02267 Software development of web services

Fall 2015 - DTU

Group 1

s150939 Ali Chegini
s123230 Daniel Gert Brand
s142290 Daniel Sanz Ausin
s151409 Quentin Tresontani
s040551 Raj Kumar Kalvala
s150962 Rustam Ali Hussaini

Summary

1) Introduction
    a) Introduction to Web services

2) Coordination Protocol

3) Web service implementations
    a) Section on the data structures used
    b) Section for the airline- and hotel reservation services
    c) Section for the BPEL implementation
    d) Section for the RESTful implementation

4) Web service discovery

5) Comparison RESTful and SOAP/BPEL Web Services

6) Advanced Web service technology

7) Conclusion

8) Who did what

Introduction

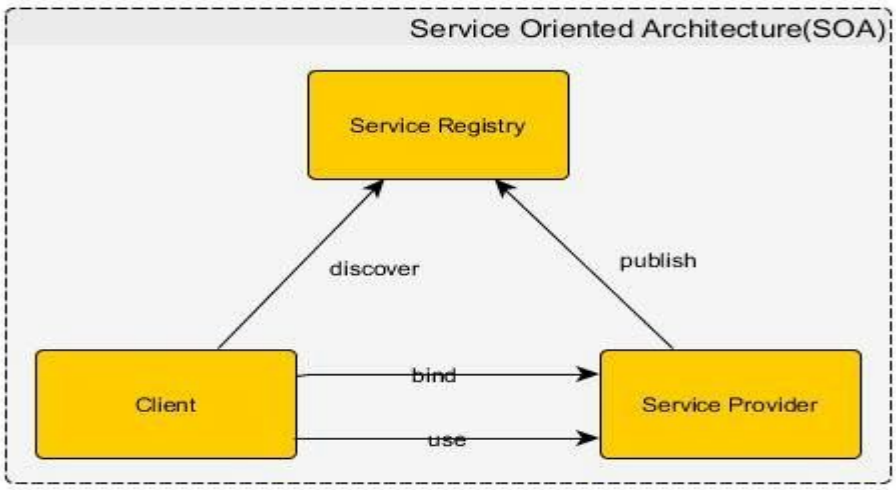
This report is the presentation of work done in developing a web services for a travel agency called TravelGood. TravelGood works together with the web services of airline reservation agency LameDuck and hotel reservation agency NiceView. The travel agency allows customer to book the flights and hotels for the trip. The customer can plan, book, cancel flights/hotels. A database of 6 hotels and 5 flights has been considered for the implementation.
This report presents introduction to Web Services, description of Coordination Protocol, implementation of Web Services for the airline reservation and Hotel reservation services using SOAP/BPEL and REST versions of the travel agency followed by description of bank service.
In the later sections, BPEL and RESTful testing is done for planning and booking, cancel booking and booking fails. In the web service description section, 3 WSIL files are created, one each for TravelGood, LameDuck and NiceView each of them linked to other two. Later Comparison of RESTful and SOAP/BPEL Web services is presented and then there is discussion of Advanced Web service Technology and Conclusion.

Introduction to web services

A web service is a framework for communication between two applications over a network or internet mainly for data exchange. The client sends data request and the server provides the required information. The two applications communicating may not be necessarily built on same platform or developed using same programming language.

A web service is implementation of Service oriented architecture (SOA). In SOA, two software applications communicate with each other where one is service provider and the other service consumer. The service provider provides the service description and places it in a public directory called UDDI - Universal Description, Discovery and Integration. UDDI is a method of publishing web services and make it available for others. This service description is written in WSDL(Web Server Description Language). The consumer sends a request to the provider to check what services are available and the provider responds after looking up in the directory after which the consumer communicates with the directory to get the information. The communication between service provider and consumer is in XML messages whereas the communication between the applications and directory is made using SOAP(Simple Object Access Protocol). SOAP is a standard which describes how to encode a message in XML in order to be able to invoke functions in other applications.

SOA is an architecture where there are a number of services that are communicating with each other over a network. The purpose of communication is usually data exchange or it will be coordinating an activity. Here a service means a self contained unit of functionality that can implement one or more actions. In SOA, services communicate with each other using a specified protocol that can describe how the messages should be passed and an architecture of SOA has been shown below.



Source: [3].

SOA is designed including some specific security requirements like authentication, authorization, reliable messaging and consumer specification.

Advantages of web services:

Web services have the following advantages:

- Web services use XML data representation which makes the communication among the applications easier and platform independent.
- Web services adopt loosely coupled architecture therefore when the web server interface changes over the time it does not affect the client's ability to communicate. A tightly coupled system is where the client and server logic are closely tied to each other and therefore if one interface changes, the other should also be updated.
- Web services support both synchronous and asynchronous implementations.

WSIL stands for Web Service Inspection Language. It is used for service discovery to replace UDDI. When the service is discovered using UDDI, it takes to centralised registry whereas WSIL takes directly to the service provider.

Web services can be separated in two kinds. On the on hand are services based on REST architecture. They are called RESTful service. REST stands for REpresentational State Transfer. This architecture uses HTTP methods and is stateless which means client context is not stored on the server, the client caches the response and can use it for the further requests. HTTP methods used in this architecture and their characteristics are :

- GET - used to read data. This method is safe- there are no side effects while performing. It is idempotent - can be performed multiple times without changing the result beyond initial application
- PUT - used to create new resource. This is Idempotent- does not change the result when performed multiple times.

- DELETE - used to remove a resource, Idempotent - once the resource is deleted, nothing is left to delete if this method is performed again.
- POST - used to update a resource.

On the other hand are the BPEL based services. BPEL stands for Business Process Execution Language. It is an XML based programming language which is used to describe the business process. It arranges the tasks and activities exposed as web services in an order. BPEL allows us to use a Web Service and write a Web Service.
Web Services coordination uses entry points of WSDL files which describe actual web services to be used, to invoke their operations. WSDL files contains information regarding partner links for each web service. These partner links can be used by BPEL process to explain the communication relationship BPEL and the associated parties.

BPEL is an XML programming language and has 3 main components:
- Programming Logic
- Data Types
- Input/Output

Data Types that are used in the program are defined using XML Schema Definition(XSD). The Input/Output operations are performed by the Web Service and defined using WSDL. The programming logic that puts everything for the tasks to be performed in an order is defined using BPEL.

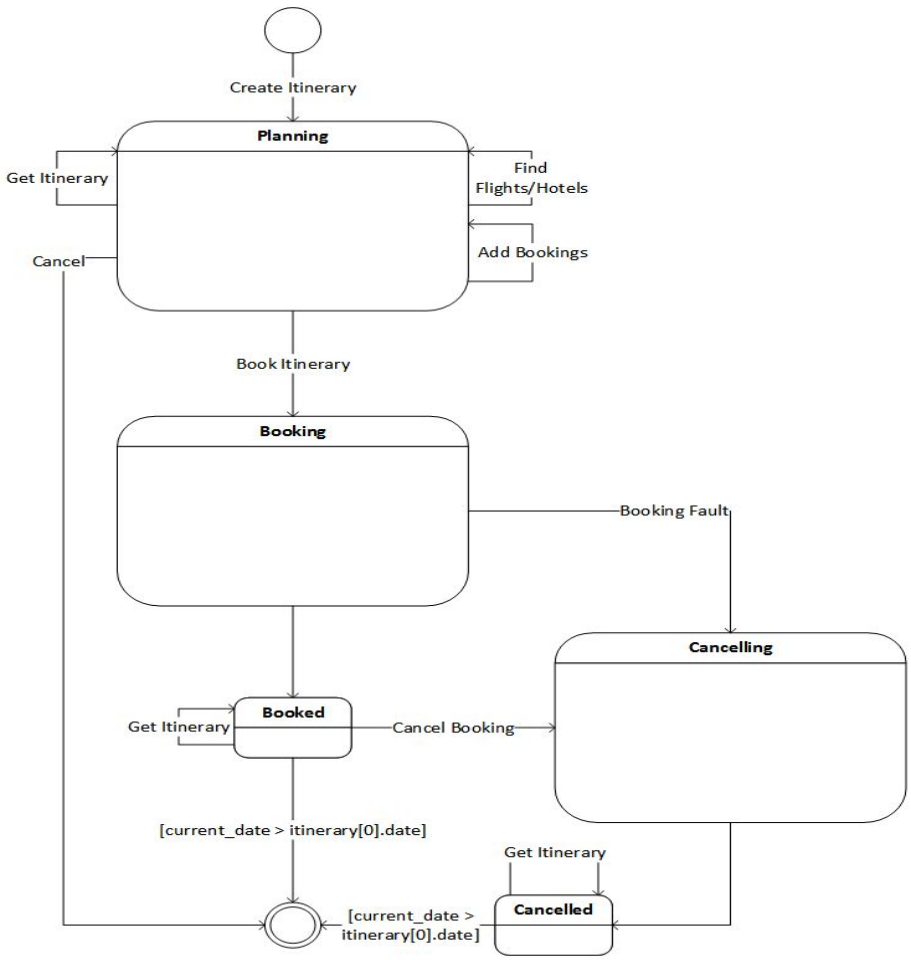
Advanced Web Services Technologies:

WS-Addressing provides information about who is using which information and WS -Reliable Messaging makes sure that the messages arrive and they arrive in correct order. WS-Security Provides ways of maintaining integrity and confidentiality of SOAP messages over a network and WS-Policy Provides specifications for secure transfer.

2. Coordination Protocol

This Protocol is used to derive the skeleton processes later these are extended to the real composed web services. The advantage of this protocol is the process of Composite is obeys to Coordinate Protocol[3].

Coordination Protocol Diagram:



Description:

The Coordination Protocol shown in the figure illustrates the interaction between the Customer and Travel Good. In the Protocol description, Sates notation is in the bold letters and the actions are in normal font. When the process begins, customer who wants to book flights and/or hotels starts the initiation and the process starts with ***"Planning" sate***. The customer provides the required data and the travel good goes and looks for the availability of flights/hotels on specified dates. The customer may cancel the plan at this stage and if so, the

process terminates. If the flights/hotels are available, then the Itinerary message is sent to the customer. If the flights and/or hotels are available, it goes to the ***"Booking" state*** If the booking fails, all the previous itineraries should be cancelled and the travel good goes to ***"Cancelling"* state** and the process terminates or the customer can start new plan.

The booking can be cancelled by the customer at any time before the travel starts and if the booking is cancelled the state changes to ***"Cancelling"*** *state*. When the travel is finished, the customer goes to ***" Process End"*** *state* and the process terminates.

3. Web service implementations

a. Data structures used for the web services

Our web services are based on different types we created. These types can be separated in two different categories. On the first hand are the types related to the data we need to store during the different processes, they are so linked to our "Database.java" files. These files allows us to store information without having to implement a real database. On the other hands are the files that are used as input and output for the operations created in the web services.

In this part, I will first describe the different types related to the SOAP web services NiceView and LameDuck, then to the BPEL process and the composite application web service and finally ending with the REST project.

SOAP based web services NiceView and LameDuck

Construction of a java based simple data base

NiceView and LameDuck are using types that are really close to each other. First of all, we are using java databases that are working the same way.

NiceView database types

- private static List<HotelType> hotelDataBase;
- private static HashMap<HotelType,String> bookingDB;

LameDuck database types

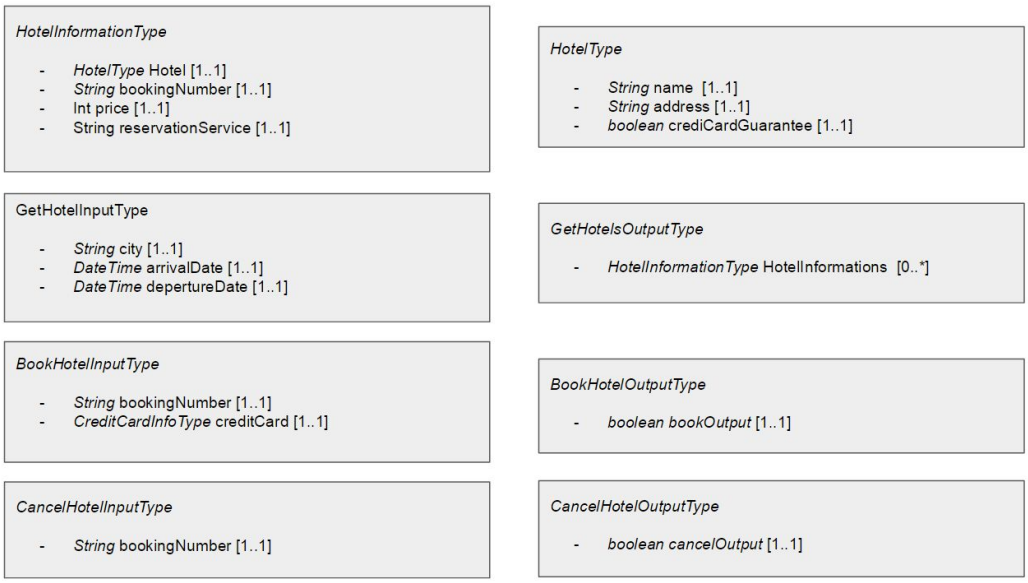
- private static List<FlightType> flightDataBase;
- private static HashMap<FlightType, String> bookingDB;

The two lists *hotelDataBase* and *flightDataBase* represents the list of flights and hotels that the web services are proposing to its clients. *bookingDB* represents each time a hashmap where flights or hotels are associated to a unique booking number. They are all initialized when the web services project is built. *HotelType* and *FlightType* are defined in the wsdl of the related web service.

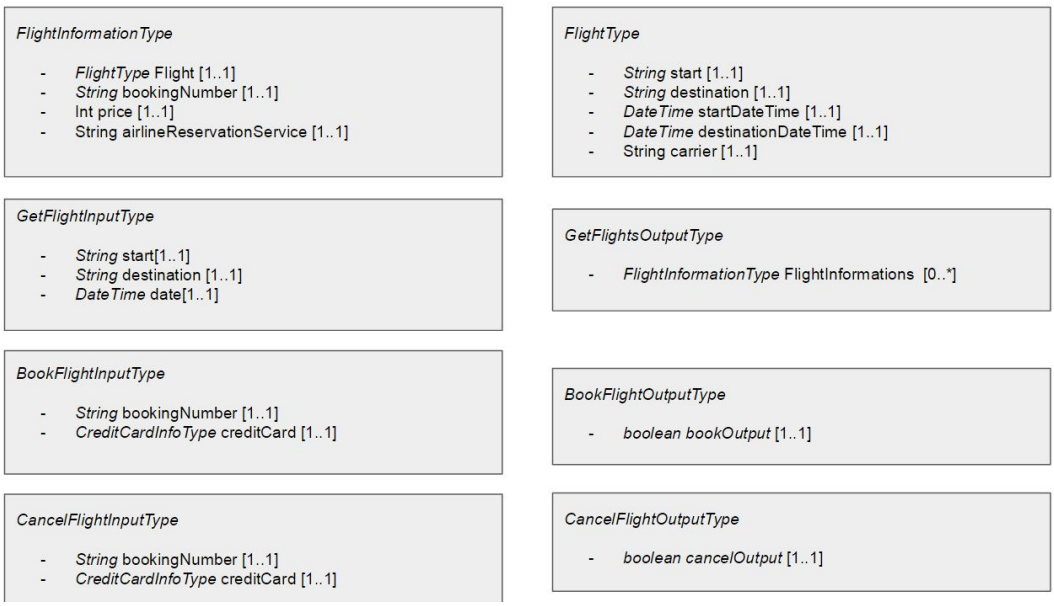
While creating these database, we made a few assumptions that need to be explained. Firstly we choose to tell the client that hotels and flights in our lists are always available. We choose then use one unique booking number per hotel and per flight. As a consequence, if a client is asking to book two times the hotel on different dates, we are so returning the same booking number. In real life we would have to create different ones and register them in the database but, here, since we are not checking the availability of the hotel, we don't need to store so much detailed information. We also choose to charge client with the same prices, in order to avoid having a database of prices for flights and hotels.

Complex types used as input and output for web services operations

Here is the representation of the data model used for NiceView webservice. These complex types have been defined while creating the wsdl file of the web service. They can be seen there in the form of a xsd schema.



Here is the representation of the data model used for LameDuck:



We can see that the data models are almost the sames and are based on the problem description of the project.

In the wsdl schema of LameDuck and NiceView we imported the bank service xsd schema in order to use directly the provided types for our complex types. *CreditCardInformationType*

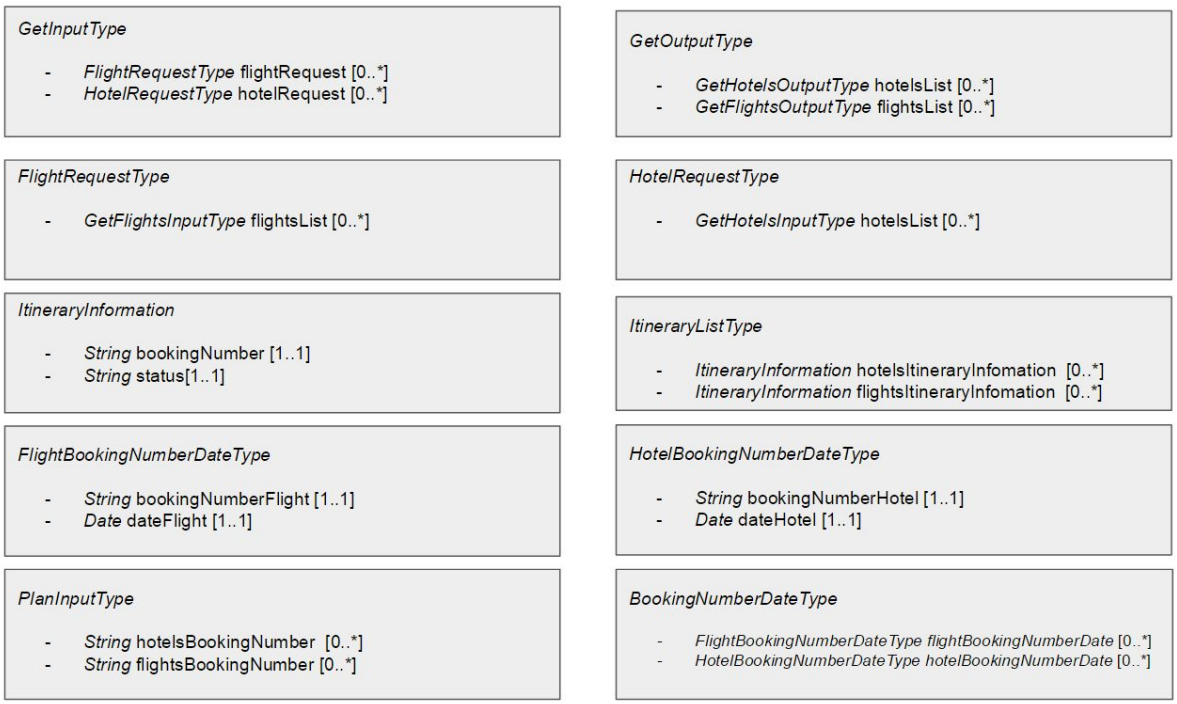
is so the one from the bank service. We choose this solution in order to avoid creating the same complex type structure and having to cast the two types before calling the bank services. In the context of this project, it made sense for us because we are creating all the web service or they are designed for us. However, in a real life project, it would probably be better to have our own types in the wsdl and cast them in the java part to make them fit with the bank ones. It would make the web services less coupled together. The client is also not supposed to know thanks to our wsdl which bank we are contacting since now he can find the information while checking the namespace of the credit card type.

While importing the xsd file of the bank, we choose to do it in a static way for more reliability. We created a copy of the xsd file inside the project in order to avoid having an import referring to an online url in our wsdl file. This way we are avoiding losing the namespace while exchanging messages with the client. This problem may occur with dynamic import links.

BPEL based TravelGood web service

In the BPEL part, we didn't need to create any database because BPEL already handles to store information that are linked to the BPEL process instance. We created so only a few global variables and most of them were used as predicates. We didn't need to store any input information because BPEL gives you access to the content of former messages that have been exchanged. For instance, while canceling flights, we can get the credit card information from the book input message. This function is closely linked to the use we made of correlation set in order to be able to handle multiple itinerary.

The complex types created for the exchanged messages are the following ones:



Once again, as we did with NiceView, LameDuck and the bank services, we are directly using the complex types from the namespaces of NiceView and LameDuck, we are not creating our own ones.
To achieve this result, we created the xsd schemas of NiceView and LameDuck and added them into the TravelGood project. However, we were getting conflicts. In fact, we were indirectly adding multiple times the bank service while importing each one of the 2 xsd schema, but also while importing the xsd schema of the bank in TravelGood wsdl in order to get the credit card type for the booking input.
We finally solved this problem by creating a new xsd schema that contains both NiceView and LameDuck types and that include - not import - the bank services types in its namespace. This way, we only have to import the new created xsd file in order to access all the data types without conflict.
We are aware that our solution is not the proper one because changes on NiceView, LameDuck or the bank side, would imply a lot of refactoring for us. We would have to change this xsd file, then the wsdl and the BPEL. This could have been better if NiceView and LameDuck had their own data types for the bank as explained before. This way we should have been able to import separate xsd schema without referring many times to the bank and avoid conflicts. However, we noticed the improvement solution at a point that would have apply to much refactoring in the project.

REST based TravelGood web service

Construction of a java based simple data base

Since TravelGood should be able to handle many itineraries, we had to implement a kind of database in order to remember information that are linked to an itinerary.

TravelGood REST database

```
- private static final HashMap<Integer, Itinerary> plannedItineraries = new HashMap<>();
- private static final HashMap<Integer, Itinerary> bookedItineraries = new HashMap<>();
- private static final HashMap<Integer, CreditCardInfoType> earliest Date = new HashMap<>()
- private static final HashMap<Integer, CreditCardInfoType> bookingCreditCard = new HashMap<>();
- private static final HashMap<String, XMLGregorianCalendar> hotelsDates = new HashMap<>();
- private static final HashMap<String, XMLGregorianCalendar> flightsDates = new HashMap<>();
```

The database is composed of hashmaps using the itinerary ID as a key. This database allows us to store the different itineraries but also the user credit card information, in order to avoid asking him his information again when we are canceling, and the reservation dates for each researched item, in order to remember the earliest date that is a later one saved when the booking has started.

Representations used as input and output for web services operations

While implementing the REST web service of Travel Good, we created a project named *SharedRepresentation*. This projects contains all the representations that the client and the server are both using in their requests and responses.

Here is a short overview of the different input types we are using:

All these types, except Itinerary are inherited from the abstract Representation. The *SearchOutputRepresentation* types is not detailed here, but the subclasses he is composed of are matching exactly the same structure as the one described on BPEL's part. We focused on having similar data types for each implementation of Travel Good.

b. Section on the airline and hotel reservation

LameDuck Airline reservation and Nice View hotel reservation Background

LamdeDuck an airline Reservation agency and Nice View a hotel reservation agency both provide their services as traditional SOAP based web service to the TravelGood travel agency.

Proposed Solution

To implement the SOAP based web service for these 2 agencies we used top down approach where we created the WSLD first then the java code implementation.

our design choice was to use document/literal type binding style for translating the Nice View and LameDuck wsdl to a soap message. because with the document/literal style model we can construct the soap body any way want.

The services operation and implementation of the operations are as following.

Airline Reservation operations and implementation

The LameDuck Airline reservation services offer three operations *getFlights*, *bookFlights* and *cancelFlights*.

Where *getFlights* operation takes start, destination and data as an arguments and returns a list of flights information, where each flight information consist of a booking number, price, airline reservation name, time arrival time and carrier operating the flight.

bookFlight operation takes a booking number and credit card information and books the flight after receiving the payment using the *chargeCreditCard* of the bank. If the booking was successful the bookFlight returns true otherwise fault exception will be thrown.

CancelFlight operation takes a booking number of booked flight, price, and *creditCard* information and cancels the flight by refunding 50% of the price of the flight using the CundCreditCard operation of the bank Service. If there a fault in the cancellation of the flight the *canelFlight* operation throws a fault exception.

Get Flights Implementation

In the get flight method, we are going through the database that have been initialized in order to find match that are matching the input information. If the request is matching the start place and destination place, the flights information are added to the response that is sent to the user.

Book Flights implementation

In the booking flight first we check if the flight input is not null. if it is null it throws empty input exception. other wise it will initialize the flight list and check if the flight is booked. if the flight is booked it get the credit card information and tries to charge the credit card, and if there is an error while charging the credit card the *bookFlightFault* method will be invoked.

Cancel Flights Implementation

The *cancelFlight* check if the cancel flight input is not null if it is null it throws a cancel flight exception otherwise it will initiate an account cancellation and will refund the user by 50% of the price value. We assumed that all flights had the same price in order to ease the implementation process. If the cancellation is successful a boolean *true* is returned. If the cancellation is not successful an exception of *cancelFlightFault* will be thrown.

Unit Test implementation

We have implemented Unit test class called LameDuckClientTest and in this class we have implemented some test method to test if the flight list works , and also test the booking for error if the the booking is empty and test the cancellation of the flight.

This way, we can test all the possible cases, when the called operation should fail, sh
in the test we used assertEqual instead of system.out.print to make sure the web server produce the correct results.
the tests are working fine and all the test are passed 100%.

Hotel reservation operations and implementation

The Nice View Hotel reservation services offer three operations getHotel, *bookHotels* and *cancelHotels*.

Where the *getHotels* operations takes the departure date, arrival date and a city as parameter and returns a list of hotels with the information containing the hotel name, address, price, booking number and name of the hotel reservation service , whether a credit card guarantee is required or not.

bookHotel operation takes a booking number and credit card information and books the hotel. If a guarantee is required the bank service *validateCreditCard* operation is called.
If the credit card information is validated and booking was successful the *bookHotel* will return true otherwise it throws a fault exception.

cancelHotel operation takes a booking number of a hotel reservation and cancels the hotel reservation if the cancellation fails it throws an exceptions.

Nice View Implementation

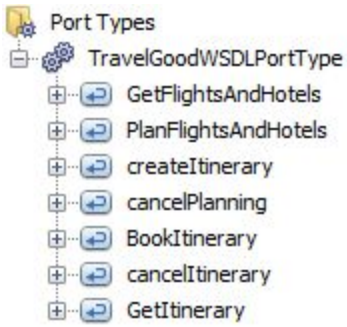
As there is very close similarity in the implementation of Nice View Hotel reservation and LameDuck Flight booking. we decided to not repeat our self and just mention the differences in the implementation.
The only difference we see is in the cancellation of the flight and hotel is that in case of cancelling a flight the 50% booking will be returned but in case of cancelling a hotel there is no cancellation charge because Nice View does not charge the client for cancellation.

c. Section for the BPEL implementation

BPEL(Business Process Execution Language)

Port type and operations

We started implementing BPEL process by creating a SOA project from the scratch. After making BPEL module we continued by creating a WSDL document for services that travel good offers. As we are dealing with complex types, we used abstract WSDL document.
These are the port types in addition to ports we use from LameDuck and NiceView:



In total we have seven operations defined, including createItinirery, cancelItinirary, getFlightsAndHotels, PlanFlightsAndHotels, BookItinerary, GetItinerary and cancelPlaning. All the operations are Request-Respond so they all have input and output message, in a way that the endpoint receives a message, and sends a correlated message.

Binding style

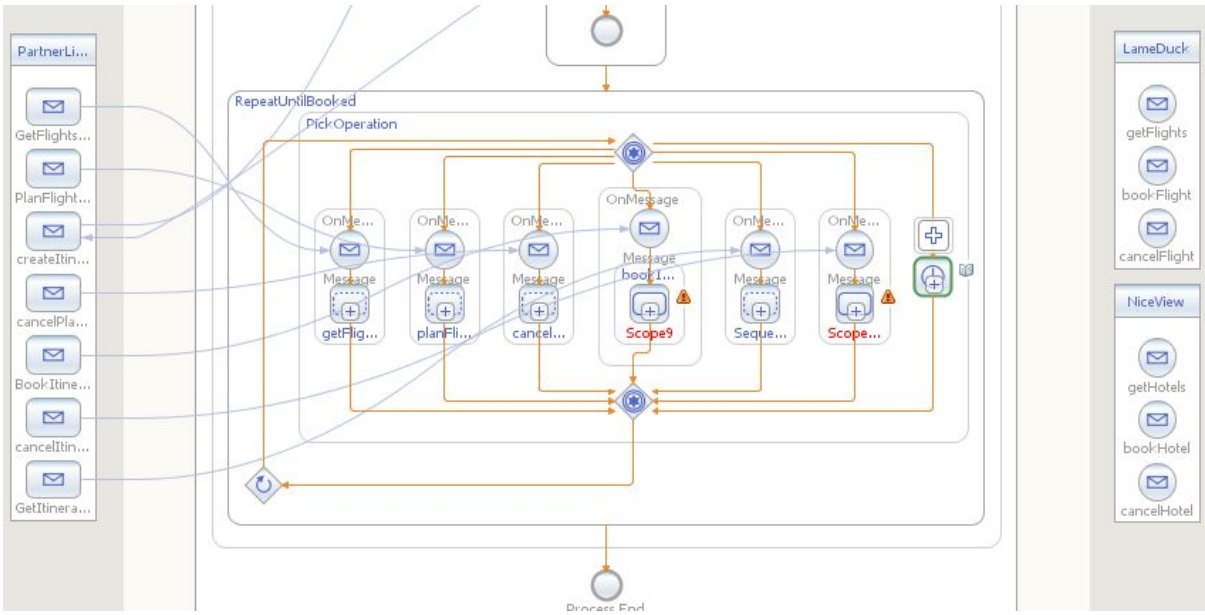
We have to explain the decisions we made regarding to binding style. At first we used document style but later on when we were defining the correlation set, we realized document binding style is not the right choice because in order to define the properties and properties alias we had to access itinerary ID as a part of messages in our operations, but document style binding can only have one part message.
Facing this issue gave us a better understanding about binding styles and how to use them, till then we used to think if we have complex types we are limited to use document style but this experience taught us RPC literal binding can also be used for complex types. Based on this experience we changed the binding style to RPC literal binding which allows more than one part in messages.

BPEL logic and implementation

Implementation got started by importing WSDL file that we had created for TravelGood, LameDuck and NiceView. LameDuck and Niceview are external web services which we will invoke any time we need their services so they will be placed on the right side. in the section DataTypes we have already explained all about the complex types and their structure.

Entire process starts when client sends out a request to create an itinerary, we generate a unique ID by using a function called Get BP ID.

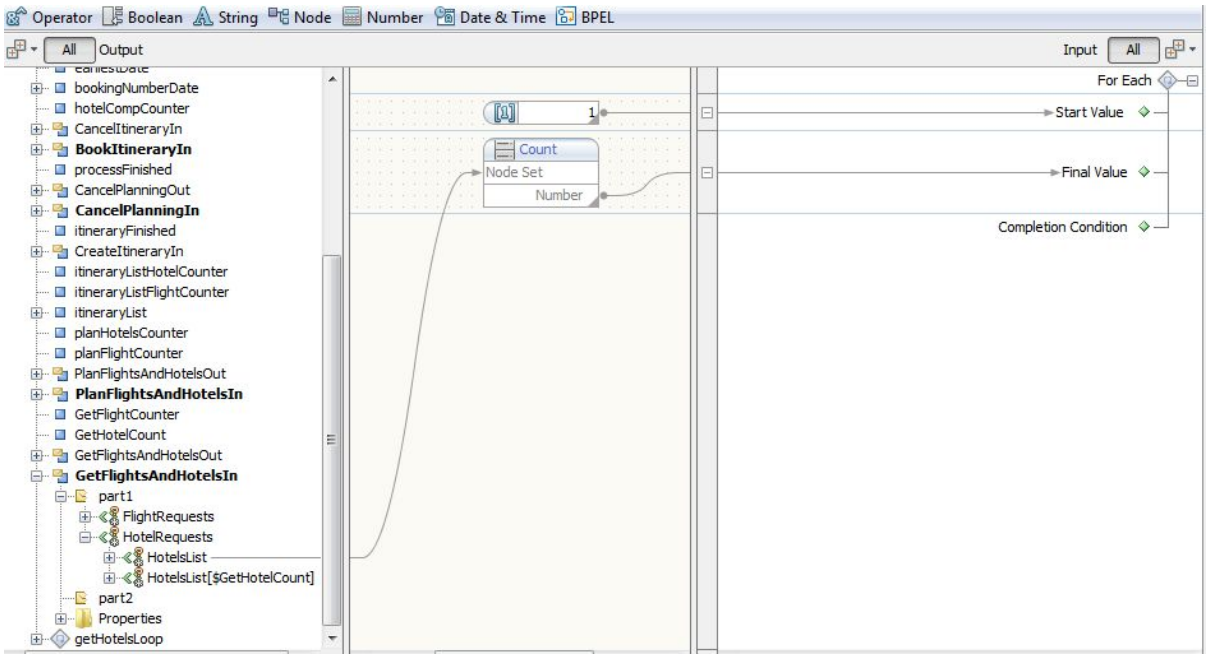
The rest of the process is quite understandable because we tried to use proper naming for our variables, so we are not going through the details and mention what we did in each assign and invoke but from now on we will explain the logic and how BPEL works, briefly.



Main loop has been implemented with a RepeatedUntil activity which is been controlled by a Boolean variable called processFinished. In createItinirery scope we set the value to false and

by setting it to true process will get terminated when needed. All the other 6 operations we have, are connected with each other through a pick activity with message events correlated with our operations. Pick event is mainly used because it can have the process wait until particular set of events is triggered. Each of the operations is being handled with onMessage.

In almost all of the operations, we use one loop for hotels and another loop for flights. In order to keep track of everything and making sure we are putting the right elements in the right place in the lists we used counters and predicates. For example before entering each loop first we check if the list for that loop is empty or not, if it's empty we set the counter to 1 so loop will add elements from beginning of the list, but if list is not empty we check the size of a list with a Node called Count, then we add 1 to it so the loop will start putting elements at the first next empty place. Same logic has been used in all cases we deals with lists in order to make sure we don't overwrite anything and we don't use the wrong elements from the list. Here is an illustrations how we iterate through the lists we want by using a For loop.



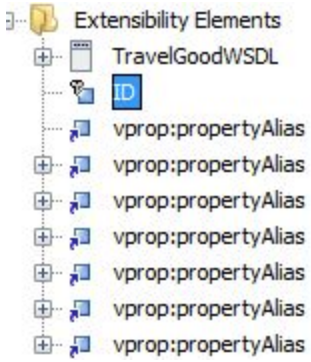
Subsequent operation after create itinerary in line, is *GetFlightsAndHotels*. This operation takes the required information such as arrivals and departures place, dates and etc. then returns a list of available flights and hotels for that request. Next operation is PlanFlightsAndHotels which let the user add and subtract flights and hotels to his plan. User has option to cancel whatever he/she wants by *cancelPlanning* operation.

Further operation is BookItinirary, if user decided to book anything he/she have to provide his credit card information. If everything goes according to plan and booking operation finishes successfully user will be able to *getItinerary* list and check what he has booked. If anything goes wrong during booking procedure all the previous booking gets cancelled and process will be terminated.

We performed this task by using compensation handlers in the booking scope both for hotel and flights. Then in the *bookItinerary* scope we added a fault handlers to catch the fault that may have happened during booking process, inside that fault handler we put a compensate activity. The reason we chose compensate activity is that it only calls the scope that has executed successfully, it means if user had booked one hotel but failed to book a flight, compensate will go back to hotel loop and cancel the hotel automatically.

If itinerary get booked successfully user would be able to cancel it by *cancelItinirery* operation only before the first booking time. We used an alarm to check date in daily period, a global variable called *earliestDate* to store the closest flights/hotels booked date, if conditions to compare current date with *earliestDate*.

In order to add correlation set we defined property called ID and then we defined 7 property alias for all the operations we have to specify where this ID can be found and correlated, after that we defined a correlation set in global scope using those properties. We initiate it while *createItinirary* operation is replying so all the other operations would be able to access it . We used this correlation set in all the other operations in order to make sure we are processing the right request and sending out the right response.



In entire implementation we used enormous number of different activities, including about 16 ForLoops, 15 Ifs, one RepeatedUntil for main loop, many assign/invoke activities and lots of different operands and variables.

Problems and issues during implementation

During the implementation we faced lots of error and issues which took spectacular time to debug and fix. In order to test different parts of the BPEL project, we have implemented 27 different test cases, considering all the scenarios that can happen, targeting different parts of BPEL project. Since enormous number of issues and bugs cannot be discussed in this paper we will only mention one of them which we had found interesting.

> We wanted to use flow construct to use parallelism for hotels and flights loops while invoking LameDuck and NiceView in all operations, but when we took that approach we realized it can't be done. We figured the fact that both of them were using one port

concurrently could be the cause of that problem. Hence we changed the design and avoided parallelism.

d. Section for the RESTful implementation

RESTful(Representational State Transfer)

Definition of Resources and Operations

In the planning phase of the development of the TravelGood REST service, we started defining the resources we will use, the methods they will support and the operations they perform.
In total, we defined three main resources, each with a number of operations and sub-resources.
We have defined the following main resources and operations:

- Itinerary
  - Create Itinerary
  - Find Planned Itinerary
  - Add To Itinerary
  - Cancel Planned Itinerary
- Search
  - Search
- Booking
  - Find Booked Itinerary
  - Book Itinerary
  - Cancel Booked Itinerary

The Itinerary resource relates to building the travel itinerary and all the operations that can be performed on the itinerary before it is booked.
The Search resource is used to search for flight and hotels. It is completely separate from the other two resources as it does not require an itinerary ID or any other information related to the planning and booking operations.
The Booking resources handles the process of booking and any operation to be performed on booked itineraries.

Itineraries are stored in two different hashmaps on the server, depending on whether the itinerary has been booked or not.
The Itinerary resource thus operates on the database containing only planned itineraries and the Booking resource operates on the database containing booked itineraries. When an itinerary is booked, it is moved from the planned to the booked database.

Following is a comprehensive list of all resources and operations defined by the service, along with a description and motivation for each choice.

Resource: /itinerary
Methods:
GET:

- **Name**
  - Create Itinerary
- **Output**
  - ItineraryOutputRepresentation
- **Links**
  - Add To Itinerary
  - Find Planned Itinerary
  - Cancel Planned Itinerary

Description

The main Itinerary resources contains only a single operation, using the GET method. The Create Itinerary operation is the first one that should be invoked by the client when attempting to plan a new itinerary, so that an ID can be obtained for all further itinerary operations.
The operation links to the three other operations under the Itinerary resource for manipulating this specific itinerary.

Resource: /itinerary/{ID}
Methods:
GET:

- **Name**
  - Find Planned Itinerary
- **Output**
  - ItineraryOutputRepresentation
- **Error Messages**
  - Itinerary not found : HTTP 404 NOT FOUND
  - Malformed Input: HTTP 400 BAD REQUEST
- **Links**
  - Add To Itinerary
  - Cancel Planned Itinerary
  - Book Itinerary *(If itinerary contains bookings)*

POST:

- **Name**
  - Add To Itinerary
- **Input**
  - AddToItineraryInputRepresentation
- **Output**
  - ItineraryOutputRepresentation

- **Error Messages**
  - Itinerary not found : HTTP 404 NOT FOUND
  - Malformed Input: HTTP 400 BAD REQUEST
- **Links**
  - Find Planned Itinerary
  - Add To Itinerary
  - Cancel Planned Itinerary
  - Book Itinerary

Description

Adding the ID as a parameter to the Itinerary resource opens up operations on a specific itinerary. This sub-resource supports the GET and POST method for finding an itinerary that is currently being build and adding booking numbers to an itinerary, respectively.
The ID parameter must be a number, otherwise any invoked operation will return a HTTP 400 BAD REQUEST status.
Likewise, for each operation under this resource, if an itinerary in planning with the given ID is not found, the operation will return a HTTP 404 NOT FOUND status.
The Find Planned Itinerary operation searches the database for an itinerary with the provided ID and returns it to the client.
The Add To Itinerary operation adds a list of flight and hotel booking numbers, obtained from the Search resource, to the itinerary with the given ID. The itinerary must, of course, exists, otherwise an HTTP error status is returned.

Resource: /itinerary/{ID}/cancel
Methods:
GET:

- **Name**
  - Cancel Planned Itinerary
- **Output**
  - ItineraryOutputRepresentation
- **Error Messages**
  - Itinerary not found : HTTP 404 NOT FOUND
  - Malformed Input: HTTP 400 BAD REQUEST

Description

Further adding cancel to the URI and invoking with the GET method will permanently delete the found itinerary from the database.

Resource: /search
Methods:
POST:

- **Name**
  - Search
- **Input**
  - SearchInputRepresentation
- **Output**
  - SearchOutputRepresentation
- **Error Messages**
  - Malformed Input: HTTP 400 BAD REQUEST

Description

The Search resource handles all searches for hotels and flights. Due to the similarity of the input for searching for hotels and flights in LameDuck and NiceView, both hotels and flights are acquired with the given input at the same time.

There are no links to or from this resource as it is intended to be a standalone, static operation which can be accessed at any time and thus falls outside the general business process of planning and booking an itinerary.

Resource: /booking/{ID}
Methods:
GET:

- **Name**
  - Find Booked Itinerary
- **Output**
  - ItineraryOutputRepresentation
- **Error Messages**
  - Itinerary not found : HTTP 404 NOT FOUND
  - Malformed Input: HTTP 400 BAD REQUEST
- **Links**
  - Cancel Booked Itinerary

POST:

- **Name**
  - Book Itinerary
- **Input**
  - BookItineraryInputRepresentation
- **Output**
  - ItineraryOutputRepresentation
- **Error Messages**
  - Itinerary not found : HTTP 404 NOT FOUND

    - Malformed Input: HTTP 400 BAD REQUEST
- **Links**
    - Cancel Booked Itinerary
    - Find Booked Itinerary

Description

The Booking resource is very similar to the Itinerary resource, except that its operations relates to booked itineraries. As such, the Booking resource throws the same status errors under the same circumstances as the Itinerary resource.
The Book Itinerary operation is the only operation that initially operates on an unconfirmed itinerary. Whether booking fails or succeeds, the itinerary is moved from the planned itineraries database to the booked itineraries database.
The Find Booked Itinerary searches the booked itineraries for an itinerary with the given ID, just as the similar method from the Itinerary resource does for planned itineraries.

Resource: /booking/{ID}/cancel
Methods:
GET:

- **Name**
    - Cancel Booked Itinerary
- **Output**
    - ItineraryOutputRepresentation
- **Error Messages**
    - Itinerary not found : HTTP 404 NOT FOUND
    - Malformed Input: HTTP 400 BAD REQUEST
- **Links**
    - Find Booked Itinerary

Description

Cancelling a booked itinerary is done in the same way that you cancel a planned itinerary, however, instead of deleting the itinerary completely, it remains on the server until the first date in the itinerary.
If cancelling fails for some of the bookings, the user can see this by checking the status of each of the bookings in the returned itinerary.

Mediatype Support

RESTful webservices can produce and consume any number of different types of data. The two most common media types used for RESTful services are XML and JSON. In most cases, both of these media types can be supported without any significant effort. The only requirement for using the XML media type is that classes containing data to be transfered use the @XmlRootElement annotation. More complicated data structures may require more configuration for both XML and JSON support.

For our implementation, we support both XML and JSON input and output and could do so without significant effort.

Implementation of Operations

The implementation of most of the operations in the TravelGood RESTful service follow the same structure.
Whenever an ID is required as a parameter to an operation, one of the first things the operation does, after checking that it has received valid input, is to fetch the itinerary connected to the ID from the database.
After fetching the itinerary that should be worked on, the service performs its operation on the itinerary.
As a last step, before returning a response to the client, the operation builds a number of links to be added to the response.

Links

Links are used for creating a business process in REST-based applications. Each operation can return one or more fully constructed links to the client in the response header returned from a REST operation call. These links include all the information to successfully invoke the operation.
For example, after creating a new itinerary, the returned links already have the correct ID filled in and ready for use with the client.

Each of the operations, except for the search operation, in the TravelGood REST implementation has an associated relative link name. This relative name can be used by the client to query the response from a previous operation call for a fully qualified link to one of the next operations in the business process.

The link relative names and their associated operations are listed below:

- http://travelgood.ws/relations/add -> Add To Itinerary
- http://travelgood.ws/relations/findplanned -> Find Planned Itinerary
- http://travelgood.ws/relations/cancelplanned -> Cancel Planned Itinerary
- http://travelgood.ws/relations/book -> Book Itinerary
- http://travelgood.ws/relations/findbooked -> Find Booked Itinerary
- http://travelgood.ws/relations/cancelbooked -> Cancel Booked Itinerary

Problems and issues during implementation

The majority of the implementation of the REST service went without problems. RESTful services immediately feel familiar to Java programmers and after the initial resources, operation and representation have been defined, it is a simple matter of implementing them.

In the end, however, there was one unresolved problem.
Deleting a booked itinerary after reaching its earliest date proved problematic, as we could not get the timer to trigger the method that deletes the itinerary at the appropriate time.

The code is set up to store the dates of the hotels and flights that have been searched for, and when booking is completed, the earliest date can be found among the bookings. However, setting a timer to the earliest date failed at triggering at all.

4. Web service discovery

[1] Web Service Discovery is the process of a client finding the most appropriate web service for the required task. Web service providers typically provide a description of the web service using a WSDL file, where the data types used by the service, the operations that the client can perform, the binding port, and so on, are explained.

However, the service provider can also register the service using Web Service Registry, which improves the interaction between the client and the business service, or between different business services. One option for this was to register the business service in the Universal Business Registry (UBR) as a Universal Description, Discovery and Integration (UDDI). The UBR is a global registry where companies providing business services registered these. A client then could access the UDDI API to find web services and get information about them, such as the link to the WSDL file of the service.

Nevertheless, the Universal Business Registry is discontinued since 2006. This is why, nowadays, a distributed web service registry is more common. This is usually done by the Web Service Inspection Language, which is presented in XML format and is local to the service definitions (although it might contain links to other services' WSIL, or be linked from other services. Unlike in UDDI, there is no API for WSIL.

WSIL have been created in this project for 3 services: TravelGood (BPEL), LameDuck and Niceview, using procedures explained in the course literature and also in reference [2]. There is no WSIL file for the RESTful implementation of TravelGood. The files have the typically used "inspection.wsil" name, and have been included in the deployed services LameDuck and NiceView, where they can be found under the following URIs:

- LameDuck: http://localhost:8080/LameDuck/inspection.wsil
- NiceView: http://localhost:8080/NiceView/inspection.wsil

Note that, for TravelGood, the WSIL has been created but it has not been deployed with the service: this is because there is no place for a WSIL file in BPEL: the file should be accessed from the web content of the web services held by BPEL, but in this case we are using a composite application, so the WSIL file is not placed there. To show how the file would look like, we include a picture here:

```
<?xml version="1.0"?>
<inspection xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/">
    <service>
        <name>TravelGoodReservationAgency</name>
        <abstract>Agency for searching and managing flight and hotel reservations</abstract>
    </service>
    <link referencedNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
        location="http://localhost:8080/LameDuck/inspection.wsil"/>
    <link referencedNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
        location="http://localhost:8080/NiceView/inspection.wsil"/>
</inspection>
```

As it can be seen in the previous figure, under the <service> tag, the name of the service and a short abstract describing it are included. The <description> tag includes the link to the WSDL file of the service. After that, in the <link> tab, one can find URI's to the WSIL files of the other services that interact with this one.

It is also important to note that one <link> found in the WSILs of NiceView and LameDuck is linking to a fake URL for the WSIL of TravelGood, because, as explained before, this one has not been deployed.

5. Comparison RESTful and SOAP/BPEL Web Services

SOAP and REST are two different protocols used for web services. SOAP based web services are the original way to develop web services: they use the concept of web services. SOAP is XML-based and can be implemented over many different protocols, but it is usually implemented over the HTTP protocol, because this is the protocol used for the internet. However, an advantage of SOAP-based web service is that they can be implemented using other transport protocols, which make it more flexible. Web Service Description Language (WSDL) is used to describe a web service: it gives information about the data types (simple or complex) that are used by the service, the operations that it offers (and the elements or types in the messages of every operation), the binding port, and the location of the WSDL file.

Business Process Execution Language (BPEL) is a standardized SOAP-based language for Web Services. It is created mainly for web services implementing business processes, and provides an easier way to interact with other web services than the simple SOAP-based services. As SOAP, BPEL is also based in XML, and also provides a WSDL file for the description of the service.

Representational State Transfer (REST) is another architectural style used for web services. It is more recently created than REST and has some advantages over it: it is not based on web services and its operations, but on resources. These resources can not only be XML-based, but also use other representations such as JSON or HTML. In this project, XML representation was used, but integration of JSON is allowed. REST is based on the semantics of HTTP, and can rely uniquely on it, not in any other transport protocol. Although this might seem to be a limitation for some cases, in general it provides a much better integration with the HTTP protocol than SOAP-based services: it allows the service to deal with HTTP errors directly, and not with other kinds of errors such as in SOAP. The RESTful services do not provide WSDL files: however, Web Application Description Language (WADL) files shall be provided by the service, in order to describe the service provided along with the data types used and the resources.

After explaining some of the theoretical differences differences between SOAP/BPEL-based and RESTful web services, the main differences encountered during the implementation of the TravelGood service will be explained. The main and most obvious advantage found of the RESTful implementation over the BPEL implementation is that, once the resources are defined, the rest of the implementation of the internal logic is done in Java. This is very helpful for experienced Java programmers because it does not require a process of getting used to the tool. On the other side, BPEL requires to get familiar to its unique graphical interface, and to learn hot to use the objects and functions that it provides, such as the structures and activities (pick. forEach…) the variables and predicates….

However, this might be helpful for non experienced Java programmers: RESTful web services require knowledge of Java, so the graphical interface provided by BPEL could be more user friendly for those users. However, it can get very messy when dealing with complex projects, and specially with lists, because of the need to use predicates.

Another important feature of REST is the usage of links, due to REST being resource-based. It allows to link from one resource to another, and this is very helpful for the client, because it prevents him from having to specify the full URI of the resource for every different operation.

Something that was found during the development of the project was that BPEL is much more suitable for using parallelism and being able to run multiple processes in the service. This becomes specially useful for example when the service has to contact many other services: BPEL allows running multiple invocations on other services at the same time (although we had some problems with this in this project). On the other side, in order to achieve the same parallelism in REST, multiple threads would need to be created, so the complexity of the service would increase.

A particular case where parallelism becomes very handy is on timing events: BPEL allows to use OnEvent activities that are triggered automatically with timing conditions, and this an extremely useful feature for web services. To use timing events with RESTful services, other threads need to be created, that wait for a certain time trigger an activity (otherwise the service would be stopped and blocked).

Another little advantage of BPEL is the possibility to access data that was sent to the server by previous operations: this prevents the service from having to store some of the data sent in previous operations in variables, because they can be accessed from the ports of those operations at any time (except if a new operation is called in the same port).

On the whole, it was found that RESTful implementation is easier and more simple, and not only because of us having experience in Java, but also because of the complexity of BPEL when building a big web service: the graphical interface becomes almost unreadable because of the amount of boxes and arrows on it, and the program becomes slower and slower. An alternative is to write directly in the source XML content of BPEL, but this is also very complex and non-user friendly. The time it takes for the service to build and deploy, and also to debug, seems to be increasing exponentially as the size of the project increases: therefore, BPEL does not seem to be the best solution regarding scalability.

On the other side, RESTful services seem to be much more suitable for projects needing high scalability, because the code is organised in a much cleaner way than BPEL, with the possibility to be splitted in different classes. Debugging in REST is also much easier than in BPEL: it is faster as in normal Java applications, and also allows printing the value of variables, which cannot be done in BPEL.

6. Advanced Web service technology

Advanced web service technology is composed of 4 different areas that are the following ones:

- WS-Addressing
- WS-Reliable messaging
- WS-Policy
- WS-Security

In this section, we will discuss these different aspects in order to find out how our web services could have been improved in the context of a real-life project.

WS-Addressing is dealing with how to identify the different actors that are involved in a web service based communication process. One of the main problem of SOAP messages is that, by default, they don't allow to know who is sending messages to whom.
A known solution is to include information in the SOAP headers in order to identify the sender and receiver of each messages. By adding basic information such as <From>, <To> and a message <ID>to all our web services, we would be able to better identify our communication and by the same way, to improve the coordination of between our different web services but, also with the client. Implementing these tags would also benefit us later in the implementation of a reliable messaging protocol.

Adding a message ID would indeed be useful for us in order to detect SOAP messages that does not arrive to their recipient. To detect them and to be able to continue communicating correctly with the client, we would have to implement a reliable messaging protocol in order to make sure that all messages are reaching their recipient.
The implementation of this protocol (that can be external or internal to the web services) could benefit all of them but, it would mostly benefit TravelGood since he is a kind of coordinator between them. In fact, TravelGood is made in some way as a sequential process - first, you have to request the creation of an itinerary, then get booking numbers, add them to an itinerary, book it and eventually cancel it. Adding a *inOrder* delivery insurance would really benefit us by ensuring that all messages are received in the right order. For instance, it would make us sure that TravelGood is not trying to cancel an hotel or a flight that he haven't booked yet because the message has been lost or is arrived after the cancelling one for any reason.

But reliable message and message identification would not be the only features we should add to our web service in a real-life project context. Using WS-Policy standards on our web services would also benefit all of them.
In fact, TravelGood, NiceView and LameDuck are handling credit card information but, also with information related to the client private life. This way should use WS-Policy in order to make our web services more trustworthy for potential users. Two aspects seems to be particularly important here: quality of service and security. In fact, we should provide the user with explanation about the use we are doing with the information he is providing us but

also ensure him that nobody that shouldn't be involved in the planning process will be able to get his information. Security measures will be detailed later on, thanks to WS-Security.
Using WS-Policy would also benefit all our web services by solving problems linked to the web service discovery. In fact, we are using WSIL in order to make our services available to clients. This solution is enough for a student kind project and also avoided us having to register in order to use UDDI. However, it does not allow our web services to be easily found by other clients than the ones we are creating. In a similar real-life context, we should better use UDDI in order to publish our services. The main problem when using UDDI is that we don't have a lot information about the web service provider. We don't know who he is, what he is really doing and if all his services are updated. Using WS-Policy standards should so be a way to solve most the potential users' questions on our services.

Lot of these questions might be linked to security since all our web services are dealing with credit card information and with bank web services. They should so have increased security measures implemented. In our project, messages are not encrypted. It so easy for a third person to catch them or to change them. An example to highlight this security failure is the use we made of TCP Monitor. We were able to get the SOAP messages without problems. If it would have been real credit card information we could have gotten them only by listening to the right port. In order to solve this problem, we should deal with the following different requirements.
At first, in continuity to addressing and reliable messaging, we have to identify the clients and web services we are talking to and to create an authentication protocol in order to check their identities. Another measure, would be to moving from HTTP protocol to HTTPS in order to ensure confidentiality to our clients. To ensure privacy, integrity and authenticity, we should add encryption systems to our web services. The best solution for us would be to use a combination of Digital Signatures and Encryption. On the one hand, the Digital Signatures ensure the authenticity and the integrity of the exchanged messages, by using, for instance, a method based on hash functions comparing the digest of the messages. On the other hand, thanks to the encryption, we would be able to ensure the privacy of the information and this way to make sure that nobody read the message. This way, we should avoid symmetric key cryptographic systems because there is a lack of security while exchanging the key. If someone is intercepting the message, he would then be able to read messages. Solution as the use of a public key or RSA should be consider.

As a conclusion, creating a real service that ensure the customer of its right implementation needs to match all the different areas of advanced web services technology. The different solution we evoked in this part are link and could in some way be integrated together. For instance, the identification of the user in the messages would benefit addressing, reliable messaging and security as well.
If possible all our web services should be as secure as possible in order to make them trustworthy for clients. But if we also had to consider the cost of implementation of these different measures, we would then give priority to increase security of all our web services and to create a reliable messaging system for TravelGood. Describing the WS Policy of the web services in the context of UDDI publication would be the next step on our to do list.

7. Conclusion

The result after completing this project is totally satisfactory: although sometimes during the last few weeks we thought it would not be possible to finish on time, we have done it, with the two implementations of TravelGood (SOAP and REST) working correctly, and checking the correct operation of the system using many unit tests, apart from those required by the project report. There are some minor features that we have not focused on, such as the alarm implementation for terminating the service process when the date of the first flight or hotel arrives: we have taken the decision of not to focus on this so much because we chose to invest our time to improve the operations, resources and features in general offered by our web services.

The project done is in general very complete: the implementation of a web service for searching for flights and hotels and managing reservations is a quite huge project, and even more when it interacts with specific services such as LameDuck and NiceView that provide flight and hotel reservation services respectively. Moreover, these services interact with a bank service. All of this is a good example of the web services that we use everyday, and has made us learn a lot about web services and be prepared to implement other complex web services in the future.

One of the most important things to highlight is the amount of time spent working in the project this project. Even if our group was formed by 6 members, the amount of work to be done was huge, more than for any other course we have. However, the amount of work spent here has made us learn a lot about web services and the advantages and disadvantages of each one of the architectures: REST and SOAP.

Regarding the two possible architectures for the implementation (SOAP or REST), as it has been explained in section 5, each one of them has its own advantages, but, according to our experience, RESTful services seem to be the most comfortable solution to work with, mainly because of the high integration with Java, for its ease of use thanks to the resources and links, which are closer to the HTTP transport protocol, and also because of its higher scalability and integration.

This project would accept many improvements and extensions, mostly the RESTful implementation because of its higher scalability. Moreover, some of the implemented operations and features could also be improved: as we were progressing in the development of the project, we could see that other things that we had implemented before (such as NiceView and LameDuck) could have some of their parts improved (such as some operations or data types), but this would be dangerous because it would break the work done by other group members. But, as a future step, the services could be implemented in a cleaner and more tidy way (this is not the case for the RESTful implementation, because it is already quite optimized).

To sum up, we can say that we are very satisfied not only with the result of the project, but also with everything we have learned about web services in this course. We are sure that all of this will be helpful to us in the future.

8. Who did what

Contribution on the report:

Introduction	Raj
Coordination Protocol	Raj
Web service implementations	
Section on the data structure used	Quentin
Section for the airline and hotel reservation services	Rus
Section for the BPEL implementation	Ali C
Section for the RESTful implementation	Daniel B.
Web service discovery	Daniel Sanz
Comparison between RESTful and SOAP/BPEL	Daniel Sanz
Advanced Web service technology	Quentin
Conclusion	Daniel Sanz

Contribution on the project implementation:

During the 5 weeks of the project, we arranged weekly meetings during the lab sessions but also on every tuesday afternoon from noon until the evening. Presence of all the group members was mandatory and none of us missed any of them.

This project has been for us a way to exchange a lot on our comprehension on web services and to work on the different part of the project. Even if the following participation is not equally split, each member of the group has been involved in the implementation of all different types of web services and should now have acquired knowledges on web services and their implementation.

Daniel Gert Brand - s123230:

- **LameDuck:**
  - WSDL file
  - getFlights, bookFlight, cancelFlight
- **BPEL:**

    - WSDL file
    - Operations: *getFlightsAndHotels*
    - Unit tests (testC2)
- **REST:**
    - Defined resources and methods to use for the operations. Defined and implemented links.
    - Resources: Itinerary, Booking
    - Data types: Itinerary, AddToItineraryInputRepresentation, BookItineraryInputRepresentation, CreateItineraryInputRepresentaiton, ItineraryOutputRepresentation, LinkRelatives
    - Unit tests: ItineraryResourceTest

Daniel Sanz Ausin - s142290:

- ***LameDuck:***
    - WSDL file
    - *getFlights* operation,
    - Unit tests
- ***BPEL:***
    - WSDL file
    - Operations*: createItinerary*, *getFlightsAndHotels*, *planFlightsAndHotels*, *cancelPlanning*, *bookItinerary*
    - Unit tests (including *testP1* and *testP2)*
- ***REST:***
    - Resources: *Search*
    - Data types: *searchInputRepresentation, searchOutputRepresentation*
    - Unit tests (including *testC1* and *testC2)*
- ***WSIL:***
    - *inspect.wsil* (3 files: TravelGod BPEL, NiceView and LameDuck)

Ali Chegini - s150939:

- ***LameDuck:***
    - WSDL file
    - *getFlights* , bookFlights and cancelFlights
    - all the JUnit tests for bookFlights and cancelFlights
    - Debugging and fixing issues
    - LameDuck.xsd
- ***NiceView***
    - *Debugging fixing issues related to exceptions*
- ***BPEL:***
    - WSDL file
    - Operations*: getFlightsAndHotels*, *planFlightsAndHotels*, *bookItinerary, cancelPlaning, cancelItinerary*

- *implementing compensation handlers for bookItinerary loop*
- creating 7 JUnit tests inside *TravelGoodClientTest.java*
- Debugging and fixing issues
- correlations set and fixing the issues related to binding
- test scenarios (B)

REST:

- *Resources: ItineraryResource.java*
- *BookingResourceTest.java (small part)*
- *UnitTest* scenario (B)

Raj Kumar Kalvala - s040551:

- ***NiceView:***
  - WSDL file
  - Unit tests implementation
- ***LameDuck:***
  - Provided support and help on implementation
- ***BPEL:***
  - Researches and suggesting solutions
- ***REST:***
  - Sharing improvement ideas and suggestions
  - Helped on implementation of BookingResources

Rustam Ali Hussaini - s150962:

- ***NiceView:***
  - WSDL file and web service implementation
  - Unit tests implementation
- ***LameDuck:***
  - Provided support and help on implementation
- ***BPEL:***
  - Researches and suggesting solutions
- ***REST:***
  - Sharing improvement ideas and suggestions
  - Helped on implementation of SearchResources

Quentin Tresontani - s151409:

- ***NiceView:***
  - WSDL file
  - *getHotel* operation, *bookHotel* operation, *cancelHotel* operation
  - implementation of the Database.java file

    - Unit tests
- ***BPEL:***
    - WSDL file
    - xsd types importation in the wsdl
    - Work on the *cancelItinerary* operation, on the booking compensation handler and on fixing bugs while running and writing unit tests.
    - Unit tests (including *B, C1, C2)*
- ***REST:***
    - Resources: *Booking resource* : implementation of the compensation loop, and logic of booking (not involved in creating the links)
    - Data types: Refactoring some resources thanks to *ItineraryOutputRepresentation*
    - Unit tests (including *testP1* and *testP2)*

8. References

[1] *Web Services - Joan Ribas Lequerica*

[2] *Web Service Discovery - Wikipedia*
https://en.wikipedia.org/wiki/Web_Services_Discovery

[3] Lecture Slides