Crossover Tech Trial Project

Air Ticket Reservation System

Software Design Document

Erdem Günay

Change History	Actor	
21/08/2016	Erdem Günay	Initial draft
22/08/2016	Erdem Günay	Enhanced design details

1. Introduction

Purpose

The software design document describes the architecture and system design of an Air Ticket Reservation system. This document is intended for team members who might participate in the project, understand the requirements, architecture and the design of the system and contribute in the project.

Scope

Air Ticket Reservation system allows public users login with their social network accounts (currently Google+ only, but the other ones can easily be supported), and reserve and buy air tickets for available flights, online check in before the flight and print their tickets.

Staff users login the system with their own private credentials and manage the system, and do the operational work like canceling tickets, etc.

Overview

This document is organised in a way to provide information about

- ✓ Requirements
- ✓ Data model
- ✓ System architecture overview
- ✓ Component design
- ✓ Human interface design
- ✓ Future enhancements

2. Requirement Analysis

Requirements are analysed and grouped in different categories by relevant use case scenarios and technical needs.

Requirements for Public user use cases

Public users shall be able to do following actions and get services

1. Login with their Google+ account without entering any private credentials on our system. If there is not an active session, Google may ask the user to enter credentials but this interaction has nothing to do with our system
2. Users will not be able to get any service, any information until logging in
3. After logging in, the user can search available flights according to different parameters (e.g. destination from, to, flight date, etc.).
4. The system should return the matching available flights with flight information including scheduled flight date
5. The user can select a flight and continue with booking. At booking step, the user shall provide the number of passengers and do reservation.
6. A booking reservation record is created for the user
7. The user must complete reservation by credit card payment. Currently only credit card payment method is supported in the scope of this phase. Credit card payments are not really provisioned and the user will not be charged any thing in the scope of this project.
8. During payment, the user should provide passenger information as well (e.g. name, surname, email, phone no, etc.)
9. Upon a successful payment, the system should generate tickets (as many as the number of passengers), and send an e-mail to the user with the flight and ticket information.
10. In the email, there will be links that the user can follow to reach "My Bookings", "Online Checkin", "Cancel booking" services.
11. The users can log in any time and see their bookings (past or future). In the "My Bookings" page, the user interface screen should guide the user for possible actions. For example, if a payment is not completed for a reservation, only payment button should be visible. If the payment is done but check in is not done, the user should see check in button. If there is checked in ticket in the booking, print ticket button should be visible. At the same time both check in and print ticket buttons can be visible in case some (but not all) of the tickets are checked in.
12. The user can check in by selecting a seat from the plane within 48 hours to 4 hours time frame to the flight. Before 48 hours and after 4 hours to the flight, check in is not possible
13. The user can print the tickets after checking in

Requirements for Staff user use cases

Staff users shall be able to do following actions and get services

1. Login with their username / passwords that are stored in our system. For the sake of simplicity due to time considerations and ease of demonstration, usernames and passwords will be predefined at this phase. As stated in the future enhancements section, in an ideal solution, there should be a more complex user management features like provisioning new users, users changing their passwords, lost password reset, captcha, etc.
2. Staff users have two different roles, administrative role, operations role.
3. Administrative staff user has rights to change systems master catalogue data like planes, destinations available, flight information between destinations and scheduled flight instances.
4. Operations staff can receive reservation chart 1 hours from departure
5. Operations staff can cancel any ticket before departure, generating email alert and refund to the public user
6. Operations staff can cancel the whole flight generating email alerts and refunds to the public users

Requirements for System use cases

The system should do following actions

1. Allow integration of new clients (probably mobile clients or TV, STB clients) without making ANY SINGLE LINE OF CODE change.
2. Generate email alerts 48 hours from departure regarding the users reservation to encourage them for early check in. The email should contain "Check in now" link that will not require the user to login.
3. Execute properly without any problems on cloud environment

3. Data Model

Data model is designed to have following major data domains

1. User domain: responsible for user, passenger, credit card information.

2. Flight domain : responsible for catalogue information that is used to compose a complete flight scheduling table. The main entities in this domain are Plane, Destination, FlightInfo, FlightSchedule. For example, in order to sell air tickets, following should be defined
    1. the destinations (cities to fly from / to) should be defined
    2. planes should be defined with their max capacity, and seat configuration
    3. a new flight information should be defined between two destinations,
    4. a new flight schedule (flight information instance) should be defined combining the plane, flight information and the scheduled flight date.
    5. Then, the users can start booking,
3. Booking domain : responsible for managing the user bookings, payments, checkins, ticketing, etc.

User Domain Model

img/cd_user_domain.png

Flight Domain Model

img/cd_flight_domain.png

Booking Domain Model

img/cd_booking_domain.png

4. Overview of System Architecture

The system is designed mainly to be a scaleable multi tiered, multi component WEB application that can be executed on a cloud environment.

Following main third party tools are considered to form the framework of the solution

1. Java 8
2. Maven 3.3+

3. MySql 5.6+
4. Spring Boot 1.4.0
5. Spring Framework 4.3.2
6. Spring Data JPA 1.10.2
7. Hibernate 5.0.9 (behind the scenes, through spring data jpa)
8. Spring Security 4.1.1
9. Spring Mail
10. AngularJS 1.5.8
11. Bootstrap 3.3.6
12. Unify 1.9.5 (purchased)
13. JasperReports 6.2.0

The system is designed as a multi module Maven project. The parent artifact information is as follows:

```
<groupId>com.crossover</groupId>
<artifactId>techtrial</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>pom</packaging>
```

Maven multi module structure is as follows, the indentation shows the parent - child relationship

```
<artifactId>techtrial</artifactId>
     <artifactId>techtrial-domain</artifactId>
     <artifactId>techtrial-api</artifactId>
     <artifactId>techtrial-admin</artifactId>
```

5. Component design

The system is designed to be composed of following components

1. Parent component
2. Domain component
3. Api component
4. Admin component

Common design principles

1. Spring beans and dependency injections will be defined as Java Bean configuration (e.g. @Component, @Service, @Bean, @Autowired).
2. No spring application context will be configured in XML. Even both have pro's and con's I prefer Java Bean configuration over XML configuration because it's much simpler, easier to implement and especially maintain after some time passes and the developers start to forget the details.
3. No web.xml and dispatcher servlet configuration will be used. Spring Boot perfectly handles much of the overhead that the developer had to do before in order to be able to deploy and start a web application before. Now it's as simple as running or debugging a standalone Java Main class to start Tomcat and deploy your application on your local environment.

```
@SpringBootApplication
public class ApiApplicationMain {

        public ApiApplicationMain() {
        }

        public static void main(String[] args) {

           SpringApplication.run(ApiApplicationMain.class);
        }

}
```

Parent Component (techtrial)

It is just a container component used to manage maven dependencies and common configurations for the other components.

For example, spring boot parent version therefore the versions of all spring and related third party components are defined here, all the other components share this definition.

For details, please refer to *techtrial/pom.xml* file

Domain Component (techtrial-domain)

This component is responsible for providing domain model entities, domain repositories and domain services for the rest of the system.

Domain component provides these entity, repository and service beans for the other components (currently techtrial-api and techtrial-admin). In the future, depending on the new requirements, if a new component is needed, the domain component provides all of the domain related services to the new component as well without spending any other effort to make these features accessible.

Some of the design principles in the domain component are as follows

1. Only JPA annotations should be used. Even if Hibernate is used behind the scenes, we will stick to JPA standard in order not to get tight dependency to Hibernate. For example;

   ```
   import javax.persistence.GeneratedValue;
   import javax.persistence.GenerationType;
   import javax.persistence.Id;
   import javax.persistence.MappedSuperclass;
   import javax.persistence.Entity;
   import javax.persistence.EnumType;
   import javax.persistence.Enumerated;
   import javax.persistence.ManyToOne;
   import javax.persistence.OneToMany;
   ```

2. Entity inheritance hierarchy should be used to eliminate duplicate attribute definition for common attributes among different entities.

   ```
   @MappedSuperclass
   abstract public class AbstractBaseEntity {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private Long id;
   ```

3. Repository interfaces should be used. The repository interfaces are just interfaces that do not require the developer to implement all the traditional DAO interfaces and implementation classes. The spring data jpa project handles auto generation of the burden SQL, Criteria generation and uses Hibernate behind the scenes for ORM operations. The developer only needs to concentrate on the business requirements and properly name repository methods so that the SQL query can be properly generated. This approach saves a considerable amount of time for the developer and

prevents potential errors that can happen and can not be detected until runtime due to typos, copy/paste errors and data model changes.

```
import org.springframework.data.repository.CrudRepository;
import com.crossover.techtrial.domain.model.user.User;

public interface UserRepository extends CrudRepository<User, Long> {

      public User findByUsername(String username);

}
```

4. Custom queries are still supported together with pagination and sorting support. As long as not required, please do not use NATIVE SQL queries here,

```
@Query(value = "select fs from FlightSchedule fs "
 + " where fs.flightInfo.from = ?1 "
 + "   and fs.flightInfo.to = ?2 "
 + "   and fs.plane.maxCapacity > (select count(*) from FlightTicket
t where t.flightBooking.flightSchedule = fs) "
 + "   and fs.scheduledDate between ?3 and ?4"
)
public Page<FlightSchedule> searchFlights(Destination from,
Destination to, Date startDate, Date endDate, Pageable pageRequest);
```

5. The methods that are updating data and performing more than one operation should be annotated with @Transaction so that the update operations can be handled consistently.

```
import org.springframework.transaction.annotation.Transactional;

@Transactional(propagation=Propagation.REQUIRES_NEW)
public Iterable<FlightSchedule> addNewFlightSchedules(…) {
```

6. In order to prevent infinite JSON deserialisation problem, the relationship on the child entities should be marked with @JsonIgnore annotation.

```
@ManyToOne
@JsonIgnore
private FlightBooking flightBooking;
```

API Component (techtrial-domain-api)

This component is responsible for providing API services for the client applications currently being Public User portal.

1. In the future a new client can be immediately integrated without changing a single line of code in this project. All of the APIs are provided as pure JSON request / response methods.
2. MVC (not spring-mvc) design pattern is used.
    1. M : Domain Model entities
    2. V : single page application written in AngularJS is the view
    3. C : REST Controllers
3. Singleton design pattern is heavily used as Spring beans by default. This brings a great flexibility to auto wire and inject dependencies wherever a bean is needed in the application. In order to guarantee thread safety, no state information is stored in these Singleton Spring beans what so ever.
4. If Spring Boot out of the box configures and starts some feature, we should stick to use that instead of trying to configure the application context ourselves. This approach brings couple of advantages mainly saving time and smoother upgrading from older versions to newer versions.
5. Except for the static html resources (html, js, css, img) all of the API services are secured. Only the requests from authenticated users are allowed and only the user's own information can be processed. At the query level, the User is passed as a parameter to make sure that no other user's information can be queried by other user's requests.

```
    /**
     * find one booking with Id and User that the booking entity 
belongs to
     *
     * @param id
     * @param user
     * @return
     */
    public FlightBooking findByIdAndUser(Long id, User user);

    /**
     * Find all booking entities that belong to given user
     *
```

```
 * @param user
 * @return
 */
public List<FlightBooking> findAllByUser(User user);
```

6. Currently only Google+ integration is supported in the scope of this project. However, the other social network applications can also be used for signing. It's just a matter of time and effort. The important thing here is abstracting the integration details of the authentication provider application from the core of our application. The secured APIs should never know if the user is authenticated from Google+ or Twitter or Linkedin, etc.
7. REST return HTTP messages and status codes should be handled properly. Following global exception handler can be used for this purpose. HTTP error messages should be externalised and internationalised.

```
/**
 * Global exception handler that can be used to customise
error messages
 * and HTTP status codes before returning the response to
the client
 *
 * @author egunay
 *
 */
@ControllerAdvice
public class GlobalExceptionHandler {
```

8. REST controllers should delegate the requested action to domain repositories for fetching data from database and to domain services for custom transactional data manipulation operations.
9. In order to prevent pollution of JSON entity deserialisation and impact performance badly, in the REST controller level, irrelevant unnecessary child entity relationships should be set to NULL so that those relationships are not traversed with lazy initialisation. It can be discussed to use DTO model here and I can see that some of the technical people argue that DTO usage is a must. However, I prefer not to use DTOs because it brings a huge overhead to manage implementing and maintaining all those DTO classes and mappers, converters, etc. So I prefer to stick to using the original domain model as the Model of the MVC pattern.

10. For the sake of simplicity (due to time limitation) and the ease of demonstrating, the public user portal application is implemented as part of the API project. The best approach would be separating the static resources (html, js, css, img) from the techtrial-api component and put them in a separate component (probably called, techtrial-web or techtrial-portal). These static resources should be served by a web server (nginx, apache, etc.). This brings also performance improvements as the web servers can manage caching static resources as well as serving. An other impact is that one layer in the tiers will be missing while serving the static content, also reducing the load on the application server.

Admin Component (`techtrial-domain-api`)

Tha same rules apply to Admin component as well as API component. They have the same principles, same structure, same technologies. I have decided to separate Admin component from API component considering two main points

1. Security. Administrative operations require administrative services like defining the master catalogue (planes, destinations, flights, etc.) and performing operations on public user records (cancelling ticket, the whole flight, etc.). It's much safer to keep these powerful services out of public user portal's access even if the services are secured by user authentication.
2. The dynamics and requirements of public user portal and staff user portal are different. So by separating these two as different deployables, we can consider applying different scaleability and deployment approaches according to the current and possibly changing requirements in the future.
3. An other main consideration is that some background operations like sending mails to users when flight time approaches requires additional resource consumption. So by separating such features from public user portal we can assure that public user portal resources are not consumed by these background jobs and it keeps serving public users.

6. Human Interface Design

Please refer to video Part1.mp4 where the user interface design and screen flows are explained.

7. Future Enhancements

There are various points where the Air Ticket Reservation system can be enhanced from. These enhancement points can be categorised in following groups

1. Business enhancements
    1. Different fares can be applied for the seats in the same flight. The earlier the users buy tickets, the cheaper prices they get
    2. Passengers can be categorised as baby, child, adult, or elderly. Different passenger categories might get discounts
    3. Previous passenger records can be returned and the user can select from these records instead of entering the same passenger information again & again. Current API supports that, only fronted should be adapted.
    4. Pricing should be enhanced to include other costs like taxes etc
    5. Promotions can be applied to attract more people
    6. Integrations and cooperation with different services (like credit cards, club memberships, etc.) can be considered to keep passengers buying more tickets for cheaper prices
    7. Frequent flyer or loyalty programs can be applied to keep the passengers fly more frequently and remain loyal
    8. The user ticketing and travelling behaviour can be analysed and potential campaigns can be offered to segmented users. For example in summer time, if a user is probably known to travel every 2-3 weekends, he/she will probably be interested in a good travel offer to similar destinations.
    9. Return trip booking shall be supported
    10. Flexible date (+ / - some days) shall be supported. Current API supports that, only fronted should be adapted.
    11. Real credit card payment integration should be implemented. Currently if the credit card number is composed of digits, the payment is assumed to be successful.
    12. Other payment methods shall be integrated such as PayPal, frequent flyer miles, etc
    13. User authentication token shall be saved on client’s local storage so that the users do not have to login every time they come back.
2. User experience enhancements.

    1. To be honest, this is my first project where I have implemented end user application. Conceptually I had an idea about how to place the technologies and where to use them, but hands on user experience development is something different. Therefore I am sure there are plenty of enhancement points that can be done. Here I will list couple of those points
    2. Visual artifacts and design on email and report templates must be improved. It just requires design capability and time.
    3. Date time formatting should be standardised throughout the GUI
    4. Message internationalisation both on client and server side and error message localisation and handling must be enhanced. It just requires time, it's a very straight forward task.
3. Technical enhancements
    1. **https** communication over **ssl** between client and backend tiers should be established in order to secure the data.
    2. a load balancer should be placed in front of application service instances in order to scale up and down.
    3. a web server (my preference is nginx because it's quite lightweight, easy to configure and execute) should be placed and static contents (html, js, css, images) should be served by web server
    4. an asynchronous persistent, transaction messaging queue (Pivotal's RabbitMQ, or Apache Kafka, etc.) should be used to prevent critical operation from failures like email delivery requirement in this project.
    5. Logging should be implemented. It's just a matter of time, there is no big magic or challenge. In the project org.slf4j.Logger used in several places.
    6. Due to lack of time, admin project is not finished completely, its API's are mostly ready, spring security integration and user interface development are missing. Spring security has a feature called UserDetailsService that needs to be implemented to authenticate username & password from any data source and load user details with user roles. Once a user is authenticated and valid jSession id is generated, the clients can make secure API calls with this jSessionid. Spring security will interpret the secure API calls,

validate if the users jSessionid is valid and and the user is authorised to access the resource.

7. the admin project (staff user api & portal) can be tested from Postman. The request collection can be obtained from this url : https://www.getpostman.com/collections/791edb73614e0511cb62