72.60 - Programación Funcional

Instituto Tecnológico de Buenos Aires

---

# Informe trabajo final

---

*Nombre y Apellido:* Bianca Ritorto

*Legajo:* 57082

*Proyecto:* iMenu - a Scotty API

*Fecha:* 17/02/2021

# Índice

**1. Introducción** **2**

**2. Implementación** **3**
2.1. Estructura . . . . . 3
2.2. Funcionalidades principales . . . . . 3
2.3. Autenticación . . . . . 5
2.4. Persistencia . . . . . 6
2.5. Manejo de errores . . . . . 6

**3. Frontend** **8**
3.1. Menu . . . . . 8
3.2. Administrador . . . . . 9

**4. Mejoras propuestas** **11**

**5. Conclusión** **12**

# 1. Introducción

Eligiendo la materia de programación funcional como electiva quise satisfacer esa necesidad de salir un poco de mi entorno educativo actual. En las carreras de informática hoy en día se enseña, por sobre todo, a programar orientado a objetos. Me pareció que un complemento a lo que yo ya sabía era necesario para abrir la mente a diferentes formas de hacer las cosas.

Más que nada quería poder decidir, teniendo yo misma la información, qué paradigma era mejor para qué situación. Habiendo trabajado casi dos años en una empresa como desarrolladora de software pricipalmente en el lenguaje Java, con este trabajo final quise averiguar si las cosas nuevas que había aprendido del paradigma funcional servían o eran mejores para hacer mi trabajo de lo que es el paradigma orientado a objetos. De aquí surge en principio la motivación a realizar una API REST puramente en Haskell.

La idea específica de la funcionalidad de dicha API surge del contexto global que tocó vivir en 2020 (y 2021, por ahora). Casi todos los lugares de comida, ya sean bares, restaurantes, o cafés, tienen en reemplazo de una carta, un código QR en cada mesa que al ser escaneado trae el menú al celular del usuario. Ya sea a través de un PDF estático, o de una página web. Mi idea fue ir un poco más allá y proveer funcionalidad básica de pedido de comidas a través de ese menú. Es decir, que no solo se pueda ver el menu, sino que se pueda enviar un pedido de lo que uno quiere comer a través de la misma web. La implementación de dicha funcionalidad será explicada en detalle en lo que resta del informe.

La herramienta que decidí utilizar en profundidad para poder llevar a cabo el proyecto es un framework web en Haskell llamado Scotty. Scotty está inspirado en Sinatra, el framework web del lenguaje de programación Ruby y utiliza los paquetes WAI (web application interface) y Warp.

Investigué otros frameworks tales como Yesod y Snap, pero me decidí por Scotty por un par de razones. Parecía ser el más simple y minimalista, que es una de las cosas que quería lograr con el proyecto: intentar disminuir las complicaciones de Java con las que me había encontrado durante mi trabajo. Por otro lado, parecía estar bastante bien documentado y utilizado por la comunidad, algo que me iba a servir mucho ya que veía desde entrada que la manera de resolver problemas bajo un paradigma al que no estaba acostumbrada iba a ser el principal impedimento en el desarrollo, y el que me tomara más tiempo.

# 2. Implementación

## 2.1. Estructura

La aplicación está dividida en un back-end (Haskell) y un front-end (React). Esta sección tratará de la estructura principal del back-end. En el back-end existen tres paquetes principales:

- Core: Dentro de este directorio se encuentra todo lo relacionado al "negocio". Es decir, las funcionalidades principales de la API están englobadas por este paquete. Dentro de cada elemento se tienen las clases de **Controller, DAO, Service** y **Types** respectivamente.
  - **Category**: Todo lo relacionado al manejo de las categorías de comida. Por ejemplo "entradas", "postres", etc.
  - **Item**: El manejo de los items de comida en sí. Por ejmplo, una milanesa, o un flan.
  - **Order**: La orden que el usuario puede crear para mandar a la cocina que puede contener varios items de distintas categorias.
- Platform: Dentro del módulo de la plataforma en sí se tienen las conexiones a la base de datos en **Postgres**, una clase utilitaria para la serialización de JSONs, **JSONUtil** y algunos **Types** utilizados a lo largo de la aplicación. También está el archivo **Home** que levanta la aplicación, configura el middleware e inicializa las rutas a exponer, entre otras cosas.
- postgresql: Este paquete contiene únicamente un script de SQL para inicializar las tablas necesarias para el funcionamiento de la API. El script únicamente corre si las tablas no fueron creadas todavía en el servidor al que este conectado.

## 2.2. Funcionalidades principales

Lo primero y lo principal que aporta funcionalidad a iMenu es la habilidad de exponer opciones de comida al usuario. Es por eso que la primer funcionalidad desarrollada estuvo centrada en los Items. La API expone los siguientes endpoints para ello:

- `GET /api/items`

  Trae todos los items cargados en la base de datos.

- `GET /api/items/:category`

  Trae todos los items de la categoría especificada por path param.

Por otro lado, están expuestas las rutas para el manejo del *backoffice* de los items. Estas rutas estan dentro de lo que serían las **adminRoutes**, y requieren de un user y password para poder ser ejecutadas (basic auth).

- `POST /admin/items`

  Crea un nuevo item.

- `PUT /admin/items/:slug`

  Modifica el item identificado por su 'slug' en el path param.

- `DELETE /admin/items/:slug`

  Elimina el item identificado por su 'slug' en el path param.

Los mismos tipos de endpoints están expuestos para las **Category**: creación, obtención y eliminación. No se permiten modificaciones, ya que el nombre de la categoría debe ser único para cada una. Si se quisiera modificar, se debería eliminar y crear de nuevo.

Para las ordenes se tienen los siguientes endpoints:

- `GET /api/orders`

  Trae todas las ordenes disponibles en el momento.

- `GET /api/orders/:table`

  Trae la orden actual de la mesa requerida, si la hay.

- `POST /api/orders`

  Publica una orden para que sea recibida por el *backoffice* (la cocina) y comenzada a preparar. Tiene asociada la mesa y los items requeridos.

El único endpoint bajo las **adminRoutes** de las órdenes es el siguiente. Si bien elimina una orden, el concepto debería ser el de "finalizar" la orden. Esto se entiende debería activarse cuando el mesero entrega la comida a la mesa. Debido al alcance del proyecto, por ahora las ordenes se eliminan, aunque no es la solución óptima. Esto será discutido con más detalle en la sección de Mejoras propuestas.

- `DELETE /api/orders/:table`

  Finaliza la orden de la mesa especificada.

## 2.3. Autenticación

No se consideró que la seguridad era un requerimiento funcional prioritario dentro del proyecto debido a su carácter de uso local dentro de la casa de comidas, y debido a que no se está trabajando con información delicada de ninguna parte (ni del cliente ni del vendedor). Habiendo dicho esto, luego del MVP implementado actualmente, uno de los siguientes pasos debería ser el de agregar JWT a los endpoints para poder confiar al 100 % en el correcto funcionamiento de la API.

Sin embargo, se consideró necesario incluir algún tipo de protección a los endpoints de administración de datos. Es por eso que se realizó una investigación del paquete wai-extra que contiene utilidades para agregar autenticación a las requests HTTP.

Luego de varias pruebas, no se lograba hacer que el middleware diferenciara los endpoints que comenzaban con `/api` con los que debían ser autenticados, `/admin`. Se logró agregar autenticación a todos los endpoints por igual, pero esto no era lo que se quería implementar.

En una segunda instancia se fue agregando autenticación en cada endpoint por separado, pero esto hacía que el código quedara engorroso y no era mantenible. Finalmente se realizó una pregunta en Stack Overflow, y la comunidad aportó la respuesta necesaria para poder hacer lo siguiente:

Dentro de **Home**, cuando se configura el middleware, se agregó esta linea:

```
middleware $ basicAuth
   (\u p -> return $ u == "username" && p == "password")
   authSettings
```

La función `authSettings` tiene tipo `AuthSettings`, definido dentro del paquete wai-extra que permite, entre otras cosas, diferenciar cuándo se necesita aplicar autenticación a los endpoints y cuándo no. De esta forma, se hace una división del endpoint con la ayuda de `pathInfo`, y se busca si el endpoint contiene un `/admin` en algún lado de la url:

```
authSettings :: AuthSettings
authSettings = "My Realm" { authIsProtected = needsAuth }

needsAuth :: Request -> IO Bool
needsAuth req = return $ case pathInfo req of
   "admin":_ -> True
   _         -> False
```

## 2.4. Persistencia

Para la persistencia se optó por utilizar un esquema relacional en PostgreSQL específicamente. Esta decisión fue tomada más que nada debido al paquete postgresql-simple que en la investigación realizada antes de comenzar el proyecto probó ser la más utilizada y mantenida por la comunidad.

Tranquilamente se podría haber escogido otro tipo de persistencia como MongoDB, por ejemplo, pero debido a la experiencia con queries SQL y manejo de bases relacionales, se creyó que se tardaría menos tiempo en implementar una solución relacional.

Sin embargo, comprender como utilizar la librería elegida tomó bastante tiempo. Luego de mucha prueba y error, las implementaciones de los DAOs quedaron bastante simples y correctas. El impedimento más grande encontrado fue al intentar guardar en la base de datos un array de Text. Otra vez se recurrió a la comunidad en Stack Overflow y la solución resultó ser el uso de `PGArray` para poder convertir una lista de Haskell en un array entendible por PostgreSQL:

```
addOrder :: Postgres r m => OrderIntent -> m ()
addOrder param =
void . withConn $ \conn -> execute conn qry (orderIntentTableNumber
    param, PGArray (orderIntentItems param))
where
  qry = "insert into orders (table_number, items, created_at,
      updated_at) values (?, ?, now(), now())"
```

## 2.5. Manejo de errores

Una API REST no sería una buena API REST sin el manejo de errores adecuado. Esto se logra en lenguajes como Java con el correcto manejo y parseo de excepciones. En el caso de Haskell, esto viene incluido con el framework expuesto por Scotty. El módulo `Trans/Except` extiende la funcionalidad básica de una mónada con la habilidad de tirar excepciones. Citando la documentación que lo explica muy bien, la mónada Except provee la siguiente funcionalidad:

> A sequence of actions terminates normally, producing a value, only if none of the actions in the sequence throws an exception. If one throws an exception, the rest of the sequence is skipped and the composite action exits with that exception.

En las clases **Service** se utiliza este módulo y en particular las funciones `throwError` y `runExceptT` para poder atajar los errores que surjan en las distinas acciones realizadas.

A este fin también se definen las clases de errores de cada elemento en específico. Por ejemplo, se tiene el tipo:

```
data ItemError = ItemErrorNotFound Text
               | ItemErrorNotAllowed Text
               | ItemErrorBadJSON String
               | ItemErrorCategoryNotFound Text
deriving (Eq, Show)
```

Para manejar los errores del tipo **Item**. Por ejemplo, si queremos agregar un item a una categoría que no existe, recibiriemos esta respuesta en formato JSON con un status de 404:

```
{
  "tag": "ItemErrorCategoryNotFound",
  "contents": "principales3"
}
```

Esto es posible gracias a que en el método que busca la categoría por nombre, si la query SQL devuelve un array vacío, levanta un error del tipo `CategoryNotFound` utilizando las funciones mencionadas anteriormente del paquete `Trans/Except` de Scotty:

```
getCategory :: (CategoryRepo m) => Text -> m (Either CategoryError
    Category)
getCategory name = runExceptT $ do
  result <- lift $ findCategoryByName name
  case result of
    [category] -> return category
    _ -> throwError $ CategoryNotFound name
```

Cada **Controller** tiene su propio ErrorHandler que define los status HTTP que debe responder ante cada excepción recibida del servicio, y como mensaje de error expone el nombre del tipo (en este caso `ItemErrorCategoryNotFound`) con un tag opcional, que sirve para clarificar si se quisiera.

# 3. Frontend

Del front end no se hablará mucho ya que el propósito del informe es la explicación del trabajo desarrollado en Haskell. Sin embargo, la funcionalidad de la API se ve reflejada en la utilización de la página web, por lo que sigue siendo un componente importante del trabajo.

El frontend está desarrollado puramente en React. Contiene dos partes principales

## 3.1. Menu

El Menu es lo que los usuarios en el restaurant podrán ver. Contiene un listado de categorías de items y la opción de agregarlos a su orden. Una vez que terminan de decidir, pueden enviar la orden a la cocina para que comience a ser preparada.

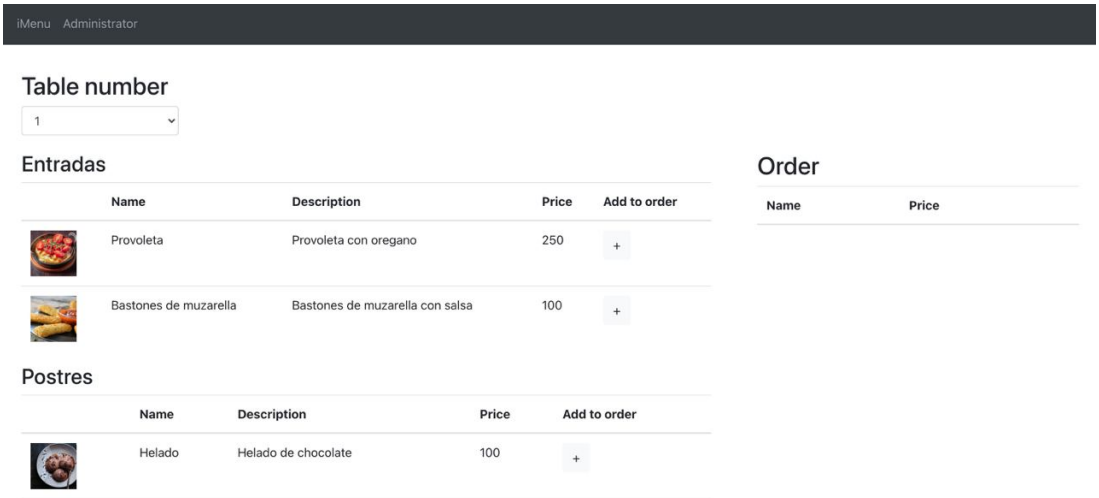


Figura 1: Página principal

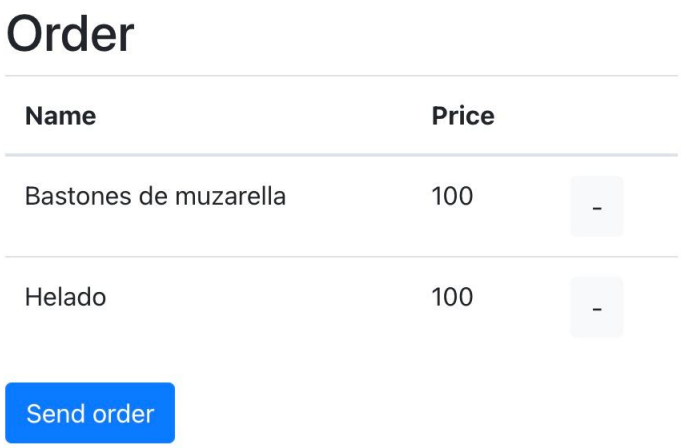


Figura 2: Ordenes

## 3.2. Administrador

Al administrador corresponde a los endpoints del backend marcados con `/admin`. El front end ya tiene integrado el header de autorización necesario para acceder a los endpoints protegidos. Se recomienda que esto deba ser ingresado manualmente.

En el administrador se puede crear un item y controlar y ver las ordenes de los usuarios. La modificación de items y la creación de categorías no están disponibilizadas en el front, se pueden acceder por cUrl.

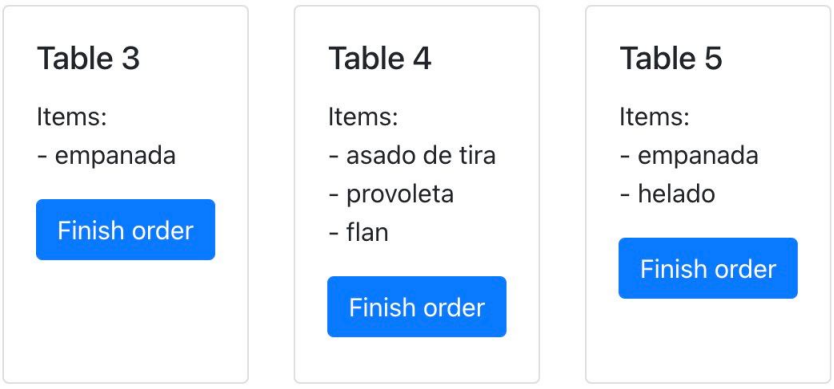


Figura 3: Órdenes por mesa

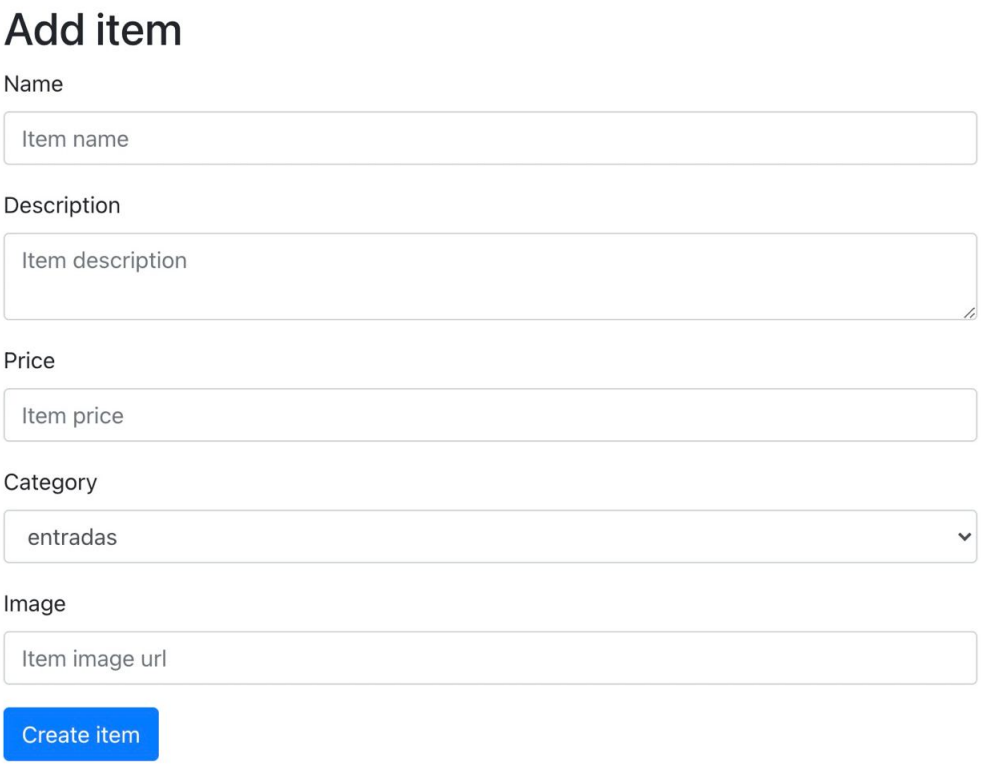


Figura 4: Agregar item

# 4. Mejoras propuestas

Se plantean varias mejoras para una v2 de iMenu:

- **Frontend**: Primero que nada, se debería invertir mucho más tiempo en la implementación de un frontend user-friendly. El provisto es básico para mostrar la funcionalidad de la API.
- **Seguridad**: Como se mencionó más arriba en el informe, se debería agregar algún tipo de seguridad más robusta que la implementada. Un ejemplo podría ser la utilización del módulo jwt de Haskell.
- **Backoffice**: El desarrollo de un nuevo front end debería estar acompañado de una interfaz para el backoffice, que actualmente solo puede ser utilizado mediante cURLs. Con esta implementación se podrían subir imágenes de los items, ya que el type **Item** está preparado para recibirlas.
- **Funcionalidad a agregar**: Se propone agregar funcionalidades tales como la integración con alguna plataforma de cobranza (Mercado Pago, por ejemplo), para poder completar el flujo de la compra en su totalidad. También se podrían incluir avisos del tiempo estimado de la finalización del pedido. En conjunto con esto último entra la propuesta de un campo "estado" para cada elemento **Order**. Así, se pueden marcar como finalizadas las ordenes, y no deben ser eliminadas de la base de datos, ayudando a la trazabilidad de la API.
- **Logging**: Se recomienda agregar un sistema de logging, especialmente para debugging y si se prevee un incremento de usuarios y funcionalidades considerable.
- **Testing**: Se deberían agregar tests tanto unitarios como integrales.

# 5. Conclusión

El trabajo final de la materia me sirvió en muchos aspectos. Principalmente, en cuanto a lo práctico. Es decir, sin bien la materia nos incitó a "jugar" con el lenguaje y las bases del paradigma funcional, fue principalmente teórico todo lo que tuvimos que programar. Enfrentarme a las configuraciones necesarias para levantar un proyecto de una magnitud mayor que lo visto en clase fue al principio frustrante, pero muy educativo.

Aprendí que la comunidad de Haskell está siempre muy dispuesta a ayudar, lo que me motiva a encarar más proyectos en el lenguaje en el futuro.

También me di cuenta mientras desarrollaba que lo que más quería hacer era que la API quedara legible, mantenible, y por sobre todo sencilla. Es algo que logré hacer mucho más fácilmente con Haskell que con cualquier otro lenguaje de programación que haya usado. Esto es lo más importante que me llevo del trabajo. Pude responder la pregunta que me planteé en la introducción: es posible, y recomendable promover el desarrollo de APIs REST en lenguajes funcionales como Haskell. Sobre todo, si la API que se necesita no es muy grande, y se tiene poco tiempo para implementarla. Esta conclusión es algo que me llevaré para mi futuro como ingeniera en informática.