Universidad de Sonora

Ingeniería en sistemas de información

Reporte final de prácticas profesionales

Luis Alberto Durazo Arvizu

209200566

Flights search application

Nearsoft Academy & Talent Incubator

Nearsoft, Inc.

Hermosillo, Sonora, 27 de Febrero de 2017

Índice de contenidos

Índice de contenidos **1**

1. Introducción **2**

2. Descripción del área **2**

3. Justificación **2**

4. Objetivos del proyecto **3**

5. Problemas a resolver **3**

6. Alcances y limitaciones **3**

7. Fundamentos técnicos y teóricos necesarios **4**

7. 1 Tecnologías back-end 4

7.2 Tecnologías front-end 5

7.3 Teorías de ingeniería de software 6

8. Desarrollo de la aplicación **7**

8.1 Estructura de la base de datos y poblado de información 8

8.2 Página de búsqueda: 10

8.3 Controladores (Flujo al servidor) 11

8.4 Página de resultados 12

8.5 Reservaciones 13

8.6 Página de confirmación 14

8.7 Lógica para la extracción de la reservación 15

9. Resultados y retroalimentación **16**

10. Conclusiones **16**

1. Introducción

En mi estancia en Nearsoft en el Talent Incubator, me dieron un proyecto que estaba dividido en diferentes **historias de usuario**[1], las cuales brevemente describen la construcción de una aplicación web en la cual un usuario podía elegir un destino para un vuelo, y la siguiente página lo redireccionará a una página para elegir los vuelos disponibles. Las requisitos técnicos eran que tenía que desarrollar una aplicación web en ember.js y comunicar a través de REST con un servicio de back-end desarrollado en java.

2. Descripción del área

Nearsoft Academy es un programa de Nearsoft, Inc. dedicado a desarrollar el talento de próximos a egresar o recién egresados de carreras universitarias afines al desarrollo de software, el programa tiene una duración de seis meses de 6 a 8 horas diarias, la intención es transformar a estudiantes con habilidades sólidas de lógica y convertirlos en junior developers al final de la estancia. En muchas ocasiones esta formación lleva a la contratación de los 'graduados' del programa dentro de Nearsoft con clientes reales. Gran parte del programa consiste en presentar retos a los estudiantes en forma de proyectos pequeños.

3. Justificación

Mis mentores decidieron que estaba casi listo para entrar a una vacante, por lo cual propusieron que ingresara al programa de Talent Incubator donde realizaría un proyecto de mayor tamaño; para probar mis habilidades, mi efectividad como desarrollador y por último pulirme en los últimos detalles que debía tener listo antes de entrar a una posición de tiempo completo y contratación oficial. Este es ese proyecto.

[1] Una **historia de usuario** es una representación de un requisito escrito en una o dos frases utilizando el lenguaje común del usuario. Las historias de usuario son utilizadas en las metodologías de desarrollo ágiles para la especificación de requisitos (acompañadas de las discusiones con los usuarios y las pruebas de validación).

4. Objetivos del proyecto

El objetivo principal del proyecto era terminar de pulir mi preparación como ingeniero de software, en puntos específicos:

- Habilidad para desarrollar en distintos paradigmas, funcional y orientado a objetos
- Demostrar flexibilidad al momento de cambiar de lenguajes entre javascript y java
- Demostrar conocimiento medio-avanzado del lenguaje Java
- Aprender cómo funcionan los contenedores de servidores, como Tomcat.
- Aprender el funcionamiento de servicios REST
- Aprender y practicar la importancia de pruebas unitarias y test driven development.
- Seguir los lineamientos de código limpio y eficiencia de código.

5. Problemas a resolver

1. Elección de servicio web que proporcione los vuelos.
2. Creación de la base de datos e insertar los aeropuertos para la búsqueda.
3. Creación del servicio web en la parte de back-end con Spring y Java
4. Creación de la aplicación web en la parte de front-end con Ember.js y javascript.
5. Realizar un demo de la aplicación funcional.
6. Realizar con desarrolladores de Nearsoft un análisis del código escrito, y proveer retroalimentación.

6. Alcances y limitaciones

Dado que la aplicación era una aplicación de prueba para soluciones de problemas en el mundo real, la aplicación debía cumplir al pie de la letra con las historias de usuario. Las restricciones técnicas era que solo podía utilizar Java y Javascript para la realización de la aplicación, esto porque al cliente al que quería Nearsoft que yo entrara estaba enfocado a estos dos lenguajes y sus respectivos frameworks (ember/spring).

7. Fundamentos técnicos y teóricos necesarios

7. 1 Tecnologías back-end

Spring MVC: Spring es un framework para el desarrollo de aplicaciones y contenedor de inversión de control, de código abierto para la plataforma Java.

Java 8 (JDK 1.8): Java es un lenguaje de programación orientado objetos, cuya mayor ventaja no es en lenguaje en sí sino el potencial de aprovechamiento de la JVM (Java virtual machine) que ofrece portabilidad ilimitada al lenguaje. La versión 8 del lenguaje fue especificada debido a que contiene muchos cambios para la plataforma de java, incluyendo habilidades seudo-funcionales (gracias a la introducción de expresiones lambda, funciones anónimas ejecutables y contenibles, aunque no con el poder de otros lenguajes funcionales como javascript.) y habilidades para operar mejor en colecciones, por ejemplo el siguiente código:

Dada una colección de Strings, itere sobre ellas e imprima el contenido.

```
public void printContent(List<String> stringList) {
	for(String string: stringList) {
		System.out.println(string);
	}
}
```

Ahora es posible hacerlo de forma más breve:

```
stringList.stream.forEach(s -> system.out.println(s));
```

En el ejemplo anterior se puede apreciar también el uso de lambdas, estos nuevos aspectos del lenguaje son también más rápidos que la forma convencional.

Tomcat: Apache Tomcat funciona como un contenedor de servlets, básicamente será quien contendrá el archivo .war que será el contenedor de la aplicación web de java.

7.2 Tecnologías front-end

Javascript: Es un lenguaje de programación interpretado (que no es compilado sino pre-compilado, después el intérprete se encarga de reorganizar las sentencias lógicas) Se utiliza principalmente en su forma del lado del cliente (client-side), implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas. Para entender más a fondo este lenguaje fue un requisito que leyera el libro 'Javascript: The Good Parts', y se me dio una semana de descanso del proyecto para dedicarme al estudio teórico del libro. A pesar de que javascript tiene el potencial de la programación orientada a objetos, en este proyecto solo se utilizó por sus habilidades de programación funcional. La parte más complicada del proyecto fue javascript, ya que personalmente tenía muy poca experiencia en el lenguaje, a pesar de que el proyecto estaba enfocado a mis habilidades en Java.

Ember.js: Es un framework web para javascript de código abierto, basado en el modelo MVVM (Modelo vista - vista modelo) que ayuda a los desarrolladores a crear aplicaciones web de una sola página escalables.

CSS: Hojas de Estilo en Cascada (Cascading Style Sheets) es el lenguaje utilizado para describir la presentación de documentos HTML, **CSS** describe como debe ser renderizado el elemento estructurado en pantalla, en papel, hablado o en otros medios. Es la forma más común de definir el estilo en navegadores actuales.

HMTL5: La versión de HTML a utilizar fue la 5, a pesar de que el proyecto no requería de ninguna nueva funcionalidad de esta versión, HTML es necesario para el desarrollo de una página o aplicación web. No es un lenguaje de programación por si mismo si no un lenguaje de marcado útil para definir la estructura del documento que contiene los contenidos de la aplicación web.

7.3 Teorías de ingeniería de software

Programación orientada a objetos: Además de los principios básicos de programación orientada objetos, gran parte del conocimiento teórico estaba basado en los cinco principios SOLID, que en español son:

- Principio de responsabilidad única: una clase/entidad sólo debe tener una razón de cambio.
- Principio de abierto/cerrado: establece que una entidad debe quedar abierta para su extensión, pero cerrada para su modificación.
- Principio de sustitución de Liskov: dada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.
- Principio de inversión de dependencia: la noción de que se debe depender de abstracciones, no depender de implementaciones.

Programación funcional: es un paradigma de programación declarativa basado en el uso de funciones matemáticas, en contraste con la programación imperativa, que enfatiza los cambios de estado mediante la mutación de variables.

REST (web-services): en la actualidad se usa en el sentido más amplio para describir cualquier interfaz entre sistemas que utilice directamente HTTP para obtener datos o indicar la ejecución de operaciones sobre los datos (POST, GET, PUT, DELETE, por ejemplo), en cualquier formato (XML, JSON, etc) sin la necesidad de agregar metadata sobre el protocolo de intercambio (SOAP es un ejemplo del caso contrario).

Test Driven Development & Unit Testing: TDD está basado en la noción de que todo código debe ser sujeto a pruebas, y que estas pruebas deben escribirse antes que el código mismo, es principalmente beneficioso porque ayuda al desarrollador a darse una idea más concreta de lo que está por diseñar (o diseñando), por medio de pruebas unitarias fue la limitante para este proyecto, no era necesario hacer pruebas de integración.

8. Desarrollo de la aplicación

El primer paso para el desarrollo de la aplicación fue elegir al proveedor de los vuelos, para este caso tenía libre elección pero decidí utilizar la API de Google QPX, que regresa los vuelos en formato json al realizar una petición a su API por medio de distintos endpoints.

Después de elegir al proveedor tuve que diseñar cómo funciona la aplicación. La cual seguiría el modelo mostrado en la figura 8.0.1

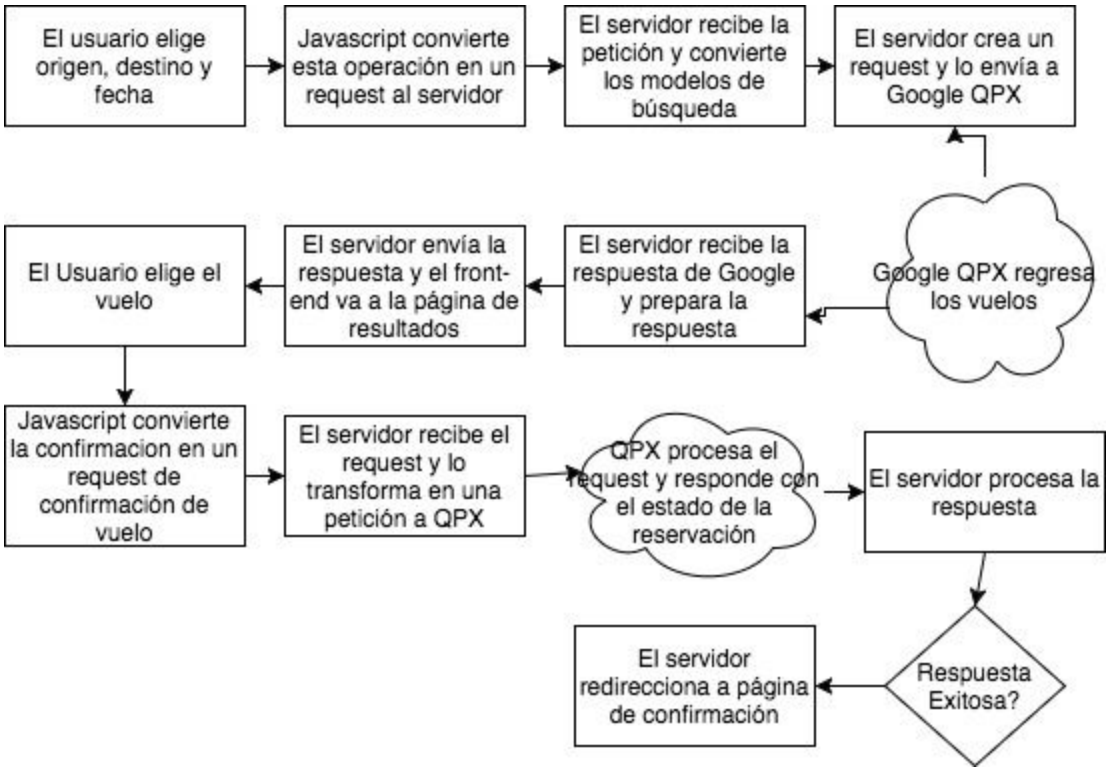

Figura 8.0.1, diagrama de diseño y estimado

Otro de los requisitos era que la aplicación contara con autocompletado, esto presento un nuevo completamente distinto ya que requería que la aplicación tuviera de antemano la información de todos los aeropuertos del mundo, para resolver este caso bajé todos los datos de aeropuertos desde flightstats. Y escribí un script que solo se ejecuta una vez para probar las bases de datos, las cuales se presentarán a continuación:

8.1 Estructura de la base de datos y poblado de información

La base de datos contaba con tres tablas descritas a continuación junto con la instrucción de creación en la Figura 8.1.1

Aeropuertos:

- ID (Serial, primary key)
- IATA code (Varchar) Representa un código de 3 letras, i.e. (MEX,SFO,HMO)
- Latitud (Varchar)
- Longitud (Varchar)
- Ciudad (Varchar)

Reservaciones:

- ID (Serial, primary key)
- Name (Varchar)
- Last Name (Varchar)
- Passengers (SmallInt)
- Cost (Varchar)
- Email (Varchar)

Vuelos:

- ID (Serial, Primary key)
- Departure date (Varchar)
- Arrival date (Varchar)
- Departure Airport (Varchar)
- Arrival Airport (Varchar)
- Reservation ID (Integer, references reservations ID)

```
CREATE TABLE IF NOT EXISTS AIRPORTS( id SERIAL PRIMARY KEY, iata VARCHAR(255), lat VARCHAR(255), lon VARCHAR(255), name VARCHAR(255), city VARCHAR(255) );
CREATE TABLE IF NOT EXISTS RESERVATIONS( id SERIAL PRIMARY KEY, name VARCHAR(255), last_name VARCHAR(255), passengers SMALLINT, cost VARCHAR(255), email VARCHAR(255) );
CREATE TABLE IF NOT EXISTS FLIGHTS( id SERIAL PRIMARY KEY, departure_date VARCHAR(255), arrival_date VARCHAR(255), departure_airport VARCHAR(255), arrival_airport VARCHAR(255), reservation_id INTEGER references RESERVATIONS(id) );
```

Figura 8.1.1 Sentencias SQL para creación de tablas

Para poblar la información de los aeropuertos por única vez, cree el script mostrado en Figura 8.1.2 que traerá todos los aeropuertos y los llenará a la base de datos:

```
require 'rubygems'
require 'json'
require 'rest-client'
require 'uri'
require 'pg'

begin
client = PG::Connection.open(:host => "localhost", :user => "root", :password => "root" ,
:dbname => "flightsdb")

url='https://api.flightstats.com/flex/airports/rest/v1/json/all?appId=e9bf38a8&appKey=75a5f310
0db3ee5e2b7fc4bdb5fa222f'
  airportsJson = JSON.parse(RestClient.get(url))
  for airports in airportsJson["airports"]
          iata = airports["iata"]
          latitude = airports["latitude"]
          longitude = airports["longitude"]
          name = airports["name"]
          name = name.gsub("'", %q(_))
          city = airports["city"]
          city = city.gsub("'", %q(_))
              client.query("INSERT INTO AIRPORTS VALUES(DEFAULT, '#{iata}' , '#{latitude}' ,
'#{longitude}', '#{name}', '#{city}')")
  end
  print "loading finished"
```

Figura 8.1.2, ruby script para aeropuertos

Después de resolver el problema de la base de datos y el llenado de los aeropuertos, el siguiente paso era crear la parte que se encarga de la página de búsqueda.

8.2 Página de búsqueda:

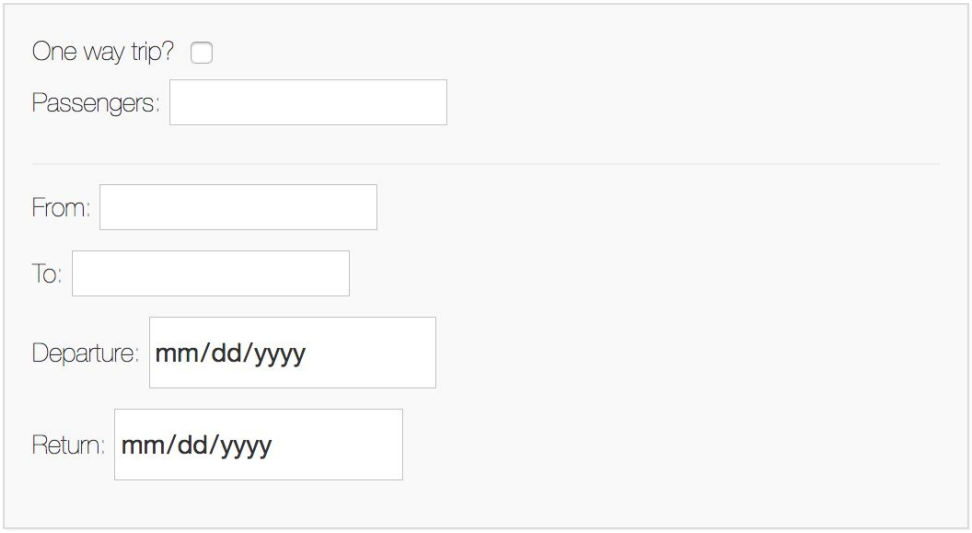

Figura 8.2.1 página de búsqueda

```
<div class="col-md-3" id="main-form">
        <form {{action "searchFlights" on="submit"}} class="input-edit">
            <label>One way trip?</label>
            {{view App.TripCheck action='tripType' checked=isOneWay}}
            <label>Passengers:</label>
            {{view Ember.Select content=passengerslimit valueBinding = "passengersNumber"}}
            <hr />
            <label>From:</label>
             {{input type="text" class="form-control" placeholder="Departure city" valueBinding
= "departureCity" id="w-departure-search"}}
            <label>To:</label>
             {{input type="text" class="form-control" placeholder="Arrival city" valueBinding =
"arrivalCity" id="w-arrival-search"}}
            <div class="row">
                <div class="col-md-6">
                    <label>Departure:</label>
                     {{input type="date" class="form-control" id="departure-date" valueBinding
= "departureDate"}}
                </div>
                <div class="col-md-6">
                    <label>Return:</label>
                     {{input type="date" class="form-control" id="returning-date" valueBinding
= "returnDate" disabled=isOneWay}}
                </div>
            </div>
            {{input type="submit" value="Search flights" class="btn" id="search-btn"}}
        </form>
    </div>
```

Figura 8.2.2 Código .hbs para la página de búsqueda

La Figura 8.2.1 representa la página de búsqueda, y la Figura 8.2.2 HTML junto con el código que no es html representa a los handlebars de ember, los cuales representan contenido dinámico que dependerá de los controladores de Ember.js

8.3 Controladores (Flujo al servidor)

Una vez que el usuario entra su búsqueda, javascript admitirá la forma y esta será recibida por un controlador mostrado en Figura 8.3.1.

```
@RequestMapping(value="/search", produces="application/json", method = RequestMethod.POST,
headers = "Content-Type=application/json")
   public @ResponseBody ResponseEntity<String> getAvailableFlights(@RequestBody SearchRequest
searchRequest) {
String response = flightsService.getAvailableFlights(searchRequest, FlightConnector.QPX);
       return new ResponseEntity<String>(response, HttpStatus.OK);
   }
```

Figura 8.3.1 código del controlador de Spring para búsqueda

Este controlador mandará llamar a un servicio que se encargará de orquestar la comunicación entre el controlador y el conector del servicio de QPX en Figura 8.3.2

```
    public String getAvailableFlights(SearchRequest searchRequest, String connectorName){
        searchRequest.setSolutions(FlightsAppConstants.NUMBER_OF_RESULTTS);
        FlightConnector connector = connectorFactory.createConnector(connectorName);
        List list = connector.getFlights(searchRequest);
        try {
            return objectMapper.writeValueAsString(list);
        } catch (IOException e) {
            e.printStackTrace();
            return "No flights found";
        } }
```

Figura 8.3.2 Código del servicio de Spring para búsqueda

El código del conector que se comunicará con el servicio de Google QPX, Figura 8.3.3

```
 @Override
    public List<TripOption> getFlights(SearchRequest searchRequest) {
        try {
            String jsonStringForRequest = objectMapper.writeValueAsString(new
QpxRequest(searchRequest));
            String qpxResponse = QpxRestOperator.getFlightsFromQpx(jsonStringForRequest);
            JsonNode node = objectMapper.readTree(qpxResponse);
            node = node.get("trips").get("tripOption");
            if(node == null){
                return null;
            }
            return objectMapper.readValue(node,
objectMapper.getTypeFactory().constructCollectionType(List.class, TripOption.class));
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
```

Figura 8.3.3 código del conector para búsqueda

Este servicio funciona convirtiendo el modelo SearchRequest a JSON, y envía el request a QPX, convierte su respuesta a un Java Object y la enviamos al frontend.

8.4 Página de resultados

A continuación en Figura 8.4.1 se presenta la página de resultados al usuario llenar correctamente el formulario de búsqueda.

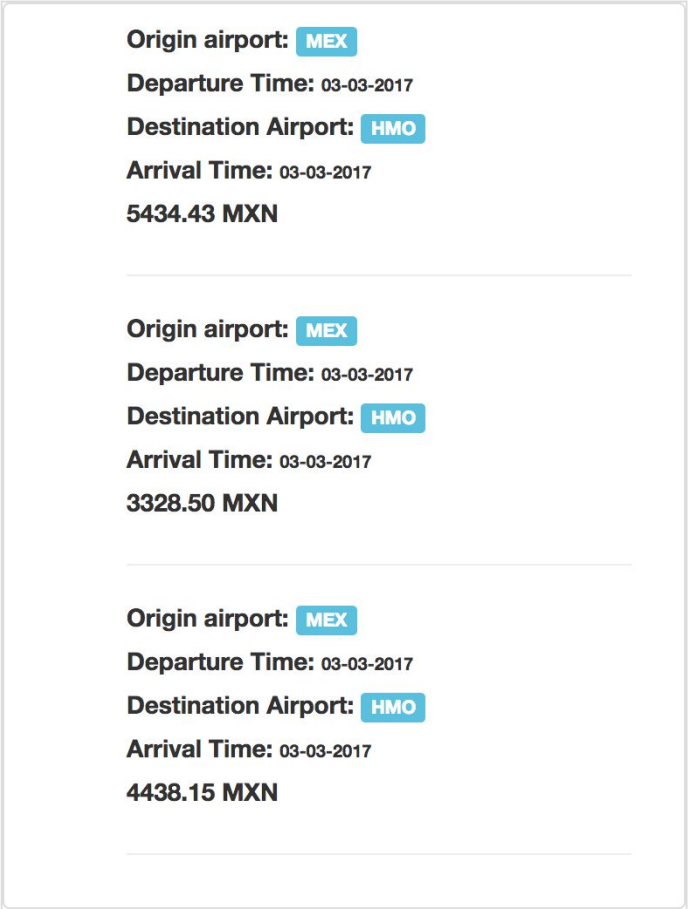

Figura 8.4.1 página de resultados

Si el usuario da click a uno de los elementos que contienen los vuelos, se tomará como seleccionado y lo enviará a otra página donde mostrará el mismo formato, pero para los vuelos de regreso (siempre y cuando sea un vuelo redondo)

Por motivos de brevedad en el documento del reporte no se mostrará el código fuente de la página, sin embargo es accesible desde esta URL:

https://github.com/ldurazo/FlightsApp-Ember-Spring/blob/master/Project/src/main/webapp/client-side-ember/index.html

8.5 Reservaciones

Para realizar las reservaciones al elegir un vuelo, tenemos que enviar a Google QPX un request de reservación y después guardar esta información en la base de datos, esto sucede a través del ReservationController Figura 8.5.1

```
    @RequestMapping(value = "/reservation", method = RequestMethod.POST, headers =
"Content-Type=application/json")
    public ResponseEntity<String> createReservation(@RequestBody Reservation reservation) {
        String response = reservationService.getReservationCustomResponse(reservation);
        return new ResponseEntity<String>(response, HttpStatus.OK);
    }
```

Figura 8.5.1 código del controlador de spring para reservaciones

Y de la misma forma que en la búsqueda, se comunicará a un servicio que se encargará de realizar tanto el request como la operación de guardado a la base de datos en Figura 8.5.2

```
    public String getReservationCustomResponse(Reservation reservation) {
        CustomResponse response = new CustomResponse();
        try {
            int reservationId = saveReservation(reservation);
            response.setMessage("your reservation id is: " + reservationId);
        } catch (FlightsAppException e) {
            response.setMessage(e.getLocalizedMessage());
        }
        return objectMapper.objectToJson(response);
    }

    public int saveReservation(Reservation reservation) throws FlightsAppException {
        int reservationId = reservationDao.save(reservation);
        if(reservationId < 1){
            throw new FlightsAppException();
        }
        return reservationId;
    }
```

Figura 8.5.2 código del servicio de reservaciones

Una vez guardada la reservación, el usuario podrá recuperar su reservación por medio del email y el número de confirmación.

8.6 Página de confirmación

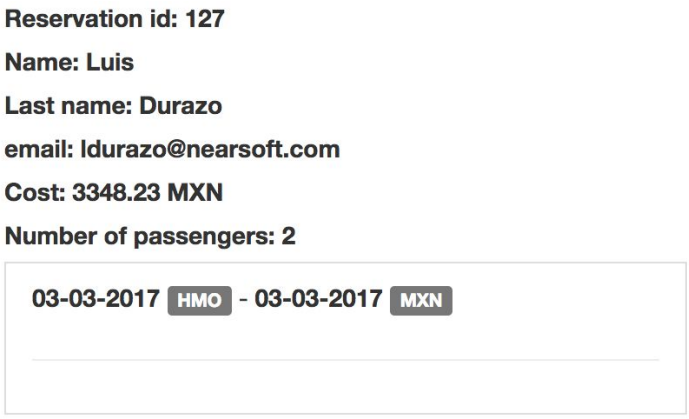

Figura 8.6.1 página de confirmación

La página de confirmación (Figura 8.6.1) utiliza el siguiente handlebar (Ember HTML file), que es fácil de explicar (Figura 8.6.2):

```
<label>Reservation id:</label>
<label>{{id}}</label>
<br/>
<label>Name:</label>
<label>{{name}}</label>
<br/>
<label>Last name:</label>
<label>{{last_name}}</label>
<br/>
<label>email:</label>
<label>{{email}}</label>
<br/>
<label>Cost:</label>
<label>{{cost}}</label>
<br/>
<label>Number of passengers:</label>
<label>{{passengers}}</label>
        <li class="list-group-item">
                {{#each flights}}
                    <label>{{departureTime}}</label>
                    <label class="label label-default">{{origin}}</label> -
                    <label>{{arrivalTime}}</label>
                    <label class="label label-default">{{destination}}</label>
                    <hr/>
                {{else}}
                    <label>No reservation found</label>
                {{/each}}
        </li>
```

Figura 8.6.2 archivo .hbs de reservación

Ember creará elementos HTML basados en cada uno de los contenido dinámicos, por ejemplo {{last_name}} está basado en un modelo de javascript, que ember tomará y lo convertirá en el texto que contiene el modelo, como en el ejemplo "Durazo" al igual que los elementos {{each}} tomará cada miembro en la lista y creará un elemento lista para cada uno.

8.7 Lógica para la extracción de la reservación

Para extraer la reservación, mostraré el código (figura 8.7.1) que se encarga de traerlo de la base de datos, que es básicamente una conexión directa por medio de jdbc que lo convierte a un objeto de Java, que posteriormente el controller convertirá en un json response para que Ember lo renderée como HTML (como se observó en figura 8.6.1)

```
    public Reservation getRecord(final int id, final String email) {
        try {
            String selectFromReservationsQuery = "SELECT * FROM RESERVATIONS WHERE ID=? AND
EMAIL=?";
            Reservation reservation = jdbcTemplate.queryForObject(selectFromReservationsQuery,new
Object[]{id,email}, new ReservationRowMapper());
            if(reservation!=null){
                String selectFromFlightsQuery = "SELECT * FROM FLIGHTS WHERE reservation_id=?";
                List<Flight> flightList = jdbcTemplate.query(selectFromFlightsQuery,new Object[]{id},
new FlightsRowMapper());
                reservation.setFlights(flightList);
            }
            return reservation;
        } catch (EmptyResultDataAccessException e){
            return new Reservation();
        }
    }
```

Figura 8.7.1 código del repositorio para acceso de base de datos

El método de arriba en su mayoría solo delega a sus dependencias el trabajo, son las clases de abajo las que serán llamadas desde queryForObject que convertirá el resultset en java objects (Figura 8.7.2)

```
    public class ReservationRowMapper implements RowMapper<Reservation> {
        @Override
        public Reservation mapRow(ResultSet resultSet, int rowNumber) throws SQLException {
            Reservation reservation = new Reservation();
            reservation.setId(resultSet.getInt(Reservation.ID_COLUMN));
            reservation.setName(resultSet.getString(Reservation.NAME_COLUMN));
            reservation.setLast_name(resultSet.getString(Reservation.LAST_NAME_COLUMN));
            reservation.setPassengers(resultSet.getInt(Reservation.PASSENGERS_COLUMN));
            reservation.setCost(resultSet.getString(Reservation.COST_COLUMN));
            reservation.setEmail(resultSet.getString(Reservation.EMAIL_COLUMN));
            return reservation;
        }
    }

    public class FlightsRowMapper implements RowMapper<Flight> {
        @Override
        public Flight mapRow(ResultSet resultSet, int rowNumber) throws SQLException {
            Flight flight = new Flight();
            flight.setId(resultSet.getInt(Flight.ID_COLUMN));
            flight.setDestination(resultSet.getString(Flight.ARRIVAL_AIRPORT_COLUMN));
            flight.setOrigin(resultSet.getString(Flight.DEPARTURE_AIRPORT_COLUMN));
            flight.setArrivalTime(resultSet.getString(Flight.ARRIVAL_DATE_COLUMN));
            flight.setDepartureTime(resultSet.getString(Flight.DEPARTURE_DATE_COLUMN));
            flight.setReservation_id(resultSet.getInt(Flight.RESERVATION_ID_COLUMN));
            return flight;
        }
    }
```

Figura 8.7.2 código de mapeo de resultset a java object

9. Resultados y retroalimentación

Aclaraciones iniciales: gran partes de este proyecto no fueron mencionadas, como ciertas clases que son parte del proyecto, en este documento solo se generalizan las ideas y las partes importantes de la aplicación sin entrar a detalle, sin embargo la aplicación completa se encuentra en un repositorio de código abierta en esta dirección: https://github.com/ldurazo/FlightsApp-Ember-Spring

El código es completamente ejecutable, con la única limitante de quién descarga el código debe dar de alta una cuenta en flightstats y Google QPX y reemplazar los application id's con los generados, y también crear las bases de datos y correr el script inicial.

Al final de este proyecto, hice un demo y una revisión de código con mis mentores, los cuales me dieron notas importantes, una de ellas fue que hice muy pocas pruebas unitarias, que son parte fundamental de la ingeniería de software. En general fue un proyecto exitoso y me abrió las puertas para conseguir una vacante de tiempo completo en Nearsoft.

10. Conclusiones

Las aplicaciones prácticas, aún cuando no son implementadas en un entorno real de producción, con la correcta mentoría y supervisión son excelentes instrumentos para comprender cómo funcionan las herramientas en el mundo real, y fue mi primer contacto con un proyecto de tamaño pequeño aunque real, este proyecto bien pudo ser la versión alfa de un proyecto totalmente viable para ventas y me abrió las puertas para entender el funcionamiento de las grandes aplicaciones donde hay más de 100 ingenieros trabajando al mismo tiempo, y en el que me encuentro actualmente. La oportunidad que me dio Nearsoft fue un 'jumpstart' para mi carrera como ingeniero de software y de sistemas de información.