Component-Based Flight Booking System with Enterprise JavaBeans 3.0

Author: Max Hyde

Supervisor: Dr. Kung-Kiu Lau

2nd May 2007
Component-Based Flight Booking System with Enterprise JavaBeans 3.0

Author: Max Hyde

Supervisor: Dr. Kung-Kiu Lau

2nd May 2007

Abstract

Component-Based Software Development involves composing reusable pieces of code to form a complete system. Enterprise JavaBeans 3.0 provides a specification and set of interfaces suited to this development style.

The objective of the project is the focus of this report: to implement a flight booking system using the EJB 3.0 framework and associated components. In order to achieve this the main features of the technology must be learnt and understood. A selection of the advanced concepts should be developed.

All aims were achieved to a large extent. The solution included server-side EJB components – designed and implemented in such a way as to allow maximum use of EJB 3.0 features – that were reused in a customer-oriented web client and a Java application administration client. The test results prove that the individual components work and that the overall system is functional.

The project was an overall success, with the author taking away a firm understanding of the EJB technology and many advanced features. The project offered experience of building two complete applications – each on a different platform and using different technologies – from scratch.
Acknowledgements

Working with such a recent technological development as EJB 3.0, the most up-to-date media – the Internet – has been a vital resource. In particular, Java forums and Sun tutorials have often provided solutions to problems that were too specific to appear anywhere else. I would therefore like to thank everyone who takes the effort to post their issues, and those who answer other people’s woes.

I would like to thank my supervisor, Dr. Kung-Kiu Lau, for his help throughout the project.

Finally, I thank the University of Manchester Computer Science Department and all its’ students for the opportunity and the support.
## Contents

Abstract ................................................................................................................................................. 2  
Acknowledgements ................................................................................................................................. 3  
Contents .................................................................................................................................................. 4  
Chapter 1: Introduction ....................................................................................................................... 7  
  1.1 Motivations for the Project .................................................................................................................. 7  
  1.2 Similar Work ...................................................................................................................................... 8  
  1.3 Project Approach ................................................................................................................................. 8  
  1.4 Report overview ................................................................................................................................. 9  
Chapter 2: Background and Literature Survey .................................................................................... 10  
  2.1 Overview of Enterprise JavaBeans 3.0 ............................................................................................. 11  
  2.1.1 Application Servers .................................................................................................................... 12  
  2.1.2 Containers ................................................................................................................................. 12  
  2.1.3 Middleware Services ................................................................................................................. 13  
  2.1.4 Roles in the EJB Application Life-cycle .................................................................................... 13  
  2.2 EJB Tools .......................................................................................................................................... 14  
  2.2 The Three Beans Overview ........................................................................................................... 14  
  2.3.1 Bean Protocol (RMI-IIOP) and Location Transparency ........................................................... 16  
  2.3.2 Java Naming and Directory Interfaces (JNDI) ...................................................................... 16  
  2.3.3 Container Environment for Beans .......................................................................................... 17  
  2.3.4 Bean Deployment ....................................................................................................................... 17  
  2.4 Entity Beans and the Java Persistence API Entities ...................................................................... 18  
    2.4.1 Object-Relational Mapping ........................................................................................................ 19  
    2.4.2 Persistence Context and the EntityManager API ................................................................... 19  
    2.4.3 Life Cycle of an Entity ............................................................................................................. 20  
    2.4.4 Entity Lookup and the Query API ............................................................................................ 20  
  2.5 Session Beans .................................................................................................................................. 21  
    2.5.1 Stateless Session Beans ........................................................................................................... 21  
    2.5.2 Stateful Session Beans ............................................................................................................. 22  
  2.6 Message-Driven Beans (MDBs) ...................................................................................................... 24  
    2.6.1 Java Message Service (JMS) ..................................................................................................... 25  
  2.7 Java Persistence – Advanced use of Entities ................................................................................ 26  
  2.8 Annotations and Dependency Injection ....................................................................................... 27  
  2.9 Transactions ..................................................................................................................................... 28  
  2.10 Security .......................................................................................................................................... 28  
    2.10.1 Web Application Security ........................................................................................................ 29  
    2.10.2 Application Security – Java Authentication and Authorisation Service (JAAS) ................... 29  
  2.11 EJB Timers ..................................................................................................................................... 29  
  2.12 Web-client Technologies ............................................................................................................... 30  
    2.12.1 Java Servlets ............................................................................................................................ 30  
    2.12.2 Java Server Pages (JSP) .......................................................................................................... 30  
Chapter 3: System Specification and Requirements Analysis .......................................................... 31  
  3.1 Project Boundaries ........................................................................................................................... 31  
  3.2 Detailed System Description .......................................................................................................... 31  
  3.3 Objectives of the System ................................................................................................................ 31  
  3.4 Inputs/Processes/Outputs ............................................................................................................... 32  
    3.4.1 Inputs (Forms) ........................................................................................................................ 32  
    3.4.2 Processes ............................................................................................................................... 32
Chapter 1: Introduction

First and foremost the project’s aim is to design, implement, deploy, and test software components, specifically with EJB 3.0, in the context of a flight booking system. As such, an early and sound understanding of the concepts and technologies surrounding these fields of computing is highly important. Researching into these areas defines the first milestone (see Gantt chart in Appendix A).

The final solution will include – at the very least – a customer-oriented web-based flight booking application, and if time is available, a stand-alone administration client application intended for use by the (fictional) flight company’s staff. The focus will be centred on the server side components and EJB’s technological features, and only basic domain functionality will be provided.

The completion of the design phase will be aimed at week ten, but throughout implementation (completion date targeted at week twenty-seven) the design will undoubtedly change. Testing should also be completed by week twenty-seven.

1.1 Motivations for the Project

The successful reuse of software components offers huge advantages when building new systems, or updating old ones. Imagine, rather than coding everything from scratch, it were possible to pick and choose pre-built function-oriented components, and combine them together quickly and with great flexibility to form an application. Projects, no matter how small or how large, could be less costly and produce more reliable solutions.

A flight booking system with a customer client and an administration client provides a perfect opportunity for code reuse via software components. The main advantages would be:

- At the lowest level, the two clients will be able to share components, as functionally, they have many similarities.
- Airlines and travel agents could in theory all access and share the same components from common repositories, and thus have the option to design their own systems based on reliable pre-built components (in practise most of these companies purchase Global Distributed Systems (GDS) from a small number of providers).
- The development of generic components for use in booking systems (which generally have the same fundamental architecture). It could be possible to provide standard components for these types of systems.

In terms of personal development, the project offers a huge amount of experience in building a distributed component-based system. With widely used cutting-edge technologies involved, the project provides an excellent chance to learn and practice the latest skills in a safe environment with free reign over design decisions to maximise EJB feature use.
1.2 Similar Work

Component-based software development has been around for decades, and is currently at its most prolific with large and influential vendors providing development tools and environments both in the open source and commercial worlds. Sun’s Enterprise JavaBeans is no exception and with a ten-year history it is widely adopted by individuals and companies of any size in almost any sector. Even so, most projects are in-house with scarce examples of component reuse between large groups and a distinct lack of central repositories for sharing code.

Traditionally, software components were typically found in the Graphical User Interface (GUI) and mathematics industries, but recent bursts of activity include the Geographic Information Systems (GIS) field (Google Maps a prime example) and the booking or ordering business. The continuing expansion and power of the Internet has provided much support for components, and its adoption will surely grow even more pronounced.

As for flight booking systems, they are clearly in abundance, with many airlines and travel agents each offering comparable, but unique solutions. However, the underlying systems are produced by only a handful of firms offering various distributed solutions. In a way, this is a representation of the component-based development style in the sense that any company that purchases the distributed systems are in fact using the same components as many other companies.

1.3 Project Approach

Due to the nature of the project – tackling a problem wherein the technology and system intricacies are unknown at the outset – it will be necessary to gain a firm understanding of all the involved concepts, design techniques, and software tools right from the start. In the first few weeks, time will be devoted entirely to research, reading and experimentation, all of which are crucial to design decisions. From this point, a specification can be produced for the client systems and the server-side EJBs. These will have to map to one another, and if necessary the client specification will be altered in order to implement all the EJB features.

The design will be a phased process, beginning with database schema (the booking system will be very data centric), followed by a close development of the clients and EJB code. Implementation will follow in much the same format, while testing will be carried out throughout implementation.

Overall, it will be sensible to follow a flexible and iterative design and development methodology. ‘Agile’ development seems to fit this approach to some extent. Throughout, as more experience is acquired, most likely it will be beneficial to carry out refactoring and changes to the design.
1.4 Report overview

The remainder of the report comprises six chapters, the first of which details essential information related to the field of software components, and also EJB specifics that are directly associated with the implementation. Following this is a chapter on the final design, including requirements specification, database schema, control flow, system architecture, and EJB component choices. Next, the implementation of the design is described with code examples and diagrams throughout. A results section follows, which illustrates with screenshots the final solution via use-cases. Chapter six is devoted to system testing, the results obtained, and an evaluation of these findings. The final chapter concludes the report with a summary of achievement and relates back to the original requirements. Further enhancements are also mentioned.
Chapter 2: Background and Literature Survey

The concept of building systems using standard reusable components has been around for almost forty years, first being published in 1968 in a paper entitled “Mass Produced Software Components” by M. D. McIlroy [14]. It became obvious that following the hardware industry, in which systems can be built up by combining component parts, would be beneficial. The difficulty occurred when attempting to define a standard software component, and even today there is no definitive specification. We therefore have to make a compromise of various definitions from a variety of sources. In the most general of these, a software component is thought of as software that exhibits the properties of an abstract idea. For example, the real world process of buying products from an online website could involve the process of calculating the total price, and later authorising the customers credit card and handling the payment transfer. Each of these business concepts could be implemented in a system using individual components. More specifically, a component must be generic enough to offer reusability. Components are thus generally more difficult to implement than a solution that is a perfect fit to the problem (similarly, a component may not be a perfect solution to any particular problem).

It has been stipulated, “Although the industry is moving toward component-based construction, most software continues to be custom built” [1, p.39]. This trend in today’s software industry is centred on the discipline of Component-Based Software Development/Engineering (CBD or CBSE), which makes use of pre-built software components wherever possible. The attraction of this approach results largely from the relative expense of custom-made software.

A fundamental principle behind CBD is that of composition. This is simply the task of arranging components to build-up a system, and the concept of how to work out attributes of a composite based on its individual components. This idea is hugely powerful as customisation is almost limitless, allowing developers the freedom of design promoted by code reuse.

Component models set standards that components must adhere to. These can include interface, usage, and deployment standards. One such model, called the 3C model [1, p.859], describes a software component by its concept, content, and context:

- Concept describes the purpose of the component.
- Content describes the inner workings of the component (of interest to developers).
- Context describes the features of the component in terms of the domain in which it will be used.

From defining a model, frameworks can be build that aid development of components and provide deployment support. Frameworks such as CORBA, COM and EJB are all used for enterprise application development of distributed, multi-tier systems (such as the three-tier model).

Modern frameworks provide various middleware services to help developers by allowing them to focus on business decisions (see section 2.1.3).
In the days when monolithic applications were commonplace and the object-oriented approach was in its infancy, it was realised that the inflexible and non-portable nature of applications was restricting code reuse. This had to change. The introduction and later maturity of object-oriented design was a step forward, however it was found that in many situations using CBD provided a superior solution. In this environment an engineer could focus on relevant business problems and good coding practices, rather than spending time reinventing middleware services.

Essentially, CBD advertises reuse and reduces cost while frameworks such as EJB do much to increase application reliability and performance.

2.1 Overview of Enterprise JavaBeans 3.0

Enterprise JavaBeans 3.0 (EJB 3.0) is the latest offering from Sun Microsystems released as part of the new Java Enterprise Edition 5.0 (Java EE 5.0). Described as “the cornerstone of Java EE” [15, p.32], EJB is a component framework ideal for building server-side, distributed components written in the Java programming language. It is a heavyweight architecture that promises rapid development of scalable, reliable, interoperable, and secure applications.

The framework itself consists of two parts:

- A specification that defines the communication between components and application servers in order to provide a programming model based on ‘write once, run anywhere’ behaviour.
- A collection of Java interfaces (part of the EJB API).

Being freely available, EJB is a popular choice for both large and small businesses, and had become the industry standard for distributed Java applications. It features an array of services including but not limited to those from CBD, distributed computing, database systems, transactional processing, security, and message-oriented computing.

The ‘beans’ in the name refers to the concrete building blocks of the framework – the components. These will be discussed in greater detail in later sections. Figure 2.1 shows an overview of the various Java EE 5.0 technologies and where EJB fits into this.
2.1.1 Application Servers

Application servers are heavyweight software offerings to the field of CBD. They consist of a complete package offering support for a specific framework (such as EJB), a container bundled with middleware services, deployment tools, web development APIs (such as JavaServer Faces), and all kinds of management support.

Recent popular servers include the Sun Java Application Server, IBM WebSphere, Oracle Application Server, and JBoss. Java Application Server 9.0 was chosen for use in this project. It is a free open source product from the company behind EJB. It offers excellent support for EJB 3.0, and can be integrated into many IDEs.

2.1.2 Containers

One of the key and fundamental parts of any EJB application server is the container. It is the essential piece of software that houses beans (software components) by providing a runtime environment for components to ‘live’. The container manages a
bean’s life cycle and access to the beans’ methods (from client calls). The bean has implicit access to the containers’ middleware services (see the next section).

Containers are extremely complex pieces of software, as they must be able to provide 24x7 reliability to a huge number of users, and cope with failures and exceptional behaviour, all within a highly secure setting.

2.1.3 Middleware Services

To spare application developers the headaches of writing their own services, EJB containers provide the following:

- Remote Method Invocation (RMI): a way of connecting tiers in a multi-tier model to allow communication such as method calls and parameter passing, and sending SQL statements (using the Java Database Connectivity (JDBC) API).
- Transaction management: there is no need to worry about writing your own transaction handling.
- Life-cycle management: includes creating, destroying, passivating, and activating beans.
- Resource management: thread, socket, and database connection management, and support for concurrent requests.
- Remote accessibility: allows external clients access to local beans.
- Clustering and load balancing: support for scalable and fail-safe deployments.
- Transparent failover: protects against network and server crashes by rerouting to another server.
- Messaging: supports loose coupling of code and components.
- Security: authentication and authorisation.
- Logging and auditing support.

Services will vary between different application servers, but the above are common features, while many more could be present.

As well as providing these services transparently, EJB offers the ability to explicitly call APIs of various middleware services. For example, gaining control of the transaction service allows the developer more flexibility when required. On the whole, an implicit approach is favourable as it

- Improves productivity and is easier to write as there is no need to code calls to middleware services.
- Enhances maintainability due to the clean nature of the code because the business logic is separate from the middleware code. Furthermore, endpoints (message consumption code) can be interchanged without causing issues.

2.1.4 Roles in the EJB Application Life-cycle

When developing an application using the EJB framework, the life cycle can be split into three distinct phases – development, deployment, and administration – each managed by one or more roles:
• The bean provider writes and tests business components.
• The application assembler uses the beans made available by the bean provider to build-up the application, often having to write other components or code to ‘glue’ together the individual parts of the system. This role also involves creating a user interface to act as the client to the server-side.
• The EJB deployer handles the production environment, fixing deployment issues unknown and unfound in the test environment.
• The system administrator supports the application once it goes into production.

Clearly, in a small application, these roles are often merged and carried out by one person.

2.2 EJB Tools

Due to the standards brought about by the strict EJB specification, coding component beans is greatly supported by Integrated Development Environments (IDEs). These tools allow quick and effective creation and refactoring of beans. The IDE of choice in this project was the open source tool NetBeans 5.5. It is highly integrated into the Java Application Server and provides support for code refactoring and EJB specific functions.

2.2 The Three Beans Overview

Enterprise JavaBeans are server-side software components capable of representing a variety of real-world objects, processes, and communications. There are often more complex than client-side components such as GUI widgets, and can perform very powerful operations due to support from underlying middleware services.

Beans are defined in a single Java class and can be implemented to carry out a diverse range of business problems. For example:

• A pricing engine with a tax calculator for a website’s shopping cart.
• E-mail confirmation of orders.
• Transaction bank account transfers.
• Complex database queries.

EJB offers three different types of bean that can interact with one another to form back-end application logic. Session beans, message-driven beans, and entity beans are described in the following sections. However, it is important to mention that although session beans and message-driven beans have been enhanced in EJB 3.0, and are now classed as Plain Old Java Objects (POJOs), entity beans have been unaltered. Instead, ‘entities’ have been introduced to largely replace entity beans (see section 2.4).

Figure 2.2 illustrates a simple view of communication between beans, and how they communicate with different clients.
Figure 2.2 EJB and client communications.

Figure 2.3 shows a more detailed picture of interactions within the EJB container. The diagram steps through the process of invoking a bean method from both local and remote viewpoints. The first step is for the clients to call an EJB method (via RMI-IIP communication – see next section), by using JNDI lookups or annotations (sections 2.3.2 and 2.8) to resolve the business interface references. Each client has their own view of the business interface that provides access to the beans’ methods. From this point, the container manages calls and services. The container has already generated wrapper classes based on the actual EJB classes. These wrappers provide methods that are invoked by the business interface rather than going straight to the bean class (step two). The wrapper classes provide a way of using the implicit middleware services offered by the container. These will have to be invoked both before and after (steps 4 and 5) the actual bean class method is called (step 3). The final step is the process of returning from the original method call, back up to the client.
2.3.1 Bean Protocol (RMI-IIOP) and Location Transparency

The Java language provides a native communication protocol for use between distributed objects located on different Java Virtual Machines (JVMs). This programming model is known as Remote Method Invocation (RMI). RMI-IIOP (Internet Inter-ORB (Object Request Broker) Protocol) is an extension of RMI and the official API of Java EE (and thus EJB). All EJB containers support this protocol, therefore allowing interoperability between different container vendors.

Unlike RMI-IIOP, location transparency is relevant to remote invocation. It provides remote clients, i.e. clients in a different JVM from the invoked object, the capability of communicating with a remote interface irrespective of the need to know the location of the machine on which the interface object resides. The Java Naming and Directory Interfaces (JNDI) facilitate remote lookup (see following section).

In EJB 3.0, Plain Old Java Interfaces (POJIs) are defined to provide access for local and remote clients.

2.3.2 Java Naming and Directory Interfaces (JNDI)

The JNDI consists of two powerful APIs that once learned can be reused with nearly every naming and directory service. Naming services are essentially bindings between names and objects, while directory services – such as LDAP and Microsoft
Active Directory – hold objects defined within an information model that can be queried and organised.

With EJB 3.0, JNDI is abstracted for the developer to supply easy access to EJB components and other resources via one API.

### 2.3.3 Container Environment for Beans

EJB containers provide the environment in which beans live and communicate externally. This communication can be with resources (such as databases), other beans, and messaging topics and queues, among other things. When referencing these objects from within a bean, an abstraction is created by the environment that is only resolved upon deployment. This is very useful and again promotes reuse, as a bean will remain separate from referenced objects even if they change.

### 2.3.4 Bean Deployment

Packaging and deploying beans can be a manual or automated process depending on what tools are used. Typically IDEs and application servers offer EJB-JAR file packaging features for deployment. As illustrated in figure 2.4, various classes, interfaces, and deployment descriptors may go into an EJB-JAR file. The process of generating this file will also automatically work out dependencies and include all other necessary files.

Once packaged an EJB-JAR file can be deployed to the container; a process that reads any included descriptors and resolves references.

The NetBeans IDE can interact with the Java Application Server to provide highly automated code compilations and deployments by generating the necessary build scripts and deployment descriptors.
2.4 Entity Beans and the Java Persistence API Entities

One of the main objectives for EJB 3.0 was to replace the entity beans of EJB 2.1 with a lightweight POJO-oriented solution to persistence. This has been achieved with the newly formed entities, which provide a quick and easy approach to persistence. The project uses these entities and as such the remainder of this report focuses solely on these permanent persistent components.

Entities are essentially models of nouns, such as customers, products, orders, credit cards, etc., that can map to relational database tables. They are synchronised to a
database by a persistence provider, and can therefore live for a long time (for as long as the database data is around).

The concept is massively powerful as it allows access and manipulation of data in the database via the entity. From the perspective of the developer, entities are all they need to worry about.

Source 2.1 shows a very simple example of an entity that models a person. The @Entity annotation tells the container that this class is an Entity and should be treated as such. The actual Java class does not have to extend any specific interfaces or EJB classes, however implementing java.io.Serializable means entity instances can be passed a parameter between beans and other Java classes. The class itself consists of four public variables (analogous to the fields that make up a row in a database), the first of first is annotated with @Id, indicating it is the primary key.

```java
@Entity
public class Person implements Serializable {
    // The id variable is the primary key and entity identifier
    @Id
    public long id
    public String name;
    public String address;
    public Date dateOfBirth;
}
```

Source 2.1 Person entity.

### 2.4.1 Object-Relational Mapping

Object-relational mapping is the concept behind how entities are stored in a relational database. It is achieved by decomposing in-memory entities into individual attributes that will become a row of fields in the database. This provides a powerful query mechanism, and allows the database data to be examined at the level of the mapped entity, thus enabling auditing and debugging.

Nowadays, the most common approach to object-relational mapping is to automate the process by using tools often offered by the database vendor. For this project, a MySQL database is used, which provides a driver to carry out the mappings.

### 2.4.2 Persistence Context and the EntityManager API

Entities can be accessed from local session or message-driven beans but not remotely. Access is provided by the persistence context, which stores entity instances and provides the connection with the database. It is controlled by the EntityManager interface. This API is used to add new entities to the persistence context by invoking the persist() method. At this point, the entity is said to be in the ‘managed’ state and is synchronised with the database. When the persistence context ends, the entity is said to be in the ‘detached’ state.
Source 2.2 demonstrates the persist() method for creating a Person entity. The EntityManager is injected using the @PersistenceContext annotation, and is used in this case to persist the person entity.

```java
@PersistenceContext
private EntityManager entityManager;
...
public Person createPerson() {
    Person person = new Person();
    entityManager.persist(person);
    return person;
}
```

**Source 2.2** EntityManager persist() method.

### 2.4.3 Life Cycle of an Entity

Managed by the EntityManager, the life cycle of an entity can be described in four distinct states, the transitions of which relate to methods in the EntityManager API. Managed and detached have been described in the previous section. The other two states are

- ‘New’: the entity is instantiated but is not held in the persistence context and thus not synchronised with the database.
- ‘Removed’: the entity has been targeted for removal from the database.

Figure 2.5 illustrates the states and transitions associated with an entity life cycle.

![Figure 2.5 Life Cycle of an Entity](image)

### 2.4.4 Entity Lookup and the Query API

To pick out a unique entity instance from the database, the EntityManager provides a find() method that returns a managed entity if the given primary key – passed as a parameter – exists (see source 2.3). While this method is very useful, the developer can write complex queries to produce more varied and interesting results.

```java
public Person find(Object pk) {
    return (Person) entityManager.find(Person.class, pk);
}
```
Source 2.3 EntityManager find() method.

The EntityManager is used to create an instance of a Java Persistence Query object, allowing EJB-QL (query language) to be executed. EJB-QL is based on an entity data model thus allowing an abstraction of the underlying database, which assures portability between databases. With EJB-QL, static and dynamic queries are supported.

2.5 Session Beans

Session beans are components that model verbs. For example the process of booking an order, calculating the cost of an order, or authorising a credit card payment, can all be developed using session beans. Unlike entities, they are non-persistent and usually do not live through server or machine crashes. They are created when needed and destroyed when the client session finishes or once the bean is no longer in use.

Session beans enter into ‘conversations’ with clients – basically a communication of method invocations. It is the length of these conversations that define the two different types of session beans, namely stateless and stateful.

2.5.1 Stateless Session Beans

For a single method invocation conversation, a stateless session bean is suitable. This type of component will not store any state from method call to method call. The container manages the bean’s life: destroying, recreating, or simply keeping it alive for other clients to use. In this way, the container can provide a small number of instances to service a potentially large number of clients. The technique of instance pooling – as show in figure 2.5 – makes for very efficient processing of bean code.

![Instance pooling of stateless session beans.](image)

A stateless session beans life cycle (see figure 2.7) has just two states, with transitions handled entirely by the container. The process begins with instantiation of a bean, injecting any dependencies if required. From here, the PostConstruct callback method
is invoked if present, which can be used by the developer to carry out any bean initialisation. Once in the pool, a bean instance is treated in exactly the same way as all other bean instances and is invoked by the container on a per-method basis. This essentially means that clients do not need to care if the container uses a different bean instance on every method call. When clients are not invoking many bean instance methods, the container may decide the pool has too many instances. In this case instances can be destroyed to increase resource.

![Diagram of bean lifecycle](image)

**Figure 2.7** Life cycle of stateless session beans.

Due to the nature of stateless beans, they are ideal for high-performance processing such as implementing complex algorithms, or validation and verification processes.

### 2.5.2 Stateful Session Beans

Stateful session beans differ from their stateless counterparts as suggested by their name, these components store state between method innovations. The container services clients with stateful beans in a different way from stateless ones: an instance of a bean will relate uniquely to a client. As such, pooling mechanisms involve greater complexity. The processes that the container implements to cope with this can be described using states and transitions, and thinking about the life cycle of the bean.

From the ‘ready state’ a bean can be passivated – transferred from physical memory to secondary storage (for instance hard disk) – in order to preserve resources. From a passivated state, a bean can be swapped back into physical memory, a process called activation. Figure 2.8 illustrates the passivation process. This occurs when the number of beans in the container’s pool is at its limit, and so an existing instance must be serialised to secondary storage. Activation is the process in reverse: a passivated bean’s method is called and so the bean must be recovered from storage and activated.
The life cycle of stateful session beans (see figure 2.9) starts in the same way as stateless session beans. The complexity arises upon bean instantiation: there is a one-to-one mapping between instances and clients. This means the container must always call the business methods of the bean instance — activating the instance if necessary — associated with a particular client.

Source 2.3 illustrates a simple stateful session bean used to keep track of an integer counter variable. The @Stateful annotation is used, and the class implements the methods of the local business interface called CounterLocal.

```java
@Stateful
public class CounterBean implements CounterLocal {
    // the counter variable stored
    private int counter = 0;

    // increments the counter and returns the new value
    public int increment() {
        return counter++; // returns the new value
    }
}
```
public int increment() {
    return ++counter;
}

Source 2.4 Stateful session bean.

2.6 Message-Driven Beans (MDBs)

Messaging in EJB provides a lightweight asynchronous alternative to RMI-IIOP. There are certain situations where opting for the messaging approach will benefit:

- Asynchrony is required.
- A decoupling of the client from the server is needed (a loosely coupled system promotes reusability).
- Reliability is a must, even when server or network crashes occur.
- Many senders and receivers are required in order for multiple clients to communicate with multiple servers simultaneously.

To provide an example of when messaging can be used to a client's advantage, think of an ordering system and the processes involved in authorising a credit card and transferring money to pay for a product order. Approaching this business logic using asynchronous messaging will produce a user-friendly solution, as the customer will not have to wait around for these processes to complete.

For all its advantages over RMI, choosing messaging is often a trade-off due to performance issues associated with the overhead of having message-oriented middleware (MOM), rather than a simple direct connection.

MDBs are stateless components that receive (consume) messages such as Java Messaging Service (JMS) messages. The key to understanding these beans is to realise that they are entirely decoupled from the client, and as such, the container invokes the bean when a message arrives at its destination (endpoint). Figure 2.10 demonstrates a client sending JMS messages to MDBs.
Figure 2.10 Client sending JMS messages to MDBs.

Source 2.5 shows an example of a MDB that consumes JMS messages off a topic.

```java
@MessageDriven(mappedName = "jms/NewMessage")
public class NewMessage implements MessageListener {
    // receives the message
    public void onMessage(Message msg) {
        if (msg instanceof TextMessage) {
            TextMessage tm = (TextMessage) msg;
            try {
                String text = tm.getText();
                // ...
            } catch (JMSException e) {
                e.printStackTrace();
                // ...
            }
        }
    }
}

Source 2.5 MDB for JMS message consumption.

2.6.1 Java Message Service (JMS)

MOM products are largely proprietary pieces of software and cause numerous problems for developers such as learning many APIs and issues when porting to another system. JMS is a means of solving these problems through standardisation. It achieves this with an API and a Service Provider Interface (SPI). The former is used to code the sending and receiving of messages, while the latter is where a JMS provider that communicates with a MOM implementation is ‘plugged in’.

With JMS, two messaging domains are available: publish/subscribe (pub/sub) and point-to-point (PTP). The EJB terminology used to represent each of these is topic and queue respectively.

Using the JMS API is non-trivial and requires a deeper understanding than that of RMI-IIOP. Figure 2.11 is an overview of a client using JMS to communicate with various queues and topics.

The first step is for the client to retrieve an instance of the JMS provider’s ConnectionFactory (created by an EJB administrator). With this, a connection can be created, which in turn is used to create a JMS session – a message consumer and producer factory. Step four is to find the message destination using JNDI lookup. From here, session and destination objects are used to create message producers and consumers respectively. These provide the facility of sending and receiving messages.
2.7 Java Persistence – Advanced use of Entities

With the new entity persistence mechanism at the forefront of the Java Persistence API, features such as inheritance, polymorphism and relationships can all be achieved. This project only involved the use of relationships.

EJB 3.0 supports every type of entity relationship, and both kinds of directionality – unidirectional and bidirectional – can be specified. Relationships in EJB also have the
concept of cascading persistence functions. Persist(), merge(), remove() and refresh() methods can all be cascaded to related entities.

Another significant setting when defining entity relationships is that of lazy and eager loading. The former loads an entity as and when required, while the latter loads the entity and all associated entities straight away. Clearly, lazy loading puts less strain on the system initially, but eager loading can be useful if the related entities will be accessed as well.

Source 2.6 demonstrates a simple relationship between Person and House entities. Taking the contrived assumption that one owner (person) can own many houses; a one-to-many unidirectional relationship is modelled. The @OneToMany annotation is used to specify the cascade and fetch type.

```java
@Entity
public class Person implements Serializable {
    // The id variable is the primary key and entity identifier
    @Id
    private long id
    private String name;
    private String address;
    private Date dateOfBirth;
    private Collection<House> houses;

    @OneToMany(cascade={CascadeType.PERSIST}, fetch=FetchType.EAGER)
    public Collection<House> getHouses() {
        return houses;
    }
}

@Entity
public class House implements Serializable {
    // The id variable is the primary key and entity identifier
    @Id
    public long id
    public int number;
    public String street;
    public String postcode;
}
```

Source 2.6 One-to-many unidirectional relationship.

### 2.8 Annotations and Dependency Injection

The use of annotations is new in EJB 3.0, and it has been introduced as part of Java SE 5.0. In brief, they can be thought of as comment annotations (such as @Param and @Return) that are added to certain parts of Java code, e.g. classes, methods and variables, to add compiler information relevant to deployment.

Another useful feature of annotations is their interaction with the container to provide resource references and resource injection. These essentially allow developers to define a reference using standard annotations, and leave it to the container to lookup up and ‘inject’ the appropriate object into place.
2.9 Transactions

EJB offers a sophisticated transaction mechanism. Developers can take full advantage of this to provide atomicity, consistency, isolation, and durability (ACID) to their bean code. The notion of “demarcating transactional boundaries” [15, p.288], i.e. who begins and ends (by committing or aborting) the transaction and when each of these happens, means transactions can be managed by the container, bean, or client.

Container-managed transactions (see figure 2.12) are the simplest of the three options. The container handles any transactions, implicitly calling begin and commit or abort statements. The advantage of this approach is that time can be saved from not having to write any transaction code.

Bean-managed transactions offer more flexibility as – unlike container or client managed transactions – it is possible to code several transactions into one bean method.

Figure 2.12 Container-managed transactions.

2.10 Security

Security can be one of the most important factors in mission critical systems, as a breach could have devastating effects – often financially related – to the entire business. Hence, security in EJB should not be overlooked, and fortunately, they are controls available to minimise the risks.

Authentication and authorisation are two of these controls. The former may involve a verification process, perhaps by comparing the given username and password with a list in a database table. Authorisation occurs once a user has been successfully
authenticated (known as the principal), and controls various access rights by validating against the principals’ privileges to resources such as bean methods.

2.10.1 Web Application Security

Similar to EJB security, but not included in the EJB specification, Web application security is dealt with in the Java Servlet and Java EE 5 Platform specifications. Both authentication and authorisation are supported. Data integrity and confidentiality protection can only be achieved using HTTPS Client authentication.

2.10.2 Application Security – Java Authentication and Authorisation Service (JAAS)

EJB security is limited to authentication and authorisation. While authentication occurs before method invocation, authorisation is performed at the start of an EJB method. The Java Authentication and Authorisation Service (JAAS) is part of Java EE 5, and can be leveraged by EJB applications to provide an almost invisible and highly flexible security implementation. Authentication modules can be written and plugged in to connect with JAAS.

EJB authorisation can be declarative (container handled checking) or programmatic (bean handled checking). Whether using one option or the other, security roles must be defined. Client identities are associated with the various security roles, and these mapping are checked when a client attempts method invocation. The advantage of this approach is the distinct separate of security roles from business logic and thus increases portability.

2.11 EJB Timers

The EJB scheduling mechanism, around since EJB 2.1, can be used with enterprise beans. The container-managed Timer Service provides APIs for creating timers set to expire at a particular date or within a specified time, with the possibility of further repeated expirations. Upon expiration, the container will invoke the timeout callback method defined by the bean developer.

All three types of bean can be implemented with a timer service, but stateful session beans and Java Persistence entities are not currently supported. As an example of an EJB timer, imagine a commercial bank that wishes to notify its’ customers when their credit cards are close to expiry. This could be achieved by creating a timer when a customer first gets issued a credit card, and setting the timer to expire say two weeks before the card itself expires. Thus, when the timer expires and the callback method is invoked on the bean, a process could kick-off that sends an email to the customer warning of their approaching card expiration.
2.12 Web-client Technologies

While EJB 3.0 will be used for server-side operations, Java servlets can be combined with Java Server Pages (JSP) to offer web-client functionality.

2.12.1 Java Servlets

Servlets can be thought of as components themselves, and like EJBs, they live a container. However, this Web container does not offer the diverse and powerful range of middleware services associated with an EJB container. As such, servlets are often used for straightforward HTTP request/response handling. Considered Java objects rather than components, they tend to follow a simple paradigm: receive request as input, process request data and perform some logic, then send a response back to the caller or on to another servlet (or JSP).

The flexibility and power of servlets comes into its own when combining EJBs, servlets, and JSPs. This is possible because all three have the Java language in common.

2.12.2 Java Server Pages (JSP)

Java Server Pages (JSP) is a script-based, presentation-oriented technology. They go hand-in-hand with servlets as they themselves are translated into servlets come runtime. However their strength and general use is in HTML rendering. By combining with servlets, the qualities of both can be taken to full advantage: a separation of control logic and data processing, from presentation issues allows maximum code reuse and provides greater readability.
Chapter 3: System Specification and Requirements Analysis

3.1 Project Boundaries

The project is limited to a nominal 240 man-hours, which includes a full development lifecycle of researching, designing, implementing, and testing. Time must therefore be spent intelligently on the essential and most important parts of the system. Considering the implementation phase; to begin with, server-side EJB development will be the focus. Website implementation will follow, then there will come a point at which it will be clear whether a Java application client can be produced or not.

Concerning the design and implementation of the potential clients, work will be concentrated on the required functional aspects rather than the presentation and sophisticated features. Style sheets, headers and footers are likely to be used as they potentially speed up development overall, however new technologies such as AJAX and other Web 2.0 concepts will not be considered.

System security will be present to a degree. The website will offer login and logout functionality, and will possibly use SSL, but this will be a nice to have as the site will be hosted locally with no intended external use. The Java application client will make use of the EJB security features applicable to it and not available to web-based clients.

The website will be limited to one user type, whereas the Java application client will allow several categories of administrators.

3.2 Detailed System Description

The system will be three-tiered: a MySQL database, an EJB 3.0 back-end providing the business logic, a servlet and JSP front-end for the customer-oriented online flight booking client, and another front-end for the administration staff. The database will store flight, booking, customer/user, and other related data. The EJB components will be developed specifically to meet business problems associated with the two clients. The website will offer users the functions expected from a commercial flight reservation system: ability to sign-up, login, search flights, book flights, change and delete relevant and historical data. The administration client will offer greater privileges such as updating, deleting, and adding new flights, airports, and aeroplanes, but these will be restricted to the appropriate user level. Flight, booking, and customer user levels will be defined.

3.3 Objectives of the System

To create a working flight booking system based on EJB 3.0 component technology and design is the ultimate aim. The system must allow customers to register, login, logout, search flights, book flights, and view and update historical data such as previous bookings and personal details.
Further aims include the implementation of Java application client to be used for flight, booking, and customer administrative tasks. This client should allow basic create, read, update, and delete (CRUD) operations where appropriate, and based on their authorisation level. The overall system should be reliable and fault-tolerant.

3.4 Inputs/Processes/Outputs

3.4.1 Inputs (Forms)

Sign-up Form: this form requires potential customers to enter their personal details such as their email address and login password. A unique email address will be required.

Login Form: users will be required to login using their email address and password. This will assign them a shopping cart, allow them to book flights, and access other account features.

Update Customer Details: allows a customer to update their personal details (e.g. name, address and email). Login is required.

Flight search: various flight details are required. Customers do not have to login to perform searches.

Payment Card Registration: customers, once logged in, can add payment cards. Forms will be available for customers to update and delete their payment cards.

Make a booking: the booking process involves many related forms and inputs. Bookings can be updated and cancelled by the customer who made them.

Add, update, and cancel (delete) flights: forms for use by the flight administrator.

Add and edit airports: forms for use by the flight administrator.

Add and edit aeroplanes: forms for use by the flight administrator.

The administrator front-end processes for flights would be very similar to that of bookings and customer (all CRUD operations), and so will not be considered for implementation. However, most of these CRUD operations will be required for the web client.

3.4.2 Processes

Most database CRUD operations should be available on every table in the database. These operations will be implemented with EJB components, and as such will be shared and reused by both clients. The standard read operation should perform a search using the entity’s (table) primary key, and thus return one entity instance.
Aeroplane details: find aeroplanes given make and model.

Airport details: find airports given the short name. Find distinct countries where airports are located.

Booking details: calculate booking price, validate booking, pay for booking.

Company Account details: find company account given bank name, sort code, and account number. Withdraw funds from Company Account. Deposit funds to Company Account.

Customer details: find customer using login details, or email address, or surname. Authenticate customer’s login. Check new customer details are valid. Check existing customer details are valid (on update).


Transfer funds: perform a money transfer upon booking confirmation by withdrawing funds from the customer’s account, and depositing the funds in the company’s account.

Emailing functionality: building and sending emails – upon customer sign-up, forgotten password form, booking completion, update booking, cancel booking, and credit card expiry.

Flight Class Type details: find flight class type given a FlightClassType string value.

Flight details: find flight given FlightNo. Find flights given departing from, destination, departure date, and class type. Check flight capacity.

Payment Card details: find all payment cards associated with a customer.

Shopping Cart: add flight, remove flight, get all flights, add booking flight, remove booking flight, get all booking flights, purchase contents of cart given a payment card.

3.4.3 Outputs (Tables)

Customer Details: for customers to update their personal details.

Flight Details: results from performing a flight search. Also for flight administrators to update altered flights.

Payment Card Details: for customer to update or delete registered payment cards.

Booking details: customer can update and cancel their previous bookings.

Airport details: for flight administrators to make changes to airports.
Aeroplane details: for flight administrators to make changes to aeroplanes.

3.5 User Requirements

A unregistered online flight booking website user:
- View site information (homepage)
- Sign-up
- Search flights

A registered customer:
- View site information (homepage)
- Login using email address and password credentials
- Logout
- Update personal details
- Search flights
- Add, update and remove flights in shopping cart
- Book flight in cart
- Add, update and remove payment cards
- View previous booking
- Update and delete previously made bookings

Flight Administrator:
- Add, view, update and cancel flights
- Add, view, update and remove airports
- Add, view, update and remove aeroplanes

Booking and customer administrators will be defined, but will not carry out any operations, but may be used to test their security on flight administrator operations.

3.6 Processing Requirements

Data entry:
Enter details of a new customer
Enter details of a new flight
Enter details of a new booking
Enter details of a new payment card
Enter details of a new aeroplane
Enter details of a new airport

Data update and deletion:
Update/delete customer details
Update/delete flight details
Update/delete booking details
Update/delete payment card details
Update/delete airport details
Update/delete aeroplane details
3.7 Performance Requirements

- User friendly
- Overall speed of the system
- System can be run on a standard, low performance PC
- Straightforward tables, forms and reports

3.8 Hardware and Software Available

The system will be implemented on a Pentium 4 - 2.4GHz PC with 1.5GB of RAM and Internet connection. The computer runs under the Windows XP Home Edition operation system, with Mozilla Firefox installed as the web browser.

3.9 Capabilities and Limitations of Resources

The computer available is capable of implementing and running the proposed system. However, distributed systems are generally run across multiple servers for performance reasons. This will not be available.

The software available is highly capable of implementing and running the proposed solution.

3.10 Constraints

Constraints are generally bought about by the underlying database.

3.11.1 Must Haves:

- Every flight must have an associated airport and aeroplane
- Every booking must have at least one flight and exactly one customer
- Every payment card must be associated with a customer
- Every payment card must be associated with exactly one customer account
- Every customer must have exactly one account

3.11.2 Should Haves

- For every booking, before the flight date is reached (earliest if multiple flights), a payment card should be associated with the booking
- Customers should be notified by email upon completion of a booking
3.11.2 Could Haves

- Customers should be notified by email upon sign-up, if they change or cancel their booking, if their payment card has expired, or if they have forgotten their password (form for this).
- All CRUD operations available for the Java administration client.
Chapter 4: Design

This chapter will focus on the major elements of design in the order they were considered and implemented in the project. Firstly, decisions made on what technologies to use for development of the server-side and two clients will be described. Next, the choice of software will be reviewed, as this affected almost all design aspects. Based on a simple understanding of development using the chosen tools, a design methodology was identified. Following this, database schema design – tables, attributes, and relationships – was considered, taking into account EJB specific issues. With the design of the database back-end defined, work flow diagrams were drawn up to model the structure and layout of the website client.

One of the largest and most crucial design tasks was scoping out the business operations and associating them with EJBs. This allowed early insight into what was feasible, most important, and relevant to the overall requirements.

4.1 Technology Decisions

With EJB 3.0 the choice for server-side development, client-side development was restricted to Java-based technologies. Servlets and Java Server Pages (JSP) go hand in hand to provide a logic and presentation coupling suitable for the web client. When looking at the whole picture – EJBs, servlets, and JSPs – a Model View Controller (MVC) paradigm becomes clear (see figure 4.1 for illustration).

The Java application client makes use of Java Swing technology for GUI work, while plain Java classes can be used to call EJB components.
4.2 Software Options

The most vital piece of software in an EJB application is the application server. Java Application Server Platform Edition 9.0 was chosen for use in this project. It is a free open source product from the company behind EJB. It offers excellent support for EJB 3.0 and can be integrated into many IDEs.

The IDE of choice in this project was the open source tool NetBeans 5.5. It is highly integrated into the Java Application Server and provides support for code refactoring and EJB specific functions.

After using the default Java EE 5 database server, a switch was made to the MySQL server. The basis of this decision was to provide more functionality and features on the database side.

To provide emailing functionality, a mail server was required. After installing and experimenting with various Java Mail Servers, hMailServer was chosen for its ease of use.

Figure 4.1 EJBs, servlets, and JSPs – a Model View Controller (MVC) paradigm.
4.3 Design Methodology

There are inherent design paradigms in EJB due to the strict specification of how code can and should be written. The specification promotes the Model View Controller (MVC) pattern wherein entities provide the model, session beans the controller, and the view is defined by the client code.

4.4 Database Schema Design

It was essential to quickly produce a schema design to represent the flight booking system, as an implementation could quickly follow which would identify design issues immediately.

With only a basic knowledge of the flight data used in booking systems, analysis and experimentation was required. By referring to existing online flight booking systems, an idea of the data stored was gathered. A perfect representation was unlikely, but a sufficiently realistic and complex model would be enough to achieve the overall goals. Figure 4.2 models the entities in the final design.

Of particular note are the relationships between the flight and booking tables. This requires a many-to-many relationship, which is modelled using a join table that stores further attributes. The reason behind this is as follows: a single customer booking will contain many flights, each of which will have specific data for the customer. This data cannot be stored in the Bookings table as it is repeated for each flight and would therefore be unidentifiable.

The two Flight-to-Airport relationships represent ‘departing from’ and ‘destination’ attributes.

Aeroplane and flight capacities are related but different. Every aeroplane has set passenger capacities for each flight type (economy, business, and first class). Every flight stores current passenger capacities, which will be updated when a customer makes a booking for that flight, or updates or cancels their booking.

The CustomerAccount and CompanyAccount entities were introduced into the system to facilitate payment transfer. When a customer completes a booking their payment card will be charged, a process that takes money from their associated account and transfers it to the flight company’s account.

FlightClassType and PassengerType store details of relative costs to be multiplied by the base flight price. For example, if an adult passenger books a first class flight, the percent values in each of these tables will be taken into account when the booking total is calculated.

The AdminClientUser table is used for the authentication process of the Java application client. It is similar in concept to the Customer table.
Figure 4.2 Database schema design.
4.5 Server-Side EJB Component Design

EJB 3.0 components will be the heart of the system. They will have to be coded with reusability (two distinct clients will accessing them), reliability, and readability in mind.

Entities, session beans, and MDBs are included in the design. Entities are the foundation on top of which everything else will be built, and so should be designed first. Session beans provide the business logic and are coupled with local and remote business interfaces to expose methods for clients to access. Both stateless and stateful session beans will be used to implement the key features and functions of the system. Lastly, MDBs will be designed, as these will be used to improve the user’s experience when communicating with certain EJBs, but are ultimately non-essential.

Figure 4.3 provides a high-level overview of the system in terms of components, both server-side and client-side. The direction of the arrows represents flow of control or interactions. Central to most processes are the EJB create, read, update and delete (CRUD) database operations.

Figure 4.3 High-level overview of system component design.
4.5.1 Data Tier

This will be a straight representation of the database schema into entities, as all table data requires permanent persistence. Entities and data will be shared between both clients. The EJB architecture will be followed to promote reuse – entities will not implement any business logic.

In theory plain Java classes could offer similar functionality to entities, but these would lose out on the performance side as the EJB container caches entity data.

4.5.2 Business Logic Tier

Session beans and MDBs will form the business logic of the application. Session beans will provide access (both remote and local) to entities. MDBs will be used in situations where asynchronous communication is appropriate. They will receive client messages and carry out business logic without the client having to wait for the operation to complete.

This tier essentially provides the processes between inputs (e.g. form entry) and outputs (e.g. web page tables). These processes were defined and assigned to various EJ Bs:

- Find aeroplanes given make and model – Aeroplane entity façade (stateless session bean).
- Find airports given short name – Airport entity façade (stateless session bean).
- Find distinct countries where airports are located – Airport entity façade.
- Pay for booking – stateless session bean.
- Validate booking – stateless session bean.
- Calculate BookingFlight price – stateless session bean.
- Find passenger type – PassengerType entity façade.
- Find company account given bank name, sort code, and account number – CompanyAccount entity façade.
- Withdraw funds from CompanyAccount – CompanyAccount entity façade.
- Deposit funds to CompanyAccount – CompanyAccount entity façade.
- Find customer using login details – Customer entity façade.
- Find customer using email address – Customer entity façade.
- Find customer using surname – Customer entity façade.
- Authenticate customer’s login – Customer entity façade.
- Check new customer details are valid – stateless session bean.
- Check existing customer details are valid (on update) – stateless session bean.
- Find customer account given customer – CustomerAccount entity façade.
- Withdraw funds from CustomerAccount – CustomerAccount entity façade.
- Deposit funds from CustomerAccount – CustomerAccount entity façade.
• Setup customer account – stateless session bean.
• Transfer funds – stateless session bean.
• Emailing functionality: building and sending emails – stateless session beans.
• Find flight class type given FlightClassType – FlightClassType entity façade.
• Find flight given FlightNo – Flight entity façade.
• Find flights given departing from, destination, departure date, and class type – Flight entity façade.
• Check flight capacity – Flight entity façade.
• Find all payment cards associated with a customer – PaymentCard entity façade.
• Shopping Cart: add flight, remove flight, get all flights, add booking flight, remove booking flight, get all booking flights, purchase contents of cart given a payment card – stateful session bean.
• Warn customer of expired payment card – stateless session bean.

Session beans will provide the majority of system functionality. They will be sandwiched between clients and entities, and may communicate with other session beans to achieve their overall business function.

Initially, session beans will be designed to provide simple CRUD operations on entities. These can be achieved using session façades (see figure 4.4), an EJB design pattern that couples a stateless session bean with an associated entity. Performance increase is one of the main benefits of this approach; another is improved entity reuse as the business logic is entirely specific to the session bean and not the entity.

The use of business interfaces (local or remote) is essential if a client requires access to bean methods. In this application, the web client used such access. Remote interfaces on the other hand were of slightly less importance. However, the Java application client was located on a separate JVM and thus made use of remote interfaces to access bean methods.
Further uses of stateless session beans include business validation, performing calculations, and general functions that could be reused. Most of the business logic tier will consist of this type of component. For example, the booking process encompasses all of these areas.

A stateful session bean will be used to implement shopping cart functionality. This could be achieved using an entity, or a servlet’s or JSP’s access to an HTTP session. An entity-based solution is inappropriate for a number of reasons:

- A persistent shopping cart is not required.
- Processing an item and storing it in the cart is not incredibly complex or processor intensive (as such the customer will not lose out if they have to redo their shopping cart).
- The extra programming time needed to code a regularly scheduled shopping cart entity cleaning process (for unused carts left the database) outweighs any benefits.

A client side shopping cart implementation using servlets or JSPs is also not suitable because it breaks the MVC model. Choosing an entity or HTTP session solution both go against the original project objective of making use of as many EJB components and features as possible. A shopping cart is the only instance in which a stateful session bean can be used to great effect.

### 4.6 Customer Flight Booking Website Design

Designing a complex commercial website has the potential to be a project in itself. To keep things as simple as possible, focus will be on flow of control and reusing EJB components are much as possible. As such, presentation will be kept under control using cascading styles sheets (CSS) and headers to avoid repetition of code. The benefit will be a uniform look and feel for all pages.

To ease the development of the website, and to make use of EJB components, business logic will be avoided in any servlets as far as possible.

The website client will have local access to EJBs, and as such remote interfaces will not be needed.

Overall, communication between servlets and EJBs will be carried out using RMI-IIOP. This is because a response is often required immediately. Asynchronous messaging may be used for parts of the booking process. For example payment card validation and payment transfer has (in reality) the potential to take some time. Rather than the customer waiting around for a response, a message can be sent to a MDB which in turn calls further beans to handle these processes. Also, emailing customers can be achieved using messaging and MDBs.

Figure 4.5 shows a high-level system design of how the web client interacts with the EJB server-side to pass data around the application. The first step involves the user accessing the website from a web browser and performing some action, such as
submitting a form. The web browser prepares an HTTP request and sends it to the appropriate client servlet. The servlet processes the request and may invoke some methods on the server-side EJB components, passing in values when required. The control now switches to the server side wherein the container takes charge. Session beans, MDBs and entities can all communicate to process the business logic. Data may be retrieved or persisted to the database, and eventually a response will be returned to the calling servlet. From here, the servlet passes any relevant data to the corresponding JSP, which renders HTML code back to the web browser.

Figure 4.5 Architecture of the web client

4.6.1 User Inputs

User inputs were defined by thinking about what forms will be required:

- Login – users will be required to login using their email address and password. This will assign them a shopping cart, allow them to book flights, and access other account features.
- Sign-up - this form requires potential customers to enter their personal details such as their email address and login password. A unique email address will be required.
- Update Customer details – allows a customer to update their personal details (e.g. name, address and email). Login is required.
- Flight search – various flight details are required. Customers do not have to login to perform searches.
- Payment Card registration – customers, once logged in, can add payment cards. Forms will be available for customers to update and delete their payment cards.
- Make a booking – the booking process involves many related forms and inputs. Bookings can be updated and cancelled by the customer who made them.
4.6.2 User Outputs

The web client will display information for the user in tabular form:

- Customer details – for customers to update their personal details.
- Flight details – results from performing a flight search.
- Payment card details – for customer to update or delete registered payment cards.
- Booking details – customer can update and cancel their previous bookings.

4.6.3 Flow of Control

To help with implementation, it was vital to visualise and plan out the website structure and control flow. Figure 4.6 demonstrates a typical customer process of booking a flight. Various validation checks are included, such as login status and flight capacity.

![Flow of Control Diagram](image)

Figure 4.6 Control flow – making a booking.

4.7 Administration Client Application Design

The Java administration client will offer users a very different set of functions to website customers. Operations will be CRUD-based and will involve entities that customers have no control over. With these greater privileges, the focus will be on security. Various roles will be defined, each with their own access levels.
4.7.1 Security Roles

The client was designed for three distinct user types: customer, flight, and booking administrators. Each administrator will have their own privileges, providing restricted access to beans. Figure 4.7 shows a table of access rights each user will have on various entities. Note that having privileges on one entity can affect access rights on related entities. The flight administrator has responsibilities for airports and aeroplanes, as these are essential to flights and are directly related.

<table>
<thead>
<tr>
<th>Users</th>
<th>Customer</th>
<th>Flight</th>
<th>Booking</th>
<th>BookingFlight</th>
<th>Airport</th>
<th>Aeroplane</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer admin</td>
<td>CRUD</td>
<td>-</td>
<td>CRUD</td>
<td>CRUD</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Flight admin</td>
<td>-</td>
<td>CRUD</td>
<td>RUD</td>
<td>RUD</td>
<td>CRUD</td>
<td>CRUD</td>
</tr>
<tr>
<td>Booking admin</td>
<td>-</td>
<td>-</td>
<td>CRUD</td>
<td>CRUD</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 4.7 Security Roles – user access to entities.
Chapter 5: Implementation

As much a part of the implementation as coding EJBs, was the set up and configuration of the environment in which the components were deployed and ran. As such, ample time was spent working on this, both at the start and throughout the entire implementation.

From the code development point of view, the implementation was tackled in a similar fashion to the design, with the EJB server-side first, then the client-side. Within the EJB component phase were sub-phases of entity implementation followed by session bean façade development. The web client implementation was integrated into the EJB coding from an early stage. It was very much an iterative development process, with the website providing feedback and issues on the EJB components. In this way, working prototypes were frequently produced, and initial design flaws could be eliminated in the early stages to avoid repercussions later.

The Java application client was left till last – being the least important and originally set as optional unless time permitted. However, it offered a useful and interesting insight into EJB security, and the use of remote interfaces.

5.1 Development Methodology

With an initial incomplete understanding and knowledge of the technologies and underlying business processes involved, which would be only truly achievable through experience gained from experimentation, a very flexible approach was required. It was clear the system would evolve throughout development. However it was unclear how the system would turn out, and how the design may change. Due to these inevitable alterations and unknowns, a traditional development methodology (such as the Waterfall model) would not be suitable here.

Although formerly not followed, Agile development was the closest methodology used in the implementation. The concepts include iterative development, sub-phasing, continuous integration, refactoring, among many others. Essentially this allowed quick deployment, frequent code improvements, and rapid implementation of alterations to the design.

Inherent in EJB is the concept of Aspect-oriented programming (AOP). This involves the use of ‘aspects’ – reusable cross-cutting services such as data persistence, security, and logging. The general idea is to provide these services to alleviate ‘concerns’ that a developer may have. The EJB container, in the form of middleware services, therefore provides aspects. Other than these intrinsic EJB aspects, interceptors can be used to produce similar effects.

5.2 Environment Setup

Many programs and tools are required to develop a complete three-tier system using EJB. An application server, a version of the Java Standard Development Kit (JDK),
an IDE, a database server, and a mail server all need to interact at some level during compilation or runtime.

5.2.1 Application Server and JDK

Java Sun Application Server Platform Edition 9.0 was installed and registered with the NetBeans IDE. Once the server was setup and started, the Admin Console (accessible through a web browser) could be used to deploy enterprise (Java) applications and web applications, search logs for debugging information, create new JMS resources and connection pools, and lots more.

The Java Standard Development Kit (JDK) 5.0 was installed to provide the essential Java APIs.

5.2.2 Database and Mail Server

The Java Application Server comes with a default Java EE 5 database server; however, the MySQL database server was used instead. The basis of this decision was to give me more functionality and features on the database side.

MySQL 5.0 together with MySQL Administrator tool offered a feature rich and easy to use database and GUI coupling.

hMailServer and related administrator tool provided the mail server requirements for the application. The University of Manchester Computer Science Department’s mailing domain (t8.cs.man.ac.uk) was used. The program was configured to the MySQL database server for its required data store.

5.2.3 Integrated Development Environment Tool

NetBeans 5.5 was installed and configured with the application server, the JDK, and the MySQL. A MySQL database driver was required, and a database connection was setup using the Application Server’s Admin console.

5.3 Enterprise JavaBeans Development

The NetBeans IDE offered excellent support for EJB application development. Not only were there initialisation wizards for all bean types, but also design patterns such as session façades for entities could be created. In the latter case (and for any type of session bean), local and remote interfaces could be generated automatically.

Throughout development, various refactoring options and quick fixes were available, such as encapsulating fields to supply accessor and mutator methods, adding session bean methods to local and remote interfaces, and importing necessary projects and libraries.
The result of these features was ultimately a rapid development and automatic code-generating tool. In practice the tedious and trivial elements of EJB development were made easy on the developer. However support for advanced features of EJB tended to produce erratic behaviour and confused the matter at hand rather than simplify it.

5.3.1 Entities

Implementation of entities was essentially a mirror of the database schema produced in the design. However, EJB provided enough quirks to make what seemed a trivial task into a complex challenge requiring a great deal of time and research.

To illustrate the principles in implementing entities, consider the flight to booking relationship. It was shown in the design that a many-to-many relationship exists between these tables, with a join table sitting in-between and providing further attributes. In EJB 3.0 the @JoinTable annotation – added to the owner entity – is available for many-to-many relationships. However, this does not provide support for defining additional attributes to the join table. To overcome this limitation, a physical entity class was created to represent the join table, with one-to-many relationships coming from the Flight and Booking entities (see sources 5.1, 5.2 and 5.3).

As shown in each of these source snippets, entities must have a primary key labelled with the @Id annotation. This value is simply an auto-incremented field in the database.

Each annotation to describe the type of relationship can take various parameters. Eager loading (fetch type) has been generally used as these entities are closely related. The Booking entity defines cascade types merge and remove on its relationship with the BookingFlight entity because if a booking is updated or deleted, this should be cascaded down to any associated BookingFlight entities. Similarly, if a BookingFlight entity is updated, so should the associated Booking entity.

The mappedBy element of the relationship annotation is used by the owner entity to tell the target entity which property is used in the mapping. In the case of the Flight entity, a mappedBy value of “flight” maps to the privately stored variable of a Flight entity in the BookingFlight entity. This essentially means the BookingFlight entity stores a flight based on an instance variable called “flight”. The Flight entity itself on the other hand stores a collection of BookingFlight entities. Accessor and mutator methods are shown for each entity. Note that for each defined relationship, the entities implement these methods for one another. This makes the relationships bidirectional.
@Entity(name="Flight")
@Table(name="FLIGHT")
public class FlightEntity implements Serializable {

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

// one to many relationship with BookingFlight (join table)
@OneToMany(fetch=FetchType.EAGER, mappedBy = "flight")
private Collection<BookingFlightEntity> bookingFlights;

public Collection<BookingFlightEntity> getBookingFlights() {
    return bookingFlights;
}

public void setBookingFlights(Collection<BookingFlightEntity> bookingFlights) {
    this.bookingFlights = bookingFlights;
}

}

Source 5.1 FlightEntity.java code snippet

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

...  
/** many to one relationship with booking and flight (this entity acts as a join table between
 * the many to many relationship of booking and flight
 */
@ManyToOne(cascade={CascadeType.MERGE}, fetch=FetchType.EAGER)
private BookingEntity booking;

@ManyToOne(fetch=FetchType.EAGER)
private FlightEntity flight;

...
public BookingEntity getBooking() {
    return booking;
}

public void setBooking(BookingEntity booking) {
    this.booking = booking;
}

public FlightEntity getFlight() {
    return flight;
}

public void setFlight(FlightEntity flight) {
    this.flight = flight;
}

...

Source 5.3 BookingFlightEntity.java code snippet

5.3.2 Session Beans

As the bulk of the server-side development, session beans provided the access to entities, the entry point of most servlet communications, and the underlying business logic. Both stateless and stateful form were used, but the latter was largely not required as persistence was achieved either using entities or HTTP sessions.

As per the design, stateless session façades were implemented to provide database CRUD (Create, Read, Update and Delete) operations on entities. Source 5.4 shows a selection of the Flight entity façade containing CRUD operations. The standard read access is provided by the EntityManager’s find method that accepts the primary key as an identifier and returns an entity instance. To provide more flexible access to Flight entities, EJB-QL was implemented in assorted find methods. These can be dynamic: variable parameters are passed in and only at resolved runtime. Result lists or single results can be queried for, but an exception will be thrown if the database returns something other than what was expected (such as multiple results being returned when only one was expected). The findFromFlightNo() method makes use of this by coding a ‘?’ followed by the parameter number into the query string, and then mapping a parameter variable to replace the ‘?1’. This can be used to create
powerful queries with many parameters, but soon becomes difficult to manage and hard to debug if problems occur.

```java
@Stateless
public class FlightEntityFacade implements FlightEntityFacadeLocal,
FlightEntityFacadeRemote {

    @PersistenceContext
    private EntityManager em;

    public void create(FlightEntity flightEntity) {
        em.persist(flightEntity);
    }

    public void edit(FlightEntity flightEntity) {
        em.merge(flightEntity);
    }

    public void destroy(FlightEntity flightEntity) {
        FlightEntity managed = em.merge(flightEntity);
        em.remove(managed);
    }

    public FlightEntity find(Object pk) {
        return (FlightEntity) em.find(FlightEntity.class, pk);
    }

    public List findAll() {
        return em.createQuery("select object(o) from Flight as o order by
        o.departureDateTime").getResultList();
    }

    public FlightEntity findFromFlightNo(String flightNo) {
        return (FlightEntity) em.createQuery("select object(o) from Flight as o " +
        "WHERE o.flightNo = ?1")
            .setParameter(1, flightNo)
            .getSingleResult();
    }

    ...
}
```

**Source 5.4** FlightEntityFacade.java code snippet

Other than façades, stateless session beans were used for a variety of validations, calculations, and business processes. For example, the ValidateBookingBean provides a simple but reusable checking method to see if the customer’s chosen passenger numbers are valid (e.g. one of these checks makes sure infants are accompanied by at least one adult). The CalculateBookingFlightPriceBean implements a single method used to calculate the price of a BookingFlight entity (i.e. one flight in a booking).

The TransferFundsBean (see source 5.5) makes use of bean-managed transactions (@TransactionManagement annotation) to have complete control over when withdrawing and depositing funds from a customer’s bank account to the flight company’s account. The @Resource annotation is an example of resource referencing; in this case the UserTransaction object (from the Java Transaction API)
is injected to provide begin(), commit(), and rollback() methods. The method is
called in the final stages of the booking process, when a customer has chosen a
payment card and confirmed their booking. The customer’s card is associated with a
bank account from which the required amount is withdrawn and deposited into the
company’s own account, all within the same transaction. This process must be
carried out using a bean-managed transaction as otherwise the ACID properties of
transactions will not be guaranteed, resulting in unwanted behaviour.

```java
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class TransferFundsBean implements ejb.session.customeraccount.TransferFundsLocal {

    @Resource private UserTransaction userTx;

    @EJB
    private CustomerAccountEntityFacadeLocal customerAccountEntityFacade;

    /** the company account session bean used to access withdraw and deposit funds */
    @EJB
    private CompanyAccountEntityFacadeLocal companyAccountEntityFacade;

    public boolean transferFundsToCompanyAccount(CustomerAccountEntity customerAccountEntity,
                                                   CompanyAccountEntity companyAccount, Double amount) {
        try {
            userTx.begin();

            // withdraw the funds from the customer account
            Boolean isSuccessfulWithdrawal =
                customerAccountEntityFacade.withdrawFunds(customerAccountEntity, amount);
            if (!isSuccessfulWithdrawal) { // not sufficient funds
                userTx.rollback();
                return false;
            }

            // deposit funds into company account
            companyAccountEntityFacade.depositFunds(companyAccount, amount);

            // commit the transaction
            userTx.commit();
        } catch (Exception e) {
            System.out.println("Exception occurred during transfer of funds. " + e.getMessage());
        }

        // if successful
        return true;
    }

    Source 5.5 TransferFundsBean.java code snippet
```

The only example of a stateful session bean in the application was the shopping cart
implementation (see source 5.6). As described previously, the shopping cart stores
Flight and BookingFlight entities. The purchase() method provides the main
functionality and is where the contents of the shopping cart is finalised into a booking
entity. The process begins by retrieving the owner (customer) of the shopping cart, and also the existing Booking entity. Using the collection of stored BookingFlight entities, one (any) is used to access the associated Booking entity. From here, Booking attributes that can only be set when finalising a booking are added. The booking status element will be different depending on whether the customer chose to delay payment of booking, or paid using one of their registered payment cards. Finally the Booking entity is updated (an action that persists the alterations to the database), and the booking ID is returned for use as an identifier later.

    @Stateful
    public class ShoppingCartBean implements ShoppingCartLocal, ShoppingCartRemote {

        @PersistenceContext
        private EntityManager em;

        ... public void addFlight(FlightEntity flight) {
            // Check for an existing flight in the cart that matches the flight passed in
            ...

            // Did not find a match, so add the flight as new
            flightList.add(flight);
        }

        public void removeFlight(FlightEntity flight) {
            flightList.remove(flight);
        }

        public List<FlightEntity> getAllFlights() {
            return flightList;
        }

        ...

        public long purchase(PaymentCardEntity paymentCard) {
            // get the owner of the cart
            CustomerEntity customer = em.find(CustomerEntity.class, owner);

            /* pick one of the BookingFlight entities (does not matter which one) to retrieve
               * the associated Booking entity
               */
            ...

            if (paymentCard == null) { // payment delayed
                booking.setStatus(GlobalConstants.getPAYMENT_DELAYED());
            } else { // payment card chosen
                booking.setStatus(GlobalConstants.getPENDING());
            }

            booking.setCustomer(customer);
            booking.setBookingFlights(bookingFlightList);
            booking.setPaymentCard(paymentCard);

            // bookingEntityFacade.edit(booking);

            // Return the booking ID
            return booking.getId();
        }
    }
The web client and EJB components both run on the same Java Virtual Machine (JVM). Local interfaces are therefore suitable. In distributed systems, clients often run on a separate server, thus requiring remote interfaces for access to session beans. This is the case of the Java application client. Source 5.7 shows the class declaration of a session façade that implements both a local and a remote interface. Both interface declarations are also shown. Source 5.8 provides usage examples for clients accessing these two interfaces.

```java
@Stateless
public class FlightEntityFacade implements FlightEntityFacadeLocal, FlightEntityFacadeRemote

@Local
public interface FlightEntityFacadeLocal

@Remote
public interface FlightEntityFacadeRemote
```

Source 5.7 FlightEntityFacade.java, FlightEntityFacadeLocal.java, FlightEntityFacadeRemote.java code snippets

Sending an email is an ideal candidate for a message-driven bean due to its asynchronous communication. This allows the online booking client, which needs to send numerous emails to its customers, to simply send a message to a JMS destination and then forget about it and carry on its business.

Source 5.9 shows a MDB that consumes JMS messages containing all the required email details (recipient, subject, body text, host, etc.). Messages are handled by the container, which invokes the onMessage() method. Several types of messages exist, but in this case ObjectMessages are received. This message contains an Email object (a simple serializable Java class implementing the attributes of an email), whose values are extracted in the sendEmail() method, which uses the JavaMail API to build and send an email.

The @MessageDriven annotation takes some values in the example. ‘mappedName’ refers to the destination resource (setup manually beforehand using the admin console...
of the Java Application Server) that consumes the JMS messages, namely, this MDB. The 'activationConfig' value is used to specify the destination type; in this case it the MDB will consume messages from a queue.

```java
@MessageDriven(mappedName = "jms/SendEmailBean", activationConfig = {
  @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue")
})
public class SendEmailBean implements MessageListener {
  ...

  public void onMessage(Message message) {
    // extract the email object and call the send method
    ObjectMessage objectMessage = (ObjectMessage) message;
    try {
      // call sendEmail method with extracted parameters
      Email email = (Email) objectMessage.getObject();
      sendEmail(email);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void sendEmail(Email email) {
    // attempt to send the email
    try {
      // extract the attributes from the email object
      ...

      // Get system properties
      Properties props = System.getProperties();

      // Setup mail server
      props.put("mail.smtp.host", host);

      // Get session
      Session session = Session.getDefaultInstance(props, null);

      // Define message
      MimeMessage message = new MimeMessage(session);
      message.setFrom(new InternetAddress(from));
      message.addRecipient(RecipientType.TO, new InternetAddress(to));
      message.setSubject(emailSubject);
      message.setText(emailText);

      // Send message
      Transport.send(message);
    } catch (Exception e) {
      ...
    }
  }
  ...

Source 5.9 SendEmailBean.java snippet
The above MDB is sent messages from the BookingEmailBean (source 5.9), among other beans. This stateless bean creates and sends an ObjectMessage after preparing an Email object with the necessary details. Two resources are required as specified by the annotations: the connection factory (also setup using the admin console) is used to create the JMS messages, while the queue matches the mappedName in the MDB.

```java
@Stateless
public class BookingEmailBean implements ejb.session.email.BookingEmailLocal {
    @Resource(mappedName = "jms/SendEmailBeanFactory")
    private ConnectionFactory sendEmailBeanFactory;

    @Resource(mappedName = "jms/SendEmailBean")
    private Queue sendEmailBean;

    public void buildAndSendBookingEmail(String emailAddress, long bookingID) {
        // prepare email confirmation of order
        try {
            sendJMSMessageToSendEmailBean(bookingEmail);
        } catch (JMSException ex) {
            ex.printStackTrace();
        } catch (NamingException ex) {
            ex.printStackTrace();
        }
    }

    private Message createJMSMessageForSendEmailBean(Session session, Object messageData) throws JMSException {
        // extract email data from messageData object
        Email bookingEmail = (Email) messageData;

        // here we create an object message, that will be sent in JMS message
        ObjectMessage message = session.createObjectMessage();
        message.setObject(bookingEmail);

        return message;
    }

    private void sendJMSMessageToSendEmailBean(Object messageData) throws NamingException, JMSException {
        Connection connection = null;
        Session session = null;
        try {
            connection = sendEmailBeanFactory.createConnection();
            session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageProducer messageProducer = session.createProducer(sendEmailBean);
            messageProducer.send(createJMSMessageForSendEmailBean(session, messageData));
        } finally {
            if (session != null) {
                session.close();
            }
            if (connection != null) {
                connection.close();
            }
        }
    }
```
5.3.4 Bean Process Flows

The flow of control and dependency between beans is very important in this application as it defines the structure and separation of components into business processes. The beans have been implemented in such a way as to model the client processes. Figure 5.1 follows the booking business process from the server-side perspective. ‘SLSB’ and ‘SFSB’ refer to stateless and stateful session beans respectively. For clarity, not all entities are presented in the diagram. The flow begins with the user adding a flight to their shopping cart (assume they have already logged in), a process that invokes the CalculateBookingFlightPrice stateless session bean. Various validations are then performed, before the customer chooses a payment card and confirms the order. Upon confirmation, an email is sent to the customer and funds from their account are transferred to the company’s own account. Finally the Booking entity is updated with all necessary details.
Also within the web client domain, the customer is able to add a new payment card. The process is illustrated in figure 5.2 and is of interest as it demonstrates the use of an EJB Timer. Immediately after creation of a payment card, the WarnOfExpiredPaymentCard stateless session bean starts a timer set to expire on the card expiration date. The timer also stores the customer’s email address using the ‘serializable info’ parameter. Upon expiration, the timer will call a callback to the method annotated with @Timeout in the WarnOfExpiredPaymentCard bean. This method implements code to send an email to the customer warning them of their card expiration.

Figure 5.2 Payment card added and timer process flow.

The transfer of funds is part of the booking process shown in figure 5.1, but can be expanded to explain the inner flow. Figure 5.3 shows this, starting with the PayForBooking bean and ending with the TransferFunds bean. The façades and entities are included for a more complete picture.
5.4 Website Development

The customer-oriented web client was implemented in conjunction with the server-side components. Prototypes were produced rapidly to allow maximum feedback on the EJB components. This feedback was used to enhance the server-side, add new beans, and fix issues. This was ultimately the focus – the actual website was secondary to the EJBs.

5.4.1 Servlets

The implementation of servlets was limited to direct user interface relevance – i.e. business logic was separated out as far as possible. A typical example would extract the necessary data from the request (and the session if appropriate), call an EJB validation component, perform some sort of database CRUD operation using a session façade, then pass control over to a JSP.
Source 5.11 below contains a selection of the servlet used when a new customer registers their details with the flight booking website. The first thing to notice is that the servlet is simply a normal Java class, which extends the HttpServlet base class. Several methods can be overridden, but the main one in this servlet is the processRequest() as illustrated. This method is passed request and response objects to manipulate before passing control to another servlet or JSP. In this case, the method begins by checking the request for parameters related to an HTML form. If these customer details are present, then the process of creating a customer entity is started. The required session façades are all accessed via local interfaces (@EJB annotations).

The potential customers’ details are validated, which includes checking the email address by attempting to send a ‘Welcome’ email. If all is successful, an HTTP session is created, the customer is logged into the site, and a shopping cart is created for the customer. Depending on these successes or failures, the servlet forwards on the request and response to another servlet or a JSP for direct user feedback (JSP will display a webpage).

```java
public class AddCustomerServlet extends HttpServlet {

    @EJB
    private CustomerEntityFacadeLocal customerEntityFacade;

    /** Processes requests for both HTTP GET and POST methods. */
    @param request servlet request
    @param response servlet response
    */
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // check for addCustomer parameter
        String hasSubmitted = request.getParameter("addCustomer");
        if (hasSubmitted != null) { // user has submitted the form
            // retrieve the form details
            String title = request.getParameter("title");
            ...
            // here we create CustomerEntity
            ...
            // call session bean to validate customer details
            boolean isCustomerValid = validateCustomerBean.isNewCustomerValid(customer);
            if (isCustomerValid) {
                // check for a valid email address by trying to send a welcome email
                ...
                if (errorMessage != null) { // email address invalid
                    ...
                } else { // email address OK
                    customerEntityFacade.create(customer);
                    ...
                    // now log the customer in
                    HttpSession session = request.getSession(true); // Set up the user's HttpSession
                    session.setAttribute("customer", customer); // store the customer in the session
                    // new Shopping Cart Session Bean, add it to the user's HttpSession object
                    ...
                }
            }
        }
    }
}
```
5.4.2 Java Server Pages

The JSPs developed also followed a common structure: a header file containing top navigation links and a secondary include file for side navigation were included in every page, then either tables or forms (styled with external cascading style sheets) followed. In some cases JavaScript was added for perform HTML DOM manipulations (e.g. changing the ‘disable’ state of a button).

Source 5.12 shows a snippet of the JSP file used to display final webpage in the booking process. Littered throughout the file are pieces of java code used to manipulate the data and communicate with EJBs. Notice the import statement at the top of the source, which is needed to refer to Flight entities. The lines after this are all presentation and standard HTML, until the centre content is reached. It is here that the Java code retrieves the list of flight from the request and accesses each Flight entity to display in the HTML table.

```html
<%@ page import="ejb.entity.FlightEntity" %>
...
<html>
<head>
<link href="../styles/style.css" rel="stylesheet" type="text/css">
...
</head>
<body>
<div id="header">
<!-- Include the header file, which contains navigation toolbar style--%>
<jsp:include page="../headers/header.jsp" />
</div>

<div id="leftNavigation">
<!-- Include the side navigation file--%>
<jsp:include page="../includes/bookingNavigation.jsp" />
</div>

<div id="centerContent">
<h2>Booking Details</h2>
<h3>Flight Details</h3>
<table border="1" cellspacing="0">
<% java.util.List flights=(java.util.List)request.getAttribute("flightList"); if (flights != null) {
... for (java.util.Iterator it = flights.iterator(); it.hasNext();) {
    FlightEntity elem = (FlightEntity) it.next();
    // departure and arrival
    String departureDateTime = GlobalConstants.getDATE_TIME_FORMAT().format(elem.getDepartureDateTime());
```
5.4.3 Security

The web client is limited to authentication via a login form, with restricted servlets containing login logic to check if the customer has been added to the current HTTP session. The actual authentication process is carried out by the customer entity façade using a method that takes the email address and password (credentials), and tries to find a customer in the database with those exact details.

5.5 Java Application Development

A Java application client for use by administrators was initially only planned for development if time permitted. However, once implementation of the website was well under way, it was clear a further client could reuse the existing EJBs to great effect. The client itself is, from a functional and GUI perspective, quite limited; this was entirely intentional. A development on the scale of the website client was not wanted, as it would have ended up becoming a replica with the same features only on a different platform. Rather than repeat tedious development and EJB features, new techniques – solely available to application clients – were implemented.

The main features used in the Java application client were security and the use of remote interfaces. Remote interfaces have already been discussed, but security is very different from the web security used in the website client.

5.5.1 Security

In the same way as the web client, the Java application client requires the user to provide credentials in the form of a username and password. A login form was implemented that shows upon start-up. The code implemented standard JAAS authentication procedures:

- A configuration file was created and stored client-side with an entry detailing the path of the customised password login module.
- A CallbackHandler was created to prompt for username and password.
- The PasswordLoginModule Java file implemented login(), commit(), abort(), and logout() methods:
  - login() retrieves username and password (using the CallbackHandler) and attempts to verify the credentials by looking up existing users in the database (these have been added previously).
  - commit() checks for successful login then creates a principle (stores user credentials) and add it the subject (representation of the user).
- abort() provides clean-up logic (such as clearing user credentials).
- logout() removes the principal from the subject and clear the user’s password.
- A login context was created by passing in the name mapped to the entry in the configuration file, and an instance of the CallbackHandler.
- With this context, attempts to login were made by calling the login modules login() method.

The configuration file was referenced using the NetBeans run options. “-Djava.security.auth.login.config=<filepath>.conf” was added as a Virtual Machine (VM) argument (with the appropriate file path included).

### 5.5.2 Graphical User Interface – Swing Framework

The NetBeans IDE provided excellent support for the GUI development. Matisse, a GUI builder for NetBeans, offered layout and component visualisation tools to aid screen design. The GUI components themselves were from the Java Swing framework, which keeps in form with the overall MVC design.
Chapter 6: Results

6.1 Deployment

The NetBeans IDE carried out deployment of EJB components and clients automatically. EJB components were deployed to the application server under the installation directory - C:\Sun\SDK\domains\domain1\applications\. Web components (servlets) were accessible from a web browser pointing to the following address: http://localhost:8080/FlightBookingsApp-war/.

6.2 Web Client

The customer-oriented online booking system defaults to a homepage (see screenshot 6.1) from which other areas of the site are navigated by use of top and side links. Quick links to the homepage and login pages (changes to a logout link once user has logged in) are provided at the top of the page for easy access. The main areas are ‘Search Flights’, ‘Your Bookings’, ‘Your Account’, and ‘Your Cart’.

![Screenshot 6.1 Homepage](image)

For new customers, a sign-up form is provided (screenshot 6.2) that asks for essential details including a unique email address (no two customers can have the same email address) and a password (does not have to be unique). Upon sign-up, the customer is automatically logged in.
A typical customer action is searching for flights – the initial stage in making a booking (although the customer does not have to sign-in for this). Screenshot 6.3 shows the search flights form with all the options and required fields filled in. Drop-down lists are used for ease of use and to limit the customer’s text entry. Upon searching, various validation rules are performed to ensure appropriate flights can be found.

Screenshot 6.4 displays the results of a search for both outbound and return flights. Base prices are given, and it is from this screen that customers can pick and choose flights to add to their shopping basket. If the customer is not logged in and tries to add flights to their cart, the client will redirect them to login page (screenshot 6.5). This requires the customer’s email address and password. A link is provided to a form for sending the password (if forgotten) to the associated email address. Upon successful login, the customer will be forwarded directly to the cart details page (screenshot 6.6), as this was the page they would have progressed to if they had been logged in. The cart items can be updated by changing the passenger types and numbers, and can be removed. Once they are happy with their cart, they can click the ‘Book Contents Of Cart’ button to take them to the next stage: choosing a payment card (screenshot 6.7). This page allows the customer to add a new card, select an existing card (previously registered), or pay later. The first of these three options takes the customer to the ‘Add New Payment Card’ form (screenshot 6.8). The other two choices take the customer to the ‘Confirm Booking’ page (screenshot 6.9), where the flight details are displayed, and – in the case of selecting a payment card in the previous step – the payment card of choice is shown. Once confirmed, the customer is forwarded to a page listing their booking and all their previously made bookings.
The page enables the customer to update and cancel bookings (as long as the earliest flight date is in the future). A ‘Pay For Booking’ option is also available if the selected booking has not yet been paid for. This will take the user to the ‘Choose A Payment Card’ page as described before. Screenshot 6.11 illustrates the form for updating a booking. The overall booking details are displayed, and the individual flights can be updated or cancelled.

Screenshot 6.12 shows the ‘Update Your Personal Details’ form, which essentially the customer to alter any details chosen during sign-up.

The ‘Update A Payment Card’ form (screenshot 6.13) follows the same format as the ‘Add New Payment Card’ form, but initialises with existing values for convenience.

Emails are generated and send to customers in a number of cases. Screenshot 6.14 shows an email received by a customer who has just signed-up. If this customer goes on to make a booking, they will receive an email – such as the one in screenshot 6.15 – detailing their purchase and how they paid for it. For customers who have registered one or more payment cards on the site, they will be warned of expiration via email (screenshot 6.16).
Screenshot 6.3 Search Flights form

Screenshot 6.4 Flight listing (search results)
Screenshot 6.5 Login form

Screenshot 6.6 Cart details page
Screenshot 6.7 Choose a payment card page

Screenshot 6.8 Add a new payment card page
**Booking Details**

<table>
<thead>
<tr>
<th>Flight No.</th>
<th>Country</th>
<th>Departure</th>
<th>Date/Time</th>
<th>Arrival</th>
<th>Country</th>
<th>Airport</th>
<th>Date/Time</th>
<th>Rate</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>MA1234</td>
<td>United Kingdom</td>
<td>MAN</td>
<td>10 Dec 06 18:00</td>
<td>United Kingdom</td>
<td>LGW</td>
<td>10 Dec 06 22:00</td>
<td></td>
<td>20.00</td>
<td></td>
</tr>
<tr>
<td>MA2372</td>
<td>United Kingdom</td>
<td>LGW</td>
<td>13 Dec 06 18:00</td>
<td>United Kingdom</td>
<td>MAD</td>
<td>13 Dec 06 22:00</td>
<td></td>
<td>20.00</td>
<td></td>
</tr>
</tbody>
</table>

**Payment Card Details**

<table>
<thead>
<tr>
<th>Billing Country</th>
<th>Card Type</th>
<th>Card No</th>
<th>Start Date</th>
<th>Expiry Date</th>
<th>Name On Card</th>
<th>Billing Address</th>
<th>Security No</th>
</tr>
</thead>
<tbody>
<tr>
<td>United Kingdom</td>
<td>Solo</td>
<td>4721 1234 5678 9012</td>
<td>01/03/06</td>
<td>01/03/10</td>
<td>MRS ISABELLE JONES</td>
<td>Flat 11, Bramen Road, Manchester</td>
<td>789</td>
</tr>
</tbody>
</table>

Screenshot 6.9 Booking details page - confirm booking
Screenshot 6.10 Booking list page

Screenshot 6.11 Update A Booking Process
Screenshot 6.12 Update Your Personal Details form

Screenshot 6.13 Update A Payment Card form
Screenshot 6.14 Welcome email for new customer

Max's Flight Booking - Welcome!

From: "hydem03@cs.man.ac.uk" <hydem03@cs.man.ac.uk>
To: hydem03@gmail.com
Date: May 1, 2007 5:36 PM
Subject: Max’s Flight Booking - Welcome!

Dear Mr. Samuel Jackman,

Welcome to Max’s Flight Booking Online Website! Thank you for signing-up on 01 May 07 at 17:35 via the online 'sign-up' form. See below for your password and a link to the homepage of our site.

Your password: sj
Site homepage: http://localhost:8080/FlightBookingsApp-war/homepage/homepage.jsp

Reply | Reply to all | Forward | Invite hydem03@cs.man.ac.uk to Google Mail

Connecting to www.google.com...

Screenshot 6.15 Booking confirmation email

Max’s Flight Booking Confirmation

From: "hydem03@cs.man.ac.uk" <hydem03@cs.man.ac.uk>
To: hydem03@gmail.com
Date: May 1, 2007 9:26 PM
Subject: Max’s Flight Booking Confirmation

Dear Mr. Samuel Jackman,

Your booking ID: 6002
Booking was made on 01 May 07 at 20:24
The sub total price of your booking is 1567.8, which has been charged to your payment card with number ending in 9891
Bookings current status: Pending (you can check the progress of your booking by logging into your account with us at http://localhost:8080/FlightBookingsApp-war/login/LoginServlet

Flight Number: MA6777
Departing From: Aberdeen
Departing At: 06 May 07 16:30
Destination: London Luton
Arriving At: 06 May 07 19:30
Flight Duration: 03hrs 10mins
Number Of Passengers: 7
Adults: 2
Children: 3
Infants: 2
Flight Total Price: 623.0

Flight Number: MA6792
Departing From: London Luton
Departing At: 13 May 07 13:00
Destination: Aberdeen
Arriving At: 31 May 07 16:00
Flight Duration: 02hrs 10mins
Number Of Passengers: 3
Adults: 1
Children: 1
Infants: 1
Flight Total Price: 934.5

Reply | Reply to all | Forward | Invite hydem03@cs.man.ac.uk to Google Mail

Looking up www.google.com...
Expired payment card warning email

6.3 Java Application Client

The administration client initialises with a login form asking for user credentials. Screenshot 6.17 and 6.18 illustrate a valid and invalid login respectively. If the login is valid, the application begins to load data from the database in preparation for the main screens being displayed.

Screenshot 6.19 displays the form for adding flights. Drop-downs are provided for ease of use. A tabbed approach has been implemented with similar styles and naming for each tab element.

Screenshot 6.20 shows the flight-listing table with a flight selected for editing.

Screenshot 6.17 Login form with valid data
Screenshot 6.18 Login form with invalid data

Screenshot 6.19 Add flight form
### Flight Booking Admin Client

**Flight list with update ability**

<table>
<thead>
<tr>
<th>ID</th>
<th>Flight Number</th>
<th>Country of Dep</th>
<th>Departure Airport</th>
<th>Country of Arr</th>
<th>Arrival Airport</th>
<th>Departure Date</th>
<th>Arrival Date</th>
<th>Base Price</th>
<th>Aircraft</th>
<th>Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>2A3J431</td>
<td>United King</td>
<td>***** *****</td>
<td>***** *****</td>
<td>***** *****</td>
<td>***** *****</td>
<td>20 Dec 06</td>
<td>30 Dec 06</td>
<td>38</td>
<td>Boeing 777</td>
<td></td>
</tr>
<tr>
<td>78911354</td>
<td>United King</td>
<td>***** *****</td>
<td>***** *****</td>
<td>***** *****</td>
<td>***** *****</td>
<td>20 Dec 06</td>
<td>30 Dec 06</td>
<td>38</td>
<td>Boeing 777</td>
<td></td>
</tr>
<tr>
<td>34567689</td>
<td>United King</td>
<td>***** *****</td>
<td>***** *****</td>
<td>***** *****</td>
<td>***** *****</td>
<td>20 Dec 06</td>
<td>30 Dec 06</td>
<td>38</td>
<td>Boeing 777</td>
<td></td>
</tr>
<tr>
<td>90123456</td>
<td>United King</td>
<td>***** *****</td>
<td>***** *****</td>
<td>***** *****</td>
<td>***** *****</td>
<td>20 Dec 06</td>
<td>30 Dec 06</td>
<td>38</td>
<td>Boeing 777</td>
<td></td>
</tr>
<tr>
<td>56789012</td>
<td>United King</td>
<td>***** *****</td>
<td>***** *****</td>
<td>***** *****</td>
<td>***** *****</td>
<td>20 Dec 06</td>
<td>30 Dec 06</td>
<td>38</td>
<td>Boeing 777</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 7: Testing and Evaluation

Testing was carried out throughout the project, both formally and informally. On the fly testing and debugging was possible as soon as client implementation began and EJB components could be deployed and run. As more and more test data was entered into the system, tests became more meaningful. Common tests included:

- Ensuring database CRUD operations (performed by session bean façades) successfully read, persisted or deleted data from the database
- Control flow through the website booking process was correct
- HTTP sessions contained the correct data
- JavaScript functions performed as expected
- Validation was performed and executed correctly
- Ensuring exceptions were did not occur
- Pricing engine worked correctly
- EJB-QL code executed without error and returned results as expected.

The NetBeans IDE debugging tools were very useful for inspecting variables and stepping through code to find the exact point of error. The Java Application Server was also used, as a console is available to search log files detailing warnings and exceptions.

Later phases of testing were carried out to provide a more complete view of whether the system

- Met the original requirements
- Was reliable and robust
- Performed accurately.

The test plan for both clients is detailed in appendix C. Tests were carried at a component level, with pre-conditions and post-conditions, and inputs and outputs defined. The tests combine several different types of testing including

- Compliance testing: to ensure the system fulfilled the initial user requirements. For the web client, this involved entering valid data into forms and searches to ensure that the product produced the expected output. For the Java administration client, this consisted of executing the various CRUD operations available.
- Destructive testing: to test the robustness and reliability of the system. The intention was to make the application crash or produce errors, by entering invalid data into forms and search facilities. To pass these tests, the system had to produce user-friendly error messages, detailing the reason for the error and how the user could fix it. Also, the system had to allow the user to correct their mistake/s, by automatically taking them back to the form/search they were filling in, and reproducing the original data.
- Exhaustive testing: for the website, exhaustive testing involved navigating through every page, making sure that links and buttons worked. For the Java application client, every button was tested for the expected outcome.
7.1 Test Analysis and Evaluation

The majority of testing occurred during development and by using the clients to gain an insight into the server-side components. The NetBeans IDE debugging tool was essential as it enabled controlled access to EJB components, and allowed the monitoring of process flow through the system.

Considering the Agile development approach taken to implementation, it was extremely disappointing and unfortunate that unit testing was not undertaken. Frequent iterations of EJB component testing would have been ideal, but due to the lack of support for EJB 3.0 and the fact that the project was essentially tied down to the NetBeans IDE, it was found to be exceptionally difficult and time consuming to find an appropriate testing framework.

The test carried out were ultimately one off, manual, and not rerunnable. This was far from the best-case scenario, but did offer a deeper understanding of the remaining bugs and issues left in the system (after the continuous informal testing throughout development). Specifically, it illustrated the failings of the web client in coping with the intricate control flow throughout the system and between servlets and JSPs, and the success of the EJB components. The majority of problems were client related – easily fixable and very visible. Mostly, GUI form validation, forwarding requests within the web client structure, and general inconsistency of code and behaviour were the main issues. None of these were high priority during development as focus was around the EJB server-side. Considering small common irregularities, such as the data for fields in forms not being reproduced upon returning after failure, were found, a Web framework (e.g. JavaServer Faces) may have been helpful.

By concentrating on the underlying EJB component code, the tests used on both clients were representative of the success or failure of the server-side beans. The clients ensured that all the beans and methods were invoked at some point during their business processes (e.g. booking a flight). This was very useful and added value to any client tests. By making sure the cause of failed tests was fixed immediately, the eventual success of all the tests suggest the initial system requirements have, to a large extent, been met.

Overall, the formal testing was limited by the manual processes involved. Fortunately, these tests were performed after completion of coding, suggesting the final system worked sufficiently. Furthermore, the recurrent informal testing during implementation secured a reliable server-side.
Chapter 8: Conclusions

Undertaking this project was a huge and important learning experience. EJB 3.0 is a heavyweight technology often used to build massive distributed systems. To use this powerful framework, many hours were spent trying to understand its inner workings.

The initial stage of researching into software components was highly useful and at the same time greatly interesting. Although the concepts behind Component-Based Software Development are not new, it is only relatively recently that it has caught on in a very big way. With major vendors pushing their component frameworks forward, what has been for a long time an ideal held mostly in the academic domain could become the future for many computing industries.

The design and implementation of a flight booking system could prove challenging in any setting. With limited time and computing resources, a working system was developed using EJB and other technologies previously unknown to the author. While feature rich clients were not created, the server side made use of a large range of EJB 3.0 concepts and design patterns wrapped up in all different types of bean components.

8.2 Initial Objectives Evaluated Against Final System

To a large extent, the project successfully met and even exceeded its initial objectives. The EJB 3.0 technology was learnt and understood to the point at which all necessary requirements could be implemented. The flight booking system produced consisted of a reusable suite of components, a customer oriented web-client, and an external Java application client. With both clients reusing the same server-side components, the power and principles of software component-based development was illustrated.

The project plan outlining the timescales of the different tasks and phases was followed up to a point. Due to the expected ever-changing nature of the implementation and the chosen development style (Agile), the system design could not be carried forward without alterations. Specific difficulties and unknowns occurred mostly in server-side environment setup and configuration, thus affecting component development. The initial effect of this essentially pushed all milestones backwards to later dates. However, subsequent to these teething problems, on the whole, development proved to be rapid and without complication. As a general observation for this project: whenever a new and previously untried feature of EJB 3.0, or a new and unknown technology was used, problems and issues were encountered.

8.3 Further Developments

While a lot has been achieved, encompassing a large proportion of the features available in EJB 3.0, and going some way into producing two functional flight booking clients, there is still much scope for development.
In terms of further server-side work, there is ample scope to add an assortment of interceptors, EJB Timers, and EJB-QL queries to implement new client features. While the system produced in the project is limited to academic use, in the commercial world a similar product would require performance optimisations, clustering, possible integration with other systems, and many more considerations that go beyond the scope and requirements of this particular project. Even so, EJB 3.0 offers support for many of these matters. To achieve the best performance, tuning beans, servers, and JVMs are all crucial activities. For systems to scale well and provide high availability, load balancing and failover techniques can be applied to each type of enterprise bean. To integrate EJB with other systems, various approaches and best practices are documented.

While EJB 3.0 is a new technology, vendors are developing and releasing ever-newer products, and updates to existing ones. This is a double-edged sword for consumers as on the one hand they are provided with an excellent choice and competitive range of products but on the other hand they incur immense cost – financially and in terms of development time – when switching to a new release. In many industries, working with cutting-edge technologies is the only way to stay in business. In any case, migration to new versions of EJB will be an inevitable part of a system’s life for the foreseeable future.

As far as the client-side is concerned, the project provided a bare bones solution. With incredible advances in web technology, the customer-oriented client has enormous potential for improvement of presentation and related features.

At a more abstract level, the underlying EJB component system could be reused in other logistics settings. For example, if a transportation company were interested in knowing the costs and flight durations for carrying goods by aeroplane, the components of the flight booking system could provide a basis for a solution.

8.4 In Closing

While EJB is based on decade-old concepts, EJB 3.0 is a relatively new technology full of cutting-edge, clever, and interesting ideas. This makes it highly powerful, but at the same time immensely complicated. This goes for any applications implemented using this framework. The flight booking system produced was a small application that tried to do a lot with EJB. It was packed full of features, yet from the outside, from a user’s perspective, it looked quite simple. Currently, this is often the price of building and using software components. So although an application may take longer to develop using EJB compared with a more flexible technology, the benefits can outweigh the costs in the long run. It is therefore vital to choose the most suitable approach from the very start of any project, considering the future of the application and its’ users. Whether EJB is a good choice will depend on a multitude of factors, ranging from technological requirements such as the need for scalability and distributed components, to whether they is enough EJB experience in the work force available.
References


## Appendix A: Project Plan

<table>
<thead>
<tr>
<th>Tasks</th>
<th>2006 Wks</th>
<th>2007</th>
</tr>
</thead>
<tbody>
<tr>
<td>Research</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search literature</td>
<td>2</td>
<td>38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18</td>
</tr>
<tr>
<td>Read software component literature</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Read EJB literature</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Software experimentation</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Environment setup</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clients’ requirements</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>EJB requirements</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Map requirements</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design Phase 1 - Database</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Reading week</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Poster submission</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design Phase 2 - Clients</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Design Phase 3 - EJB</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation Phase 1 - Database</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>End of Phase - Database</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seminar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vacation and exams</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Implementation Phase 2 - EJB</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Implementation Phase 3 - Web Client</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Implementation Phase 4 - Java App. Client</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing Phase 1 - EJB</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Testing Phase 2 - Clients</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Complete practical work</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Demonstration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Easter vacation</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Max Hyde

Page 86

5/2/2007
<table>
<thead>
<tr>
<th>Task</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report writing</td>
<td>6</td>
</tr>
<tr>
<td>Draft report for supervisor</td>
<td></td>
</tr>
<tr>
<td>Report submission</td>
<td></td>
</tr>
</tbody>
</table>

**Key**
- milestone
- work time
- non-work time
Appendix B: Database Tables

Admin Client User Entity

DROP TABLE IF EXISTS `flight_booking`.`adminclientuser`;
CREATE TABLE  `flight_booking`.`adminclientuser` (
    `ID` bigint(20) NOT NULL,
    `TITLE` varchar(255) default NULL,
    `USERNAME` varchar(255) default NULL,
    `FORENAME` varchar(255) default NULL,
    `PASSWORD` tinytext,
    `SURNAME` varchar(255) default NULL,
    `USERTYPE` varchar(255) default NULL,
    PRIMARY KEY  (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

Aeroplane Entity

DROP TABLE IF EXISTS `flight_booking`.`aeroplane`;
CREATE TABLE  `flight_booking`.`aeroplane` (
    `ID` bigint(20) NOT NULL,
    `BUSINESSCLASSCAPACITY` int(11) default NULL,
    `FIRSTCLASSCAPACITY` int(11) default NULL,
    `MAKE` varchar(255) default NULL,
    `TOTALPASSENGERCAPACITY` int(11) default NULL,
    `ECONOMYCLASSCAPACITY` int(11) default NULL,
    `LUGGAGECAPACITY` double default NULL,
    `MODEL` varchar(255) default NULL,
    `LASTSERVICEDATE` date default NULL,
    PRIMARY KEY  (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

Airport Entity

DROP TABLE IF EXISTS `flight_booking`.`airport`;
CREATE TABLE  `flight_booking`.`airport` (
    `ID` bigint(20) NOT NULL,
    `FULLNAME` varchar(255) default NULL,
    `CITY` varchar(255) default NULL,
    `SHORTNAME` varchar(255) default NULL,
    `COUNTRY` varchar(255) default NULL,
    PRIMARY KEY  (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

Booking Entity

DROP TABLE IF EXISTS `flight_booking`.`booking`;
CREATE TABLE  `flight_booking`.`booking` (
    `ID` bigint(20) NOT NULL,
    `TAX` double default NULL,
    `STATUS` varchar(255) default NULL,
    `SUBTOTAL` double default NULL,
    `BOOKINGTIME` datetime default NULL,
    `CUSTOMER_ID` bigint(20) default NULL,
    `PAYMENTCARD_ID` bigint(20) default NULL,
    PRIMARY KEY  (`ID`),
    KEY `FK_BOOKING_CUSTOMER_ID` (`CUSTOMER_ID`),
    KEY `FK_BOOKING_PAYMENTCARD_ID` (`PAYMENTCARD_ID`),
    .
)
CONSTRAINT `FK_BOOKING_CUSTOMER_ID` FOREIGN KEY ("CUSTOMER_ID") REFERENCES `customer` ("ID"),
CONSTRAINT `FK_BOOKING_PAYMENTCARD_ID` FOREIGN KEY ("PAYMENTCARD_ID") REFERENCES `paymentcard` ("ID")
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

Booking Flight Entity

DROP TABLE IF EXISTS `flight_booking`.`bookingflight`;
CREATE TABLE  `flight_booking`.`bookingflight` (
  'ID' bigint(20) NOT NULL,
  'NUMBEROFINFANTPASSENGERS' int(11) default NULL,
  'FLIGHTTOTALPRICE' double default NULL,
  'NUMBEROFADULTPASSENGERS' int(11) default NULL,
  'NUMBEROFCHILDPASSENGERS' int(11) default NULL,
  'FLIGHTCLASSTYPE_ID' bigint(20) default NULL,
  'BOOKING_ID' bigint(20) default NULL,
  'FLIGHT_ID' bigint(20) default NULL,
  PRIMARY KEY  ('ID'),
  KEY `FK_BOOKINGFLIGHT_FLIGHTCLASSTYPE_ID` ("FLIGHTCLASSTYPE_ID"),
  KEY `FK_BOOKINGFLIGHT_BOOKING_ID` ("BOOKING_ID"),
  CONSTRAINT `FK_BOOKINGFLIGHT_BOOKING_ID` FOREIGN KEY ("BOOKING_ID")
  REFERENCES `booking` ("ID"),
  CONSTRAINT `FK_BOOKINGFLIGHT_FLIGHTCLASSTYPE_ID` FOREIGN KEY ("FLIGHTCLASSTYPE_ID") REFERENCES `flightclasstype` ("ID"),
  CONSTRAINT `FK_BOOKINGFLIGHT_FLIGHT_ID` FOREIGN KEY ("FLIGHT_ID")
  REFERENCES `flight` ("ID")
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

Company Account Entity

DROP TABLE IF EXISTS `flight_booking`.`companyaccount`;
CREATE TABLE  `flight_booking`.`companyaccount` (  
  'ID' bigint(20) NOT NULL,
  'SORTCODE' varchar(255) default NULL,
  'ACCOUNTNUMBER' varchar(255) default NULL,
  'BANKNAME' varchar(255) default NULL,
  'BALANCE' double default NULL,
  PRIMARY KEY  ('ID')
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

Customer Entity

DROP TABLE IF EXISTS `flight_booking`.`customer`;
CREATE TABLE  `flight_booking`.`customer` (  
  'ID' bigint(20) NOT NULL,
  'EMAILADDRESS' varchar(255) NOT NULL,
  'PASSWORD' varchar(255) NOT NULL,
  'TITLE' varchar(255) default NULL,
  'CLASSTYPE' varchar(255) default NULL,
  'FORENAME' varchar(255) default NULL,
  'LASTLOGIN' bigint(20) default NULL,
  'ADDRESS' varchar(255) default NULL,
  'SURNAME' varchar(255) default NULL,
  PRIMARY KEY  ('ID'),
  UNIQUE KEY `UNQ_CUSTOMER_0` ("EMAILADDRESS")
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
Customer Account Entity

DROP TABLE IF EXISTS `flight_booking`.`customeraccount`;
CREATE TABLE  `flight_booking`.`customeraccount` (   `ID` bigint(20) NOT NULL,
    `ACCOUNTNUMBER` varchar(255) default NULL,
    `BALANCE` double default NULL,
    `SORTCODE` varchar(255) default NULL,
    `BANKNAME` varchar(255) default NULL,
    `CUSTOMER_ID` bigint(20) default NULL,

    PRIMARY KEY  (`ID`),

    KEY 'FK_CUSTOMERACCOUNT_CUSTOMER_ID' (`CUSTOMER_ID`),

    CONSTRAINT `FK_CUSTOMERACCOUNT_CUSTOMER_ID` FOREIGN KEY ('CUSTOMER_ID') REFERENCES `customer` ('ID')

) ENGINE=InnoDB DEFAULT CHARSET=latin1;

Flight Entity

DROP TABLE IF EXISTS `flight_booking`.flight;
CREATE TABLE  `flight_booking`.flight (   `ID` bigint(20) NOT NULL,
    `CURRENTECONOMYCLASSCAPACITY` int(11) default NULL,
    `CURRENTBUSINESSCLASSCAPACITY` int(11) default NULL,
    `FLIGHTNO` varchar(255) NOT NULL,
    `CURRENTFIRSTCLASSCAPACITY` int(11) default NULL,
    `ARRIVALDATETIME` datetime default NULL,
    `CURRENTPASSENGERCAPACITY` int(11) default NULL,
    `BASEPRICE` double default NULL,
    `FLIGHTDURATION` datetime default NULL,
    `DEPARTUREDATETIME` datetime default NULL,
    `AEROPLANE_ID` bigint(20) default NULL,
    `DESTINATION_ID` bigint(20) default NULL,
    `DEPARTINGFROM_ID` bigint(20) default NULL,

    PRIMARY KEY  (`ID`),

    KEY 'FK_FLIGHT_AEROPLANE_ID' ('AEROPLANE_ID'),

    KEY 'FK_FLIGHT_DESTINATION_ID' ('DESTINATION_ID'),

    KEY 'FK_FLIGHT_DEPARTINGFROM_ID' ('DEPARTINGFROM_ID'),

    CONSTRAINT `FK_FLIGHT_AEROPLANE_ID` FOREIGN KEY ('AEROPLANE_ID') REFERENCES 'aeroplane' ('ID'),

    CONSTRAINT 'FK_FLIGHT_DEPARTINGFROM_ID' FOREIGN KEY ('DEPARTINGFROM_ID') REFERENCES 'airport' ('ID'),

    CONSTRAINT 'FK_FLIGHT_DESTINATION_ID' FOREIGN KEY ('DESTINATION_ID') REFERENCES 'airport' ('ID')

) ENGINE=InnoDB DEFAULT CHARSET=latin1;

Flight Class Type Entity

DROP TABLE IF EXISTS `flight_booking`.flightclasstype;
CREATE TABLE  `flight_booking`.flightclasstype (   `ID` bigint(20) NOT NULL,
    `CLASSTYPE` varchar(255) default NULL,
    `PERCENTVALUE` double default NULL,

    PRIMARY KEY  (`ID`)

) ENGINE=InnoDB DEFAULT CHARSET=latin1;

Passenger Type Entity

DROP TABLE IF EXISTS `flight_booking`.passengertype;
CREATE TABLE  `flight_booking`.passengertype (   )
`ID` bigint(20) NOT NULL,
`PASSENGERTYPE` varchar(255) default NULL,
`AGERANGE` varchar(255) default NULL,
`PERCENTVALUE` double default NULL,
PRIMARY KEY (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

Payment Card Entity

DROP TABLE IF EXISTS `flight_booking`.`paymentcard`;
CREATE TABLE `flight_booking`.`paymentcard` (
  `ID` bigint(20) NOT NULL,
  `SECURITYNUMBER` int(11) default NULL,
  `BILLINGADDRESSLINE1` varchar(255) default NULL,
  `CARDTYPE` varchar(255) default NULL,
  `BILLINGADDRESSLINE2` varchar(255) default NULL,
  `NAMEONCARD` varchar(255) default NULL,
  `BILLINGADDRESSLINE3` varchar(255) default NULL,
  `EXPIRYDATE` datetime default NULL,
  `POSTCODE` varchar(255) default NULL,
  `CARDNUMBER` varchar(255) default NULL,
  `BILLINGCOUNTRY` varchar(255) default NULL,
  `STARTDATE` datetime default NULL,
  `CUSTOMER_ID` bigint(20) default NULL,
  `CUSTOMERACCOUNT_ID` bigint(20) default NULL,
  PRIMARY KEY (`ID`),
  KEY `FK_PAYMENTCARD_CUSTOMER_ID` (`CUSTOMER_ID`),
  KEY `FK_PAYMENTCARD_CUSTOMERACCOUNT_ID` (`CUSTOMERACCOUNT_ID`),
  CONSTRAINT `FK_PAYMENTCARD_CUSTOMERACCOUNT_ID` FOREIGN KEY (`CUSTOMERACCOUNT_ID`) REFERENCES `customeraccount` (`ID`),
  CONSTRAINT `FK_PAYMENTCARD_CUSTOMER_ID` FOREIGN KEY (`CUSTOMER_ID`) REFERENCES `customer` (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
## Appendix C: Testing

<table>
<thead>
<tr>
<th>Component</th>
<th>Test No.</th>
<th>Procedure</th>
<th>Preconditions</th>
<th>Input</th>
<th>Output</th>
<th>Postconditions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login/Logout</td>
<td>1</td>
<td>Login</td>
<td>User is not logged in. User exists in system</td>
<td>Username='<a href="mailto:hydem06@gmail.com">hydem06@gmail.com</a>', Password='ij'</td>
<td>Directed to the homepage. Shopping cart created. Session created.</td>
<td>User is logged in</td>
<td>Passed</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Logout</td>
<td></td>
<td></td>
<td>Login failed. Form shown again.</td>
<td>User login denied</td>
<td>Passed</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Login</td>
<td>User is not logged in. Items were added to user's shopping cart when logged in last. User exists in system</td>
<td>Username='<a href="mailto:hydem06@gmail.com">hydem06@gmail.com</a>', Password='ij'</td>
<td>Your shopping cart is currently empty.</td>
<td>User is logged in. No items exist in shopping cart</td>
<td>Passed</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Logout</td>
<td>User is logged in. User exists in system</td>
<td>N/A</td>
<td>Directed to the homepage</td>
<td>User is logged out. Shopping cart cleared. Session invalidated</td>
<td>Passed</td>
</tr>
<tr>
<td>Sign-up</td>
<td>5</td>
<td>Sign-up</td>
<td>Customer with same email address does not exist</td>
<td>Default</td>
<td>Validation Error: required fields are missing. Please enter all required data.</td>
<td>User sign-up form is displayed again. User is not signed-up</td>
<td>Passed</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Sign-up</td>
<td>Customer exists with email address='<a href="mailto:hydem06@gmail.com">hydem06@gmail.com</a>'</td>
<td>All valid data. Email address='<a href="mailto:hydem06@gmail.com">hydem06@gmail.com</a>'</td>
<td>Validation Error: your specified email address is already in use. Please enter another.</td>
<td>User sign-up form is displayed again. User is not signed-up</td>
<td>Passed</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Sign-up</td>
<td>Customer with same email address does not exist</td>
<td>All valid data</td>
<td>Homepage displayed welcoming the user by name</td>
<td>Homepage displayed. User registered and logged in</td>
<td>Passed</td>
</tr>
<tr>
<td>Step</td>
<td>Test Scenario</td>
<td>Expected Result</td>
<td>Actual Result</td>
<td>Notes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>---------------</td>
<td>-----------------</td>
<td>---------------</td>
<td>-------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Update customer details</td>
<td>Customer with same email address does not exist</td>
<td>Required fields left blank</td>
<td>Validation Error: required fields are missing. Please enter all required data.</td>
<td>Update customer details page is displayed again</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Update customer details</td>
<td>Customer exists with email address='<a href="mailto:hydem06@gmail.com">hydem06@gmail.com</a>'</td>
<td>All valid data. Email address='<a href="mailto:hydem06@gmail.com">hydem06@gmail.com</a>'</td>
<td>Validation Error: your specified email address is already in use. Please enter another.</td>
<td>Update customer details page is displayed again</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Update customer details</td>
<td>Customer with same email address does not exist</td>
<td>All valid data, but changed</td>
<td>Homepage displayed welcoming the user by name</td>
<td>Homepage displayed. User details updated</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Search flights</td>
<td>Database populated</td>
<td>Default</td>
<td>Validation Error: required fields are missing. Please enter all required data.</td>
<td>Search form shown again</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Search flights</td>
<td>Database populated</td>
<td>Passengers left default</td>
<td>Validation Error: at least one passenger must be selected, and infants must be accompanied by at least one adult. Data displayed in search form as before.</td>
<td>Search form shown again</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Search flights</td>
<td>Database populated</td>
<td>Date left default ('DD/MM/YY')</td>
<td>Validation Error: departure date: 'DD/MM/YY' is in an incorrect format. Please use DD/MM/YY.</td>
<td>Search form shown again</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>Test Case</td>
<td>Action</td>
<td>Status</td>
<td>Reason 1</td>
<td>Reason 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>--------</td>
<td>----------</td>
<td>----------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Search flights</td>
<td>Database populated</td>
<td>Date='01/01/06'</td>
<td>Validation Error: departure date: '01/01/06' is in the past. You must choose a future date.</td>
<td>Search form shown again</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Search flights</td>
<td>Database populated</td>
<td>Date='30/02/08'</td>
<td>Validation Error: departure date: '30/02/08' is in an incorrect format. Please use DD/MM/YY.</td>
<td>Search form shown again</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Search flights</td>
<td>Database populated</td>
<td>Departure Airport='ABZ - Aberdeen', Destination='ABZ - Aberdeen'</td>
<td>Validation Error: you cannot choose the same airport for both departure and destination.</td>
<td>Search form shown again</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Search flights</td>
<td>Database populated</td>
<td>Adults=0, Children=1, Infants=1</td>
<td>Validation Error: at least one passenger must be selected, and infants must be accompanied by at least one adult.</td>
<td>Search form shown again</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>Shopping Cart</td>
<td>Add flight(s) to shopping cart</td>
<td>Flight(s) selected. User not logged in</td>
<td>Flight Number: MA6723</td>
<td>Login page</td>
<td>Login request. Flight(s) selection remembered</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Add flight(s) to shopping cart</td>
<td>Flight(s) selected. User logged in</td>
<td>Flight Number: MA6723</td>
<td>Cart details page with selected flight(s) and correct passenger numbers</td>
<td>Flight(s) added to users cart</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Add flight(s) to shopping cart</td>
<td>Flight(s) selected. User logged in. Flight Number: MA6723 already in cart</td>
<td>Flight Number: MA6723</td>
<td>Cart details page with Flight Number: MA6723 appearing once (no repeated entries in list)</td>
<td>Cart details page remains the same as before</td>
<td>Passed</td>
<td></td>
</tr>
<tr>
<td>Step</td>
<td>Action Description</td>
<td>Scenario Details</td>
<td>Result</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------------</td>
<td>-----------------</td>
<td>--------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Add flight(s) to shopping cart</td>
<td>Flight(s) selected. User logged in. Flight Number: MA6723 already in cart</td>
<td>Flight Number: MA6740</td>
<td>Cart details page with both flights in list</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Add flight(s) to shopping cart</td>
<td>Flight(s) selected. User not logged in</td>
<td>Flight Number: MA6723, MA6724, MA6740</td>
<td>Cart details page with all three flights in list</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Update cart item</td>
<td>Flight(s) selected. Flight(s) in cart. User logged in</td>
<td>Flight Number: MA6725, Adults=2, Children=3, Infants=1</td>
<td>Cart details page refreshed with flight showing updated passenger details</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Update cart item</td>
<td>Flight(s) selected. Flight(s) in cart. User logged in</td>
<td>Flight Number: MA6725, Adults=0, Children=1, Infants=1</td>
<td>Validation Error: at least one passenger must be selected, and infants must be accompanied by at least one adult.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Remove cart item</td>
<td>Flight(s) selected. Flight(s) in cart. User logged in</td>
<td>Flight Number: MA6722 (first flight in list)</td>
<td>Cart details page, excluding previous first flight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Remove cart item</td>
<td>Flight(s) selected. Flight(s) in cart. User logged in</td>
<td>Flight Number: MA6725 (middle flight in list)</td>
<td>Cart details page, excluding previous middle flight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Remove cart item</td>
<td>Flight(s) selected. Flight(s) in cart. User logged in</td>
<td>Flight Number: MA6726 (last flight in list)</td>
<td>Cart details page, excluding previous last flight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Remove cart item</td>
<td>Flight(s) selected. Flight(s) in cart. User logged in</td>
<td>Flight Number: MA6722 (only flight in list)</td>
<td>Your shopping cart is currently empty.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Book contents of cart</td>
<td>Flight(s) in cart. User logged in. Customer has no payment cards</td>
<td>N/A</td>
<td>No payment cards were found.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Passed
<table>
<thead>
<tr>
<th>Step</th>
<th>Task</th>
<th>Description</th>
<th>Expected Result</th>
<th>Actual Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>Book contents of cart</td>
<td>Flight(s) in cart. User logged in. Customer has payment card(s)</td>
<td>Choose a payment card page displayed with all payment cards associated with the user</td>
<td>Passed</td>
</tr>
<tr>
<td>31</td>
<td>Choose a payment card</td>
<td>Flight(s) in cart. User logged in. Payment card selected</td>
<td>Payment card with ID=3</td>
<td>Booking details displayed - flight details and payment card details</td>
</tr>
<tr>
<td>32</td>
<td>Choose a payment card</td>
<td>Flight(s) in cart. User logged in. 'Pay Later' option chosen</td>
<td>Payment card selected</td>
<td>Booking details displayed - flight details but no payment card details</td>
</tr>
<tr>
<td>35</td>
<td>Confirm booking</td>
<td>Customer choose 'Pay For Later' option. User logged in</td>
<td>Confirm booking page</td>
<td>Passed</td>
</tr>
<tr>
<td>39</td>
<td>Display booking list</td>
<td>No previous bookings. User logged in</td>
<td>Display booking list page</td>
<td>Passed</td>
</tr>
<tr>
<td>40</td>
<td>Display booking list</td>
<td>Previous bookings include both unpaid and paid. User logged in</td>
<td>Two bookings with statuses of 'Payment Delayed' and 'Completed'</td>
<td>Passed</td>
</tr>
<tr>
<td>41</td>
<td>Update a booking</td>
<td>Previous booking selected. User logged in</td>
<td>Update booking attempted with valid details</td>
<td>Booking updated successfully</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>42</td>
<td>Cancel a flight in the booking</td>
<td>Previous booking selected contains more than one flight. User logged in</td>
<td>Cancel flight attempted</td>
<td>Flight cancelled successfully</td>
</tr>
<tr>
<td>43</td>
<td>Cancel a flight in the booking</td>
<td>Previous booking selected contains one flight. User logged in</td>
<td>Cancel flight attempted</td>
<td>Cancellation Error: a booking must have at least one flight. You may try to cancel the booking.</td>
</tr>
<tr>
<td>44</td>
<td>Cancel booking</td>
<td>Two bookings in list. User logged in</td>
<td>Cancel booking attempted</td>
<td>Booking cancelled successfully</td>
</tr>
<tr>
<td>45</td>
<td>Cancel booking</td>
<td>One booking in list. User logged in</td>
<td>Cancel booking attempted</td>
<td>Booking cancelled successfully</td>
</tr>
<tr>
<td>47</td>
<td>Pay for booking</td>
<td>Customer has an unpaid booking</td>
<td>Pay for booking attempted by choosing a payment card</td>
<td>Booking listed as complete</td>
</tr>
<tr>
<td>61</td>
<td>Add payment card</td>
<td>User is logged in</td>
<td>Required fields left blank</td>
<td>Validation Error: required fields are missing. Please enter all required data.</td>
</tr>
<tr>
<td>62</td>
<td>Add payment card</td>
<td>User is logged in</td>
<td>Valid data except security number='90.2'</td>
<td>Validation Error: security number: '90.2' is in the incorrect format. You must enter a number.</td>
</tr>
<tr>
<td></td>
<td>Add payment card</td>
<td>User is logged in</td>
<td>Valid data</td>
<td>Payment card list displayed with newly added payment card details</td>
</tr>
<tr>
<td>---</td>
<td>------------------</td>
<td>-------------------</td>
<td>------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>64</td>
<td>Update payment card</td>
<td>User has at least one payment card. User is logged in</td>
<td>Required fields left blank</td>
<td>Validation Error: required fields are missing. Please enter all required data.</td>
</tr>
<tr>
<td>65</td>
<td>Update payment card</td>
<td>User has at least one payment card. User is logged in</td>
<td>Valid data except security number='90e'</td>
<td>Validation Error: security number: '90.e' is in the incorrect format. You must enter a number.</td>
</tr>
<tr>
<td>66</td>
<td>Update payment card</td>
<td>User has at least one payment card. User is logged in</td>
<td>Valid data</td>
<td>Payment card list displayed with updated payment card details</td>
</tr>
<tr>
<td>67</td>
<td>Remove payment card</td>
<td>User has at least one payment card. User is logged in</td>
<td>N/A</td>
<td>Payment card list displayed without removed payment card details</td>
</tr>
</tbody>
</table>