del mismo, pero se hará un breve manual de uso.

2.3.1. Página principal

En esta página se puede ver visualizaciones de la vacunación. La barra de navegación permite ir a la página de dose, jurisdiction, department y vaccine. Clickeando el logo se puede volver a esta página. También se muestra el horario de la última actualización, cabe destacar que se puede clicar para actualizar los datos.

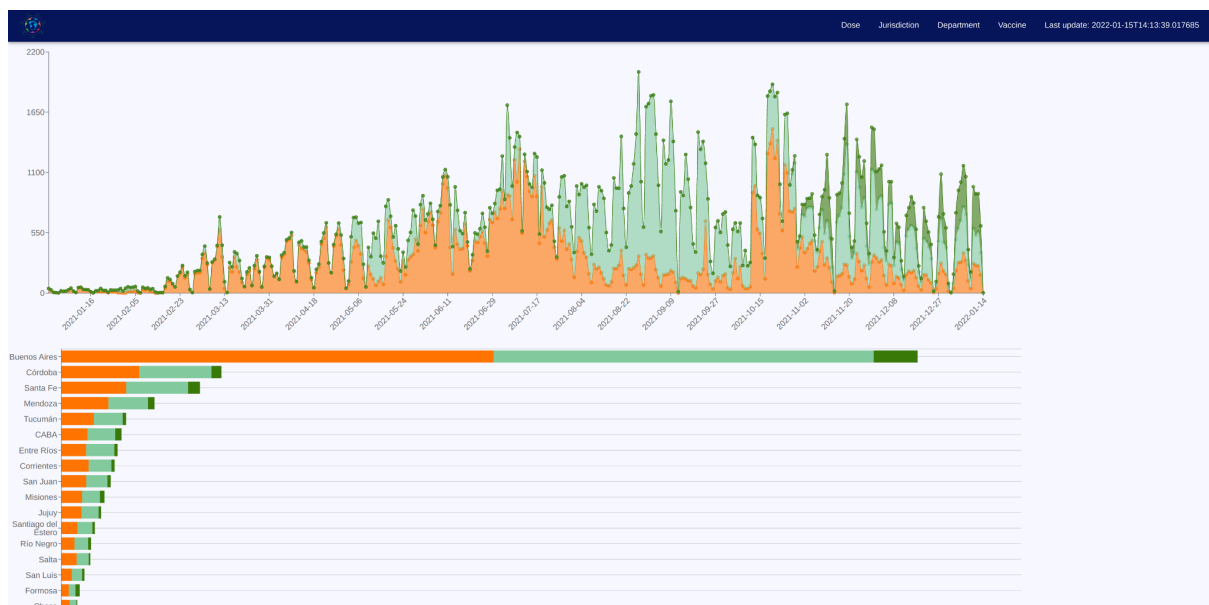


Figura 2: Página principal.

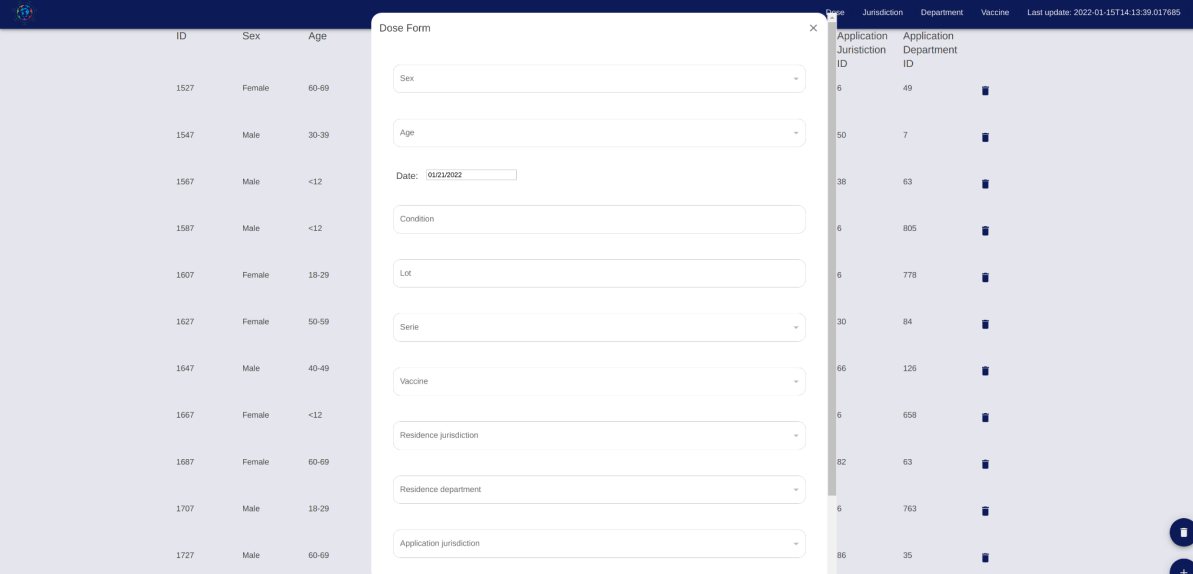
2.3.2. Lista de entidad

En esta vista se muestra un listado de la entidad, utilizando paginado infinito. Se permite borrar un elemento de la lista o todos clickeando el botón flotante de borrado. Además se puede crear un nuevo elemento clickeando el “+”. Lo cual muestra un formulario de dicha entidad.



ID	Sex	Age	Condition	Lot	Date	Serie	Vaccine ID	Residence Jurisdiction ID	Residence Department ID	Application Jurisdiction ID	Application Department ID	Vaccine	Last update: 2022-01-15T14:13:39.017685
1527	Female	60-69	60-69	B-100221	2021-06-15	2	1534	6	49	6	49		
1547	Male	30-39	30-39	NH0174	2021-06-10	1	1551	50	49	50	7		
1567	Male	<12	<12	20210781978	2021-10-13	1	1538	38	63	38	63		
1587	Male	<12	<12	20210882086	2021-10-24	1	1538	6	805	6	805		
1607	Female	18-29	18-29	20210681185	2021-07-17	1	1538	6	778	6	778		
1627	Female	50-59	50-59	PW40097	2021-11-24	3	1551	30	84	30	84		
1647	Male	40-49	40-49	NH0335	2021-07-16	1	1551	66	126	66	126		
1667	Female	<12	<12	20210882570	2021-12-14	1	1538	6	658	6	658		
1687	Female	60-69	60-69	I-220321	2021-04-16	1	1534	82	63	82	63		
1707	Male	18-29	18-29	NH0084	2021-08-30	1	1551	6	623	6	763		
1727	Male	60-69	60-69	I-250321	2021-04-09	1	1534	86	35	86	35		

Figura 3: Página de dosis.



ID	Sex	Age	Condition	Lot	Date	Serie	Vaccine	Residence jurisdiction	Residence department	Application jurisdiction
1527	Female	60-69			06/21/2022					
1547	Male	30-39								
1567	Male	<12								
1587	Male	<12								
1607	Female	18-29								
1627	Female	50-59								
1647	Male	40-49								
1667	Female	<12								
1687	Female	60-69								
1707	Male	18-29								
1727	Male	60-69								

Figura 4: Formulario de dosis.

3. Trabajo futuro

A continuación se plantea mejoras del trabajo:

- **Autenticación:** Se propone agregar un esquema de autenticación como el de JWT. De esta forma se podría restringir los endpoints que modifican la base de datos.
- **Testing:** Agregar test unitarios e integrales.
- **HTTPS:** Agregar que la api funcione con HTTPS. Este punto es necesario para poder llevar a esta aplicación a un ambiente de producción.
- **Logging:** Agregar logs a archivos con el fin de mejorar la trazabilidad de la API. De esta manera se simplificará el proceso de desarrollo de nuevas funcionalidades.
- **Sincronización:** Mejorar la sincronización para que funcione más rápido. En esta versión se reutilizo las funcionalidades que proveen los distintos servicios lo cual resultó en grande tiempo sincronización. Una alternativa es usar el método COPY de postgresql que está diseñado para copiar datos.
- **Endpoints de consulta:** Agregar más endpoint de consultas con el fin de realizar más visualizaciones.
- **Frontend:** Mejorar el frontend y agregar las nuevos gráficos a partir de las consultas mencionadas previamente.

4. Conclusión

En el trabajo se logró realizar API que soluciona el problema planteado. Para esto se aprovechó de distintas ventajas que proporciona Haskell. Una de estas es el uso de mónadas para abstraerse. Uno de los resultados de esta abstracción es la reutilización de código. Además con las mónadas se puede utilizar efectos secundarios, entre ellas están las que nos permiten interactuar con el mundo externo. Otra ventaja del lenguaje es el uso de clases e instancia permite utilizar un polimorfismo ad-hoc, de esta forma cambia la implementación usada en función del contexto.

Este trabajo permitió bajar las ideas teóricas, vistas en clase, a tierra. De esta manera se logró consolidar dichos conceptos. Lo cual es una buena forma de dar cierre a la materia.