DTU - DENMARKS TECHNICAL UNIVERSITY

Software Development of Web Services (02267)

Travel Good - Group V

s131343 - Diego González
s131650 - Alina Gherman
s131342 - Miguel Gordo
s131857 - Pierre Emmanuel Dalidec
s131889 - Soukaina Hsaini
28. October 2013

CONTENTS

1 Introduction

1.1 General Introduction

For the final evaluation of the course 02267: Software Development of Web Services, students are asked to realize a project involving all the knowledge acquired about Web Services along this semester. This project consists of the creation of a web service designed to enable a client to plan his/her own trip. In order to do so, the client will use the services provided by TravelGood, the trip registration company, to create and book the itineraries wanted.

These itineraries are composed of the hotels where the clients will stay and the flights they will take to go from one city to another. To manage the bookings of both flights and hotels, TravelGood will directly communicate with the relevant services which are the ones provided by NiceView, the hotel registration company, and LameDuck, the flight registration company. It will also give the important information about the trip to the client, as for instance the price of the flights or the name of the hotels. A last service will finally be considered in the process, the one enabling the client to pay the trip he booked and provided by the bank FastMoney.

Our tasks in this project will be to implement the LameDuck and NiceView services as SOAP based web services, and the TravelGood service both as a BPEL and RESTful web service. The FastMoney service already exists and is deployed as a SOAP based web service; we will thus not need to implement it but only to adapt our services so that they can communicate properly with FastMoney.

The rest of this report will be structured as follows:

Section 2 will describe the coordination protocol of our system, illustrated by a state diagram. Section 3 will specify our Web service implementation, going through data structures and BPEL/RESTful implementations of the services. Section 4 will focus on Web service discovery and describe the WSIL files created for our services. Section 5 will consist of the comparison between RESTful and SOAP services, highlighting the advantages and drawbacks of both implementations. Section 6 will deal with the topic of advanced web services. Section 7 will be the conclusion of this document and Section 8 will present the work each group member did in this project.

But before presenting our results, we will focus more deeply on the basic concepts of web services in the second part of this introduction. This part will introduce more precisely the following concepts: service orientation (SOA), basic service technologies, Web service description (WSDL), Web service discovery (WSIL and UDDI), Web service coordination, Web service composition, and RESTful services.

1.2 INTRODUCTION TO WEB SERVICES

This part introduces the basic concepts of Web Services.The first concept that needs to be introduced is the service-orientation.'Service-orientation Architecture (SOA) is an architectural concept where the software components are modeled as services. Each service has an interface that allows him to interact with other services.

SOA is based on these following principles.

- Service Loose Coupling: this principle promotes the independent design which means that the parts of the architecture are self-standing units and each can be reused by multiple architectures.
- Service Abstraction: the service only describes the interface and not the underlying details of the service.
- Service Discoverability: the service needs to be easily identified and understood when opportunities for reuse present themselves.
- Service Composability: the service is expected to be able of participating as an effective composition member, regardless of whether it needs to be immediately enlisted in a composition.

The web services are a type of SOA. They have the advantage of using a simple text-based XML messaging to communicate and supporting both RPC and document-centric messaging which makes them highly flexible.

The Web Services exchange SOAP messages based on standard protocols like HTTP. The advantage of using these messages is that they can be created and received by applications written in any programming language and on any operating system. However, the SOAP does not provide any standardized data types or structures.

That is why, we use XML Schemas so that any complex data type can be defined and shared by Web Services. The WSDL schema must be exchanged between the different services because it provides an interface for the principle Web Service.It describes the services operations and the data types manipulated during the operations. Each operation implemented contains certain logic like invoking other services. Besides, the WSDL defines messages including complex data types and defines the ports where the service can be found.

Concerning the discovery of web services, both the Web Services Inspection Language (WSIL) and the Universal Description Discovery and Integration (UDDI) are ways that facilitate it.

The major difference between UDDI and WSIL lies in the cost and complexity. On the one hand, UDDI is like a "Yellow Pages" directory where published Web services from various organizations can be categorized and organized. Hence, organizations can share and use UDDI registries to maintain a number of Web services under different categories.

On the other hand, the WSIL is a cheaper solution for organizations to share Web services. In this case, the WSDL can be stored at any location, and requests to retrieve the information are generally made directly to the entities that are offering the services through regular Web servers without a comprehensive and complex services registry infrastructure.

In our project, we will use the WSIL for the web services discovery.

Other concepts to be introduced are the Web Services Coordination and Composition which are essential between multiple services.

Web Services Coordination, also called Choreography, describes how Web services interact. This external view can be described using a state machine, an activity diagram or a sequence diagram. In our case, we are going to focus on the state machines for the coordination protocol between the client and the travel agency TravelGood.

Web service composition, as part of a SOA and also called orchestration, describes how a Web service is constructed from other Web services. In this context, a high degree of automation is desired, which can be achieved by using BPEL (Business Process Execution Language) which enables us to develop processes quickly by defining the order in which services will be invoked.

Finally, the last concept to introduce is the RESTful Web API. Restful Web services are based on the structure of the Web as defined through HTTP. It uses the concept of resources and links. A client can access resources using the unique URI and a representation of the resource is returned. With each new resource representation, the client is said to transfer state. In RESTful Web Services, we use standard HTTP operations such as GET (get the resource), PUT (update the resource), DELETE (delete the resource) and POST (create a resource). These operations are equivalent to create, read, update and delete (or CRUD) operations.Another difference with the SOAP web services is that the RESTful ones are stateless.

2 Coordination Protocol

To interact with TravelGood the client has to start by creating an itinerary. After having an itinerary the client will be able to find out information about hotels and flights (get hotels, get flights) and add them to the itinerary (add hotels, add flights). He can perform each of these actions in any order. Then, the client can book the itinerary.

Note: The client cannot book each flight/hotel separately; he can book only the whole itinerary.

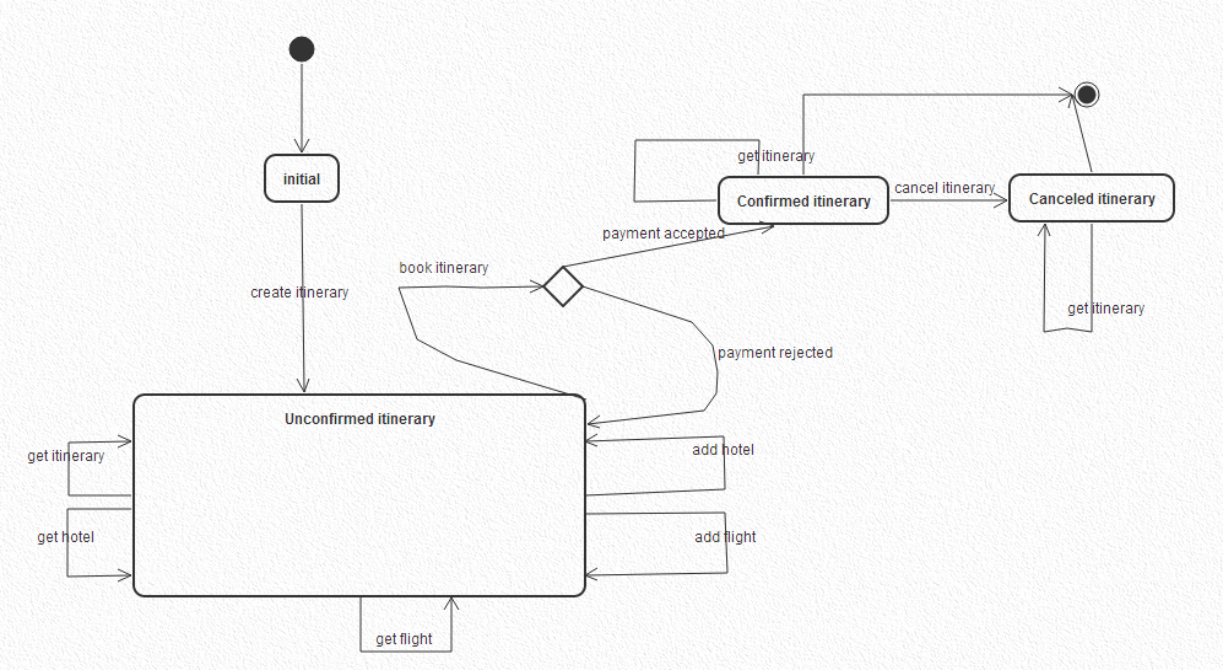

Figure 2.1: State machine representing the coordination between the client and TravelGood

If the action of booking the itinerary has been successfully executed, the itinerary will pass in a new state: confirmed. If an error has occurred for the booking of one of the hotels/flights, the itineraries will pass to the unconfirmed state and all confirmed hotel/flights before the exception will be cancelled.

If the itinerary is in a confirmed state, the user cannot add more flights or hotels to this itinerary. Also, he cannot make any other getFlight or getHotel within the same process.

Any confirmed itinerary can be cancelled, but we cannot cancel an unconfirmed itinerary. Also, after an itinerary is cancelled the process is arriving to the final state.

A user can at any time start creating more than one itinerary in parallel.

3 Web service Implementation

3.1 Data structures used

3.1.1 Data structures for TravelGood as a RESTful service

The structure of the used entities in the RESTful web services is presented through this UML diagram (Fig.3.1 below).

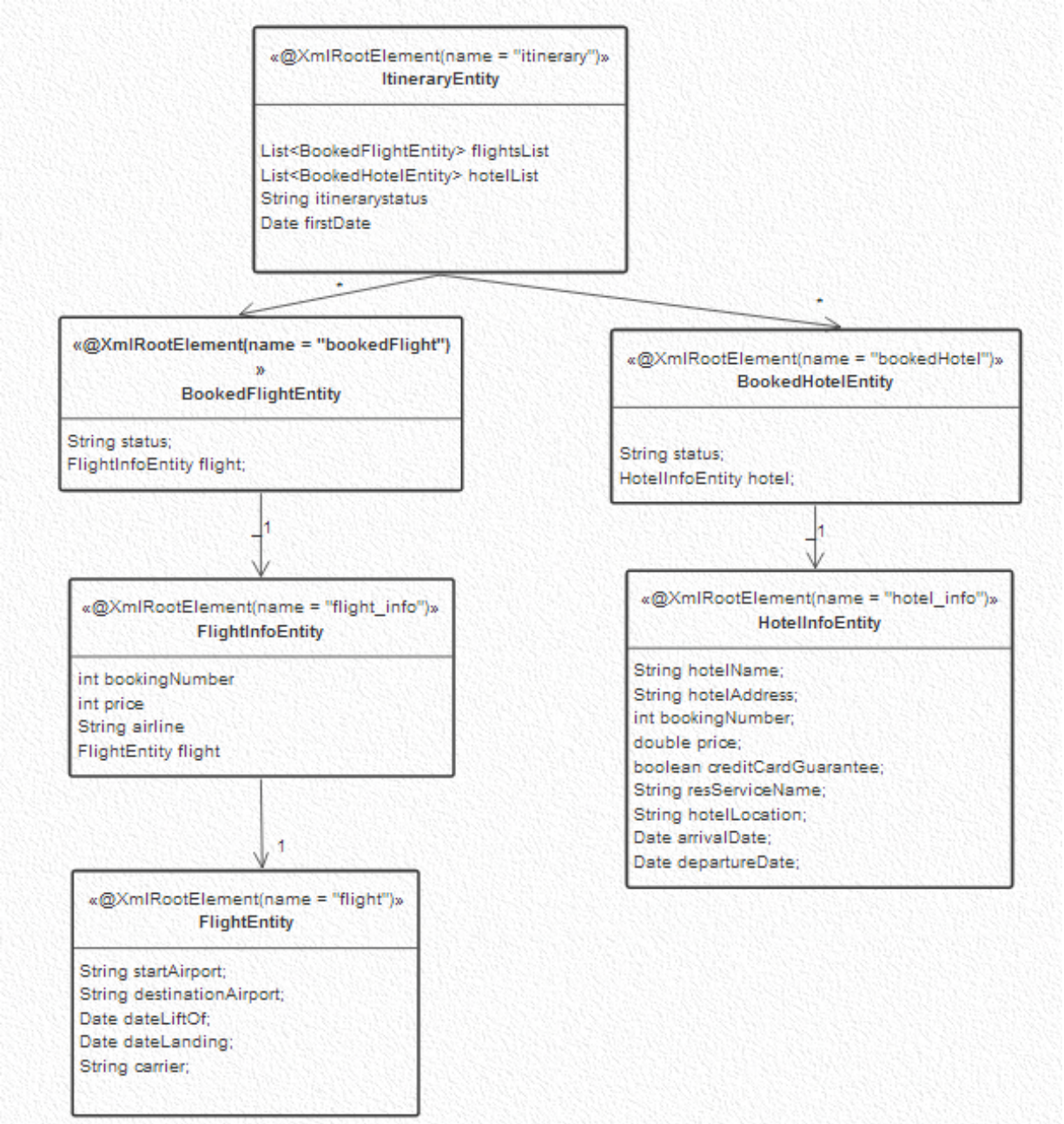

Figure 3.1: RESTful data structure

The ”ItineraryEntity” is representing an itinerary. This itinerary contains two lists: one of flights and one of hotels. Each list contains the whole object (flights/ hotel) in order to be able to return to the client all the information about the flights/hotels he booked. This information is also stored for being able to have the CRUD operations.

The ”BookedFlightEntity” and the ”BookedHotelEntity” are two entities that are contain-

ing the Flights/Hotels and their status. It is important to keep their status in order to be able to cancel all the booked Flights/Hotels if the booking of the itinerary fails. Also, this status is needed in the getItinerary operation. The FlightInfoEntity (and FlightEntity) and the HotelInfoEntity are based on the LameDuck and NiceView messages.

LameDuck and NiceView messages are not structured in the same way. If FlightInfoEntity is containing the bookingNumber, the airline and a Flight object inside, the HotelInfoEntity is not containing any Hotel object inside. This object is containing all the necessary hotel informations: hotel name, hotel address, booking number, price, credit card, reservation name, hotel location, arrival date and departure date.

3.1.2 Data structure for TravelGood as a BPEL process

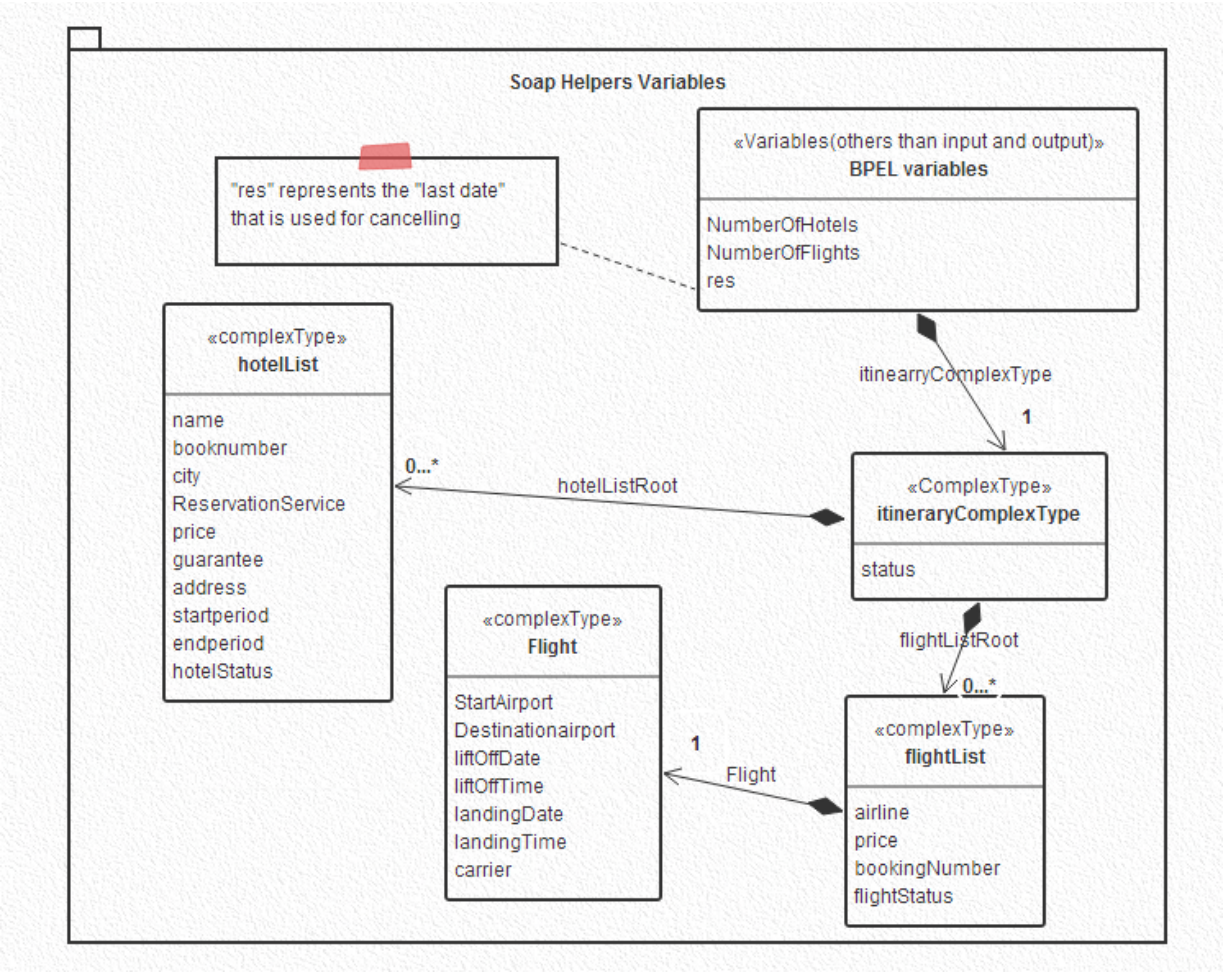

Figure 3.2: SOAP data structure

SOAP Data structure is more difficult to represent through an UML diagram since all the needed data are created as variables. In fact, variables are used as inputs/outputs for op-

erations but also as "helpers" for storing data through the same process. However, the represented data structure of these helper variables is presented in figure 3.2 above.

As for the RESTful data, one itinerary variable is stored. This itinerary variable contains only two lists: a hotel list and a flight list with their respective status. This itinerary variable contains also a variable for storing the status of the itinerary.

Also, since the clientId is not used in the correlation set, it is not stored nor checked, there is no need to use a variable only for it (another variable apart the variable used in the operation).

The counters: variables NumberOfFlights and NumberOfHotels are used so that we do not count the number of elements of the lists each time we need to loop over them.

Also, we are having a date variable that will store the last date in which the client can cancel the itinerary.

3.2 Airline and Hotel services

In this section we present the Airline and Hotel Services, named LameDuck and NiceView respectively. They are responsible for keeping all information about flights and hotels, booking them and also cancellation. The interface for this services is well specified in the Project Description document. For these services we decided to create LameDuck with RPC style binding and NiceView with document style binding.

3.2.1 Hotel Services: NiceView

NiceView is the company responsible for finding and booking hotels at the request of TravelGood, a request that ultimately has been made by a user editing an itinerary. The company offers its operations in a WSDL document implemented with document binding. With it we can avoid the need to know about the RPC convention beforehand, even though it implies a bit more complication in the WSDL. In the end we choose it so as to do one service, LameDuck, with RPC binding, and the other (NiceView) document-binding.
We store three hotels. We initialize the data the first time any of the operations offered (getHotel, bookHotel and cancelHotel) is called, and from then onwards it is never modified. The three operations offered are those specified in the problem Description document for the project. They offer the specified functionality and take the specified arguments. The data structures used in the WSDL file are stored in an external XML Schema file, *hotelschema*, which can be easily imported in any other WSDL in need of them.

3.2.2 Airline Services: LameDuck

LameDuck is the company which offers the services of getting a list of flights, booking flights and cancelling them. It offers its services as SOAP-based Web services (as it is stated in the Problem Description). To implement it, we first made a xsd file which has the definition of the elements and types used in the WSDL. Second, we made a WSDL file with the interface of the operations. These operations are described in the Problem Description so we just had to decide how to put them into a WSDL file. We chose the RPC binding style because it allows us to have more than one part in a message and this makes the interface of the operations with the client very clear.

From this WSDL file we created a class that implements the web service. As the goal of the project is not to develop a real tool, the class that implements the operations only pretends that it has a flights data base and it has 4 flights hard coded in the class. The operations implement the functionality required in the Problem Description and thus they do not look if a particular cancellation is possible. Furthermore, LameDuck does not keep track neither of the bookings nor of the seats on the flights as this is not the objetive of this project.

3.3 BPEL IMPLEMENTATION

In our BPEL implementation ,TravelGood process performs eight operations:

createItinerary	getItinerary	getFlights	getHotels
addFlights	addHotels	bookItinerary	cancelItinerary

This section will introduce the implementation of each operation. In order to be able to perform asynchronous actions, we implemented correlations. We defined one correlation set for the itineraryID initialized in the response of the createItinerary Response. We also used different schemas from the Web Services: Bank, LameDuck and NiceView. They were used in importing some complex types used in Travel Good. Furthermore, we have a new web service called DateCalculator which allows Travel Good to calculate the latest day when a booked itinerary can be cancelled. In the WSDL file of Travel Good there are described all operations that the service may perform.

- CreateItinerary operation: This operation takes a customerID and sends him back an itineraryID. We also have a variable itinerary that contains a list of the hotel bookings and the flight bookings. So in this operation, we should initialize all the elements of the itinerary variable. We also have two other variables that should be initialized in this operation, the first one is called NumberOfFlights which represents the size of the list of flights in the itinerary variable and the second one, called NumberOfHotels represents the size of the list of hotels .After the initialization step, the customer has an empty itinerary and gets an ID for it. We also initialize a starting date in order to compare and calculate later in the process which day is the day before the itinerary starts. This is done via a call to the DateCalculator auxiliary service.

- getFlight operation: This operation allows the customer to search the flights that he wants, so in the request he needs to precise the start and the destination airports, the date of the flight and the itineraryID. After invoking the getFlight operation from LameDuck Web Service, the operation returns a list of flight information containing the booking number, the price, the airline, the start and the destination airport, date and time of landing, date and time of lift-off and the carrier.

- getHotel operation: This operation allows the customer to search the hotels that he wants. In the request, he needs to precise the arrival and departure dates, the city and the itineraryID. After invoking the getHotel operation from NiceView Web Service, the

operation returns a list of hotel information containing the booking number, the price of the stay, the name of the hotel, its address, a creditcard guarantee which is a Boolean stating if the guarantee is required or not and finally the name of the HotelReservation service.

- addFlight operation: This operation allows the customer to add a flight to his itinerary. For that, TravelGood needs to provide the itineraryID and information about the flight which is the response of the getFlight operation. In this operation, we need to increment the NumberOfFlights variable to show that we added a flight element in the list and then we reply by a confirmation (True Boolean). In this operation,we compare the starting date of the new added flight with the starting date of the other elements that are already in the itinerary.Then we keep the closest date.

- addHotel operation: This operation allows the customer to add a hotel to his itinerary. Like in the addFlight operation, TravelGood needs to provide the itineraryID and information about the hotel which is the response of the getHotel operation. In this operation, we need to increment the NumberOfHotels variable to show that we added a hotel element in the list and then we reply by a confirmation (True Boolean). In this operation,we compare the starting date of the new added hotel with the starting date of the other elements that are already in the itinerary.Then we keep the closest date.

- getItinerary: This operation allows the customer to get the current itinerary, he needs to provide the itineraryID and the operation returns the itinerary containing the list of flight and hotel bookings. These lists contain also a status showing to the customer the current status of his bookings.

- bookItinerary: In this operation, the customer can book his itinerary. In the BPEL, we have two ForEach loops, one for booking each flight and which invokes the bookFlight operation of LameDuck and the other for booking each hotel and which invokes the bookHotel operation of NiceView. In both cases, the customer should provide the booking number of the flight or the hotel he wants to book and the credit card information. The web service returns a Boolean confirmation true if the booking was successful. If the booking fails, we execute a compensation handler where we invoke cancelFlight or cancelHotel then we set the status of the flight or the hotel that failed to unconfirmed. We also have a FaultHandler for handling the faults coming from invoking cancelOperation.

- cancelItinerary: In this operation, we invoke the DateCalculator service by assigning to it the most approaching date of all the bookings, it calculates the duration in ms between today and a day before the beginning of the trip. Then we send this duration , which represents the remaining time for cancelling the itinirary, to the alarm that sets a variable finishProcess to true when the time is completed. This variable shows actually when the overall process is done.So when it is true , the customer cannot cancel his itinerary when it is false,the customer can still cancel his itinerary. If the itinerary is confirmed, we proceed like in bookItinerary , we have two ForEach loops, one for cancelling each flight using the cancelFlight operation of LameDuck and the other one

for cancelling each hotel using the cancelHotel operation of NiceView. If the itinerary is already cancelled, we send a message ”already cancelled”. The cancelItineray succeeds when all flights and hotels have been cancelled.

The figure below shows the BPEL implementation for the Travel Good Process.

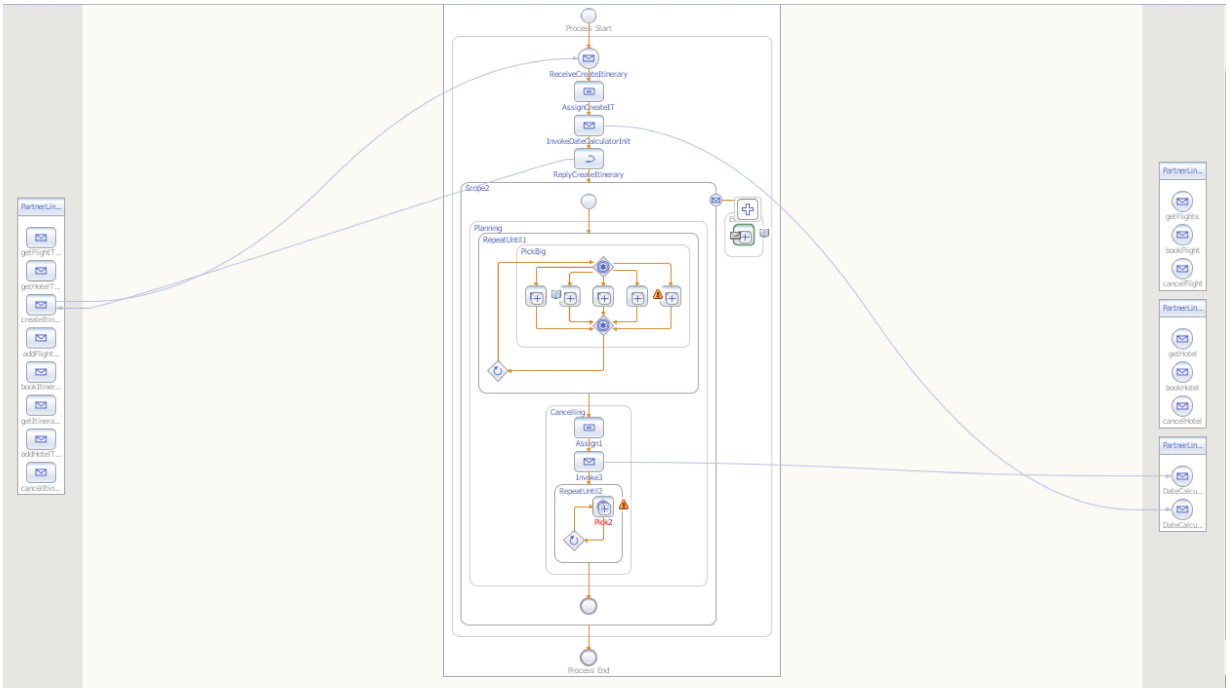

Figure 3.3: BPEL implementation for the Travel Good Process

The figure below shows the BPEL implementation of the operations :getFlights, getHotels,addHotels,addFlights,getItinerary and createItinerary.

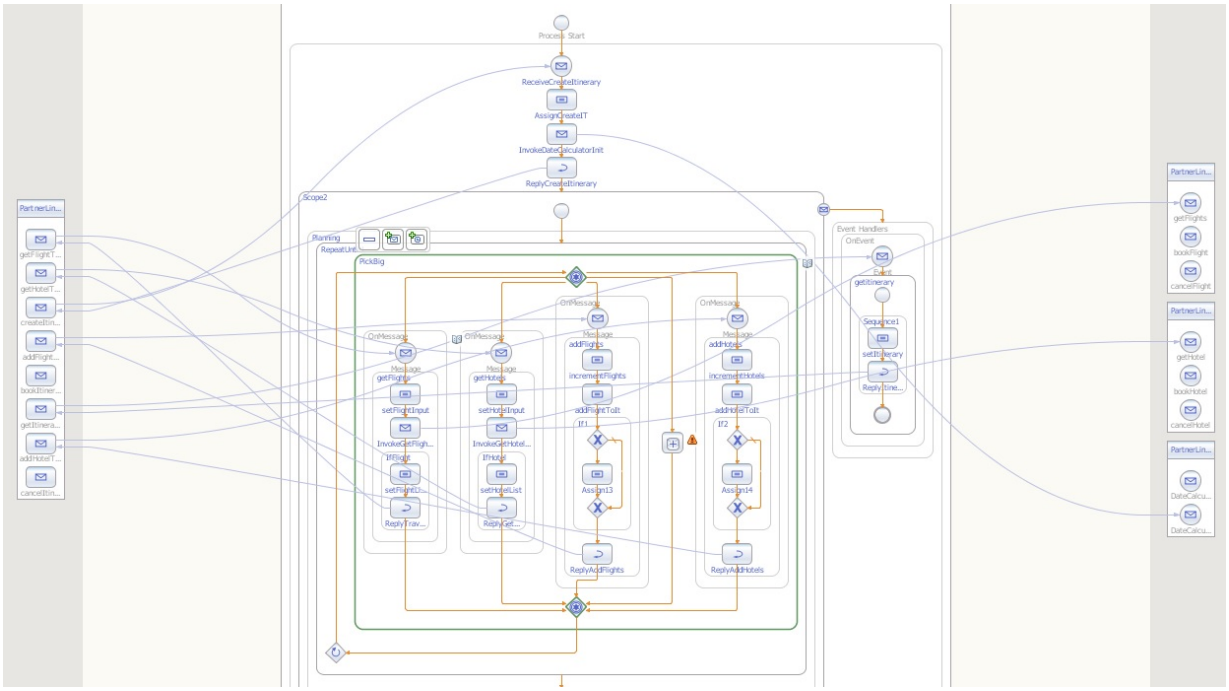

Figure 3.4: BPEL implementation for getFlights, getHotels,addHotels,addFlights,getItinerary and createItinerary

The figure below shows the BPEL implementation of the bookItinerary operation.

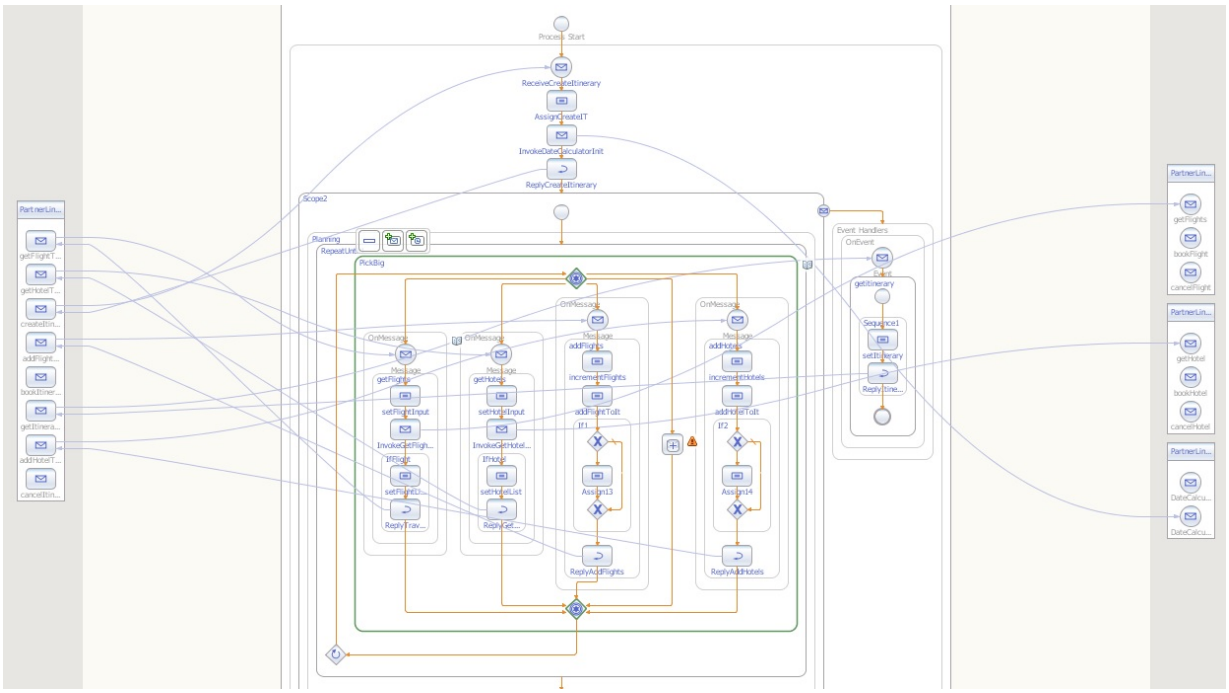

Figure 3.5: BPEL implementation for bookItinerary

The figure below shows the BPEL implementation of the cancelItinerary operation for the hotel part.

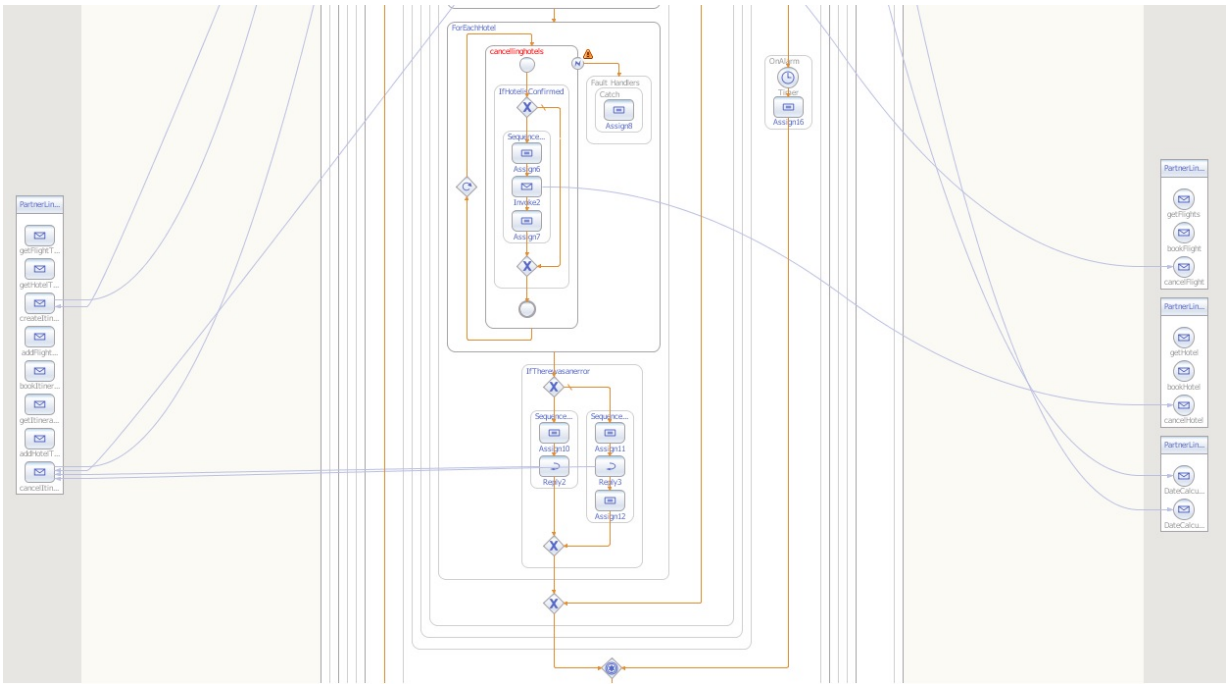

Figure 3.6: BPEL implementation for cancelItinerary

3.4 RESTFUL IMPLEMENTATION

The other way to proceed to the implementation of the TravelGood service is to use a RESTful implementation instead of a SOAP-based approach. The main difference here is that the implementation of RESTful Web Services and their data structures are relying on the concept of resources and not on the concept of services. Each of these resources is identified by a URI providing a path to access them, and associated with a media type describing their actual representation (e.g. XML, HTML, etc.).

All these different resources created for the whole implementation can be accessed through the usual HTTP methods which are PUT, GET, POST and DELETE. The aim in the RESTful implementation is to define how these operations will be used to act on the existent resources and therefore to access or modify the data. The services we have created have been thought to only provide a restricted access to the data through the use of these methods. That is, every operation depends on the ID of the client requesting it: it is thus impossible for him to act as he is not supposed to, for example to change his own client ID.

You can find below a table displaying an overview of the implementation of our services through the URI Path of the resources we used, the corresponding classes implemented, and

the HTTP methods used on each of these resources.

URI Path	Resource Class	HTTP Methods
/flights/{iid}	FlightResource	GET
/hotels/{iid}	HotelResource	GET
/itineraries/reset	Itineraries	PUT
/itineraries/{iid}	ItineraryResource	GET
/itineraries/{iid}/flightadd	ItineraryResource	PUT
/itineraries/{iid}/hoteladd	ItineraryResource	PUT
/itineraries/{iid}/book	ItineraryResource	PUT
/itineraries/{iid}/cancel	ItineraryResource	PUT
/login/{cid}	LoginResource	PUT

Figure 3.7: RESTful implementation of the resources

As we can see in the table, our choice for the resources used has been to create one resource for each of the main objects we are going to manage; that is, flights, hotels and itineraries, that can all be accessed by using an itinerary ID. Basically, these resources are used to get information from each of these objects. We also needed what we called a login resource, that can be accessed by using a customer ID, and enabling the creation of a new itinerary. Finally, we created a resource for each order related to the managment of the itineraries, which are: reseting an itinerary, adding a flight, adding an hotel, booking an itinerary and canceling it.

The two first resources of the table are implemented the same way, enabling to get the information related to either flights or hotels after receiving the corresponding request from the client. The corresponding classes are respectively FlightResource where the getFlight method is implemented and HotelResource where the getHotel method is implemented. That is why the only HTTP operation used here is the GET operation, in order to simply access the information from these resources.

The only other resource using the GET operation is accessed using the path "itineraries/{iid}", the itinerary ID enabling a client to uniquely access the itinerary he wants information about. Similarly to the two previous methods, the corresponding class is ItineraryResource and the actual name of the corresponding method is getItinerary.

All the following methods will access the different ressources thanks to a PUT operation, including the ones theoretically creating something; that is, the LoginResource and the two ressources enabling the addition of a flight or a hotel. As an explanation, in these methods

the names of our orders have already been decided, and the server does not decide anything about the names of these orders in the process. Therefore, even though a POST operation is usually mapped to the Create function of CRUD, we will use here PUT operations that can also be used for creation purposes (even though it is more commonly mapped to the Update function of CRUD).

The LoginResource class contains the login method, performed through a PUT operation that enables a new empty itinerary to be created when a new user logs in thanks to his customer ID (cid). This resource is accessed by following the path "login/{cid}".

The PUT operation used on the resource attached to the "itinerary/reset" path will essentially be used for test purposes, as it enables to reset the data relative to an itinerary before potentially try to add flights and hotels and book them afterwards. This reset method is implemented in the Itineraries class.

The addFlight and addHotel methods are implemented the same way and basically enable to add either a new flight or hotel to the already existent list. To proceed with these additions to the different lists, we use the PUT operation that will allow us to access them respectively at the paths "itinerary/{iid}/flightadd" and "itinerary/{iid}/hoteladd". The presence of the itinerary ID is explained by the need of specifying on which itineraries we want to add a new flight or hotel. Besides, both these methods are implemented in the ItineraryResource class, as they are parts of the global process of creating a whole itinerary.

The resource linked to the "itinerary/{iid}/book" path will be accessed using a PUT operation again, in order to proceed with the bookItinerary method. The aim of this method is to act more particularly on the status on both the list of flights and the list of hotels belonging to an itinerary (once again identified by its itinerary ID), so that it becomes a "confirmed" status, which basically means that the itinerary is actually booked. This method is implemented in the ItineraryResource class and also handles several possible faults or errors that could happen in the process.

The last method considered and displayed in this table is the cancelItinerary method, also implemented in the ItineraryResource class. This method enables us to access the status of an already booked itinerary, and so the status of both flights and hotels constituting it, through the path "itinerary/{iid}/cancel". We use here again a PUT operation to change this status, as in the previous method. cancelItinerary also handles several faults or errors likely to happen and related to the time of the cancel attempt or to the original status of the itinerary for example.
The ItineraryResource class also contains all the methods for the creation of the links that are needed by the other classes in the perspective of the RESTful implementation.

As for the representations we first decided to create a class Link along with its associated mediatype, relation and URI parameters. We also created an abstract class Representation, from which all the other Representation classes will inherit from. In this class, we define a

list of links along with the method getLinkByRelation allowing the further correspondances between links and relations.

All the other representation classes are directly inherited from this parent class. HotelRepresentation and FlightRepresentation are then implemented in the same way, creating respectively a list of hotels and flights (using the corresponding entities). The itineraryRepresentation only contains an itinerary, that will own both a list of flights and a list of hotels. Finally, we also created a StatusRepresentation class which creates a String parameter used for the status.

The entities are implemented in the entity package where we can find the constructors of the actual objects managed in the service that is, hotels, flights and itineraries, but also clients and status. These objects own all the parameters described in the Problem Description file of this project and mentionned in the previous sections.

Finally, a last package called "utils" is used to define all the constants, paths and relations needed in the Constants class.

4 WEB SERVICE DISCOVERY

As mentioned in the introduction to Web Services in Section 1, the discovery of Web Services can be made either using UDDI or WSIL. In the frame of this project, we will use WSIL files to provide the discovery feature. Therefore, we created three WSIL files, respectively for LameDuck, NiceView and TravelGood, that can be found below in figure 4.1.

As these files are supposed to be easily accessible for a user in order to discover new services, they will respectively be stored in the root directory of the corresponding services. These directories especially contain a META-INF folder and an index file both corresponding to the service considered. Thus, the "inspection.wsil" of LameDuck and NiceView can be respectively found at the following path in our project: ServiceSOAP/Service/build/web (replace the word "Service" respectively by "LameDuck" or "NiceView"). As there is not the same folder architecture in the TravelGoodSOAP directory (with the index and the META-INF), we put the corresponding WSIL in the root directory of TravelGoodCompositeApp. After the deployment of the services, the WSIL files can be found at the usual place, which is for example for LameDuck: localhost:8080/LameDuck/inspection.wsil.

Each of these WSIL files are of course XML files and everything in these WSIL files is written between two <inspection> tags, which are the main tags of any WSIL file. The common part of the three files is the description of the three services in the beginning, each service description being contained between two service tags (<service></service>). For each service, the description consists of the actual name of the service, a short sentence informing us about the services offered, and the location of the corresponding WSDL file.

Then, the second part composed of the link tags (<link></link>) is not the same for each

WSIL file. For each service, these link tags will enable to create a pointer towards the two other services which directly communicate with it. This is the step enabling the discovery of services related to the ones already known through the use of WSIL. This time, we provide in the link tags the location of the WSIL file related to the service which is pointed out, along with a brief description of the service.

```
<?xml version="1.0" encoding="UTF-8"?>
<inspection xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/">
    <service>
        <name>LameDuck</name>
        <abstract>The airline reservation services offered by LameDuck </abstract>
        <description referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
                     location="http://localhost:8080/LameDuck/LameDuckService?wsdl"/>
    </service>
    <link referencedNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
          location="http://localhost:8080/NiceView/inspection.wsil" >
        <abstract>Hotel registration company</abstract>
    </link>
    <link referencedNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
          location="http://localhost:9080/inspection.wsil"
        <abstract>Travel planning company</abstract>
    </link>
</inspection>
```

Figure 4.1: LameDuck WSIL file

```
<?xml version="1.0" encoding="UTF-8"?>
<inspection xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/">
    <service>
        <name>NiceView</name>
        <abstract>The hotel reservation services offered by NiceView</abstract>
        <description referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
                     location="http://localhost:8080/NiceView/NiceViewService?wsdl"/>
    </service>
    <link referencedNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
          location="http://localhost:8080/LameDuck/inspection.wsil" >
        <abstract>Flight registration company</abstract>
    </link>
    <link referencedNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
        location="http://localhost:9080/inspection.wsil"
        <abstract>Travel planning company</abstract>
    </link>
</inspection>
```

Figure 4.2: NiceView WSIL file

```
<?xml version="1.0" encoding="UTF-8"?>
<inspection xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/">
    <service>
        <name>TravelGood</name>
        <abstract>Planning and booking services provided by The TravelGood agency
        </abstract>
        <description referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
                 location="http://localhost:9080/service?wsdl"/>
    </service>
    <link referencedNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
          location="http://localhost:8080/NiceView/inspection.wsil" >
      <abstract>Hotel registration company</abstract>
    </link>
    <link referencedNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
          location="http://localhost:8080/LameDuck/inspection.wsil" >
      <abstract>Flight registration company</abstract>
    </link>
</inspection>
```

Figure 4.3: TravelGood WSIL file

5 COMPARISON RESTFUL AND SOAP/BPEL WEB SERVICES

SOAP and the RESTful web services are two different approaches of implementing web services. The two approaches can be implemented in a lot of different languages.

These approaches of implementing web services are very different in terms of type of implementation but also in terms of way of representing and managing the resources.

Firstly, RESTful web services can only be transmitted over the HTTP transport protocol while the SOAP web services can be transmitted over different transport protocols: HTTP, JMS and SMTP. In this project we choose of course to use the same transport layer: HTTP.

Secondly, these two approaches of web services are very different in the way we have to perceive the services that we have to offer. For the RESTful approach we had to focus on resources and links: what data does the user have to be able to have access to and by which way. For the SOAP approach we had to focus on the services: what operations should the user be able to do.

In terms of implementation, for RESTful web services there is one way of implementing: creating the resources and the links that will let others web sources/applications to access the resources. SOAP web services can be created by two approaches: bottom-up or top-down.

1. With the bottom-up approach we create a web service by creating simple methods (that represent operations that are to be possible to call) to which we have only to add one annotation: @WebService. The WSDL will be generated from these methods. Thus, with this method of creating SOAP-based web services we have to be careful about the shape of the messages we are transmitting.

2. The top-down method of creating SOAP-based web services consists in creating the WSDL that is requested for communicating with other web services and generating from this WSDL the respective stubs. With this approach we are having full control over the shape of our messages and over the operations that we want the user to have access to. While creating the WSDL we are not thinking at all about resources or data. We are only thinking about the operations which have to be available and the shape of the messages that are to be transmitted through these operations.

Another difference is that SOAP-based web services can communicate either one with each other, either within a common channel of communication, by implementing a SOA (Service-Oriented Architecture). In Java, this kind of communication can be implemented by using a JBI (Java Business Integration). In order to create this channel within this project, we used a BPEL support and for creating this BPEL a "drawing"is created. This "drawing" can easily be understood by a non-developer person. For the RESTful approach it is not commun to use a commun channel of communication: we are only communicating from one service to another by the way offered to us and which we could consult in a WSAL.

If within the SOAP approach,"the drawing" used for the SOA architecture can be easily understood by a non-developer person, the RESTful approach cannot be easily understood by a non-developer person. For a developer, the RESTful approach of implementing a web service could be preferred to the SOA architecture constructed with the help of a BPEL since the code is the normal way of creating programs (without speaking of creating interfaces for programs).

Also, the web services should be maintainable. If the business process of the company

changes, the web services should be easily changed in order to add new requirements. For web services using a RESTful approach, that could easily be done because changing one web service normally means changing one or two files of code: the resource file, and possibly the related entities and test files. The web services that depend on this web service have to "be announced" by changing also the WSAL. For a SOAP-based web service, that means changing the WSDL (if a top-down approach was used) and then, if a SOA architecture is involved, changing the BPEL support. The related services will know about the change through the WSIL. The impact of the change in the SOAP web services depends on the type of the change.

Moreover, with the RESTful web services, CRUD operations can more easily be done. If CRUD operations are desired Within a SOAP web service approach, the WS-Transfer standard should be used since this standard provides loosely coupled CRUD endpoints.

Another difference consists in the fact that RESTful web services are more suitable to mobile devices for which the overhead of additional parameters like headers and other SOAP elements are less. For websites, the RESTful web services are also more suitable since JSON messages can be send (and JSON is the standard Java script object).

To sum up, both approaches can be used in order to create web services. Thought, in some types of applications, one approach can be more suitable.

6 Advanced Web Services

In the following section we will present some extensions to SOAP Web Services, briefly describe them and discuss how our project would benefit with the use of these additional mechanisms.

6.1 WS Addressing

WS Addressing is a specification that seeks to include routing information in the SOAP headers and take that responsibility from the network layer. In this way, the SOAP applications can know who the sender is, whom to reply, if they are the rightful recipient and so on and so forth.

Perhaps its most important feature is that it can separate the HTTP GET/REQUEST protocol from the SOAP application itself, enabling long-running asynchronous interactions. This feature is obtained with the tag "ReplyTo", establishing a new connection to the endpoint specified in such tag. In our project we could use this specification, which would make our connections more general, but would imply a change in the established communication protocols so as to make use of the features above. It would be more useful for identifying if the senders of the messages are those we expect communication from, and also for sending our own addressed to the correct end user.

6.2 WS ReliableMessaging

This specification defines a way to reliably send messages between a source and a destination over an unreliable infrastructure. It is based on the standard of SOAP and WSDL. The process is transparent to the application source (AS) and the application destination (AD). The AS

sends its messages to a Reliable Messaging Source (RMS) and the RMS communicates with a Reliable Messaging Destination (RMD) and this is the one that gives the messages to the AD. We did not use this standard in our project but it would have benefited the final product in many aspects. For example, this standard can be used to ensure the ExactlyOnce delivery assurance. When the client adds a flight, it is very important that the flight is added because otherwise the traveler could have a lot of trouble far away from home (if he had not checked the itinerary before booking it). In addition we would like that the message of addFlight arrives only once because due to some retransmission protocol, the message could arrive two or more times and in that case, the client would have more reservations than he wanted.

Clearly the Delivery Assurances of AtLeastOnce, AtMostOnce and ExactlyOnce would benefit all the interactions between the client and the server. In addition, the WS ReliableMessaging standard also supports the InOrder assurance. Imagine that the client has been adding flights and hotels to his itinerary and then he wants to book it. Obviously he wants that the messages of addflights and addhotels arrive earlier than the order to book the itinerary.

6.3 WS Security

SOAP-based Web Services not using any kind of extensions have critical security issues which would make them useless in a real life environment. If we think about our connection with the BankService, we send very sensible data (credit card information) without encryption and can also be modified by an attacker if he intercepts the message. Because SOAP messages are sent in plain text, they are very vulnerable to Man-in-the-middle (MiM) attacks. Anyone in our wifi network could intercept our packages and read them without effort, and use that information for their advantage. In our particular case, the use of our tool could result in an identity theft with a man-in-the-middle attack. It is thus necessary to provide SOAP with an extension that can ensure some of the goals of cyber security, such as confidentiality, integrity, authentication, authorization... We cannot rely on the security of the protocol layer because SOAP is in principle independent and should provide its own security mechanisms.

WS Security is an extension to SOAP to provide some security mechanisms to Web Services. This is a protocol specified to ensure that integrity and confidentiality are enforced in Web Services' communications. In particular, it provides mechanisms for:

- How to sign some or all parts of a SOAP message to assure integrity.
- How to encrypt some or all parts of a SOAP message to assure confidentiality
- How to attach security tokens to ascertain the sender's identity.

This last point is for network protocols, such as Kerberos, that rely on a trusted authority to issue tickets for a user in order to be authenticated and gain access to some resource.
In our project, all comunication with the bank would benefit with the use of WS Security, and we could also implement it whenever the user starts adding flights/hotels to the itinerary as the itinerary data should also stay private.

6.4 WS POLICY

The Web Services Policy Framework (WS-Policy) provides a general purpose model and corresponding syntax to describe the policies of a Web Service. WS-Policy is designed to work with the general Web services framework, including WSDL service descriptions and UDDI service registrations. In our case, as we did not use any additional feature and option (like security) it does not make much sense to implement it. We could just support WS Policy specifying that we do not use any special policy, to make it more general. But, in a better implementation, our product should use at least the security policy for sending bank information. In this case, it makes sense to use also the WS Policy standard, because for instance there are a lot of ways to sign a message or to encrypt it. The WS Policy allows us to define that our Web Service supports (for example) SHA-1 with RSA and MD5 with RSA as signing methods and that the user must use at least one of them to communicate with the service.

7 CONCLUSION

In this project, we have implemented the services needed as we were requested to, using different kind of web services implementations such as SOAP-based web services, composite web services and RESTful web services.

Through this report, we have illustrated in the first place the coordination protocol taking place between the TravelGood service and the client, explicating how it is possible to go from one state to another one. Then, we detailed more precisely how we designed the several aspects of the global implementation, displaying an overview of the data structures both for BPEL and RESTful using UML diagrams. We also proceeded to the description of the service offered by both the NiceView and the LameDuck companies before describing more precisely our implementation for both BPEL and RESTful implementations.

Finally, after mentioning the discovery of web services and therefore the WSIL files attached to this project, we discussed about web services related topics by drawing a comparison between the two approaches favoring the RESTful one and then having a deeper look into advanced web service technology.

On a technological aspect, this project has enabled us to go further into the design and actual implementation of web services, the main concern often being to make everything work together and to make the two aspects of the TravelGood implementation correspond to each other. The fact of properly transcribing the methods, entities and their sets of data in the implementation of our services was also a challenge, as it is important to correctly manage all of this in order to design the actual service requested in the problem description. The validation of the final tests has also been an important issue, as they forced us to come back and modify our implementation in order to make everything work properly.

On a more general aspect, we may have made some mistakes in the beginning of the project in terms of team management. That is, we might have spent too much time on discussing the project after the very first meeting instead of actually splitting the work and assigning tasks. Thankfully, we quickly corrected this in order to create some deadlines for everybody and fairly distribute the tasks so that everybody could be able to work on every part of this project. Also, as mentioned before, some errors about the comprehension of the problem description and the data management could have been avoided and made us lose less time during this project.

As a conclusion, this project has been an efficient way to strengthen our knowledge and skills in web services implementation. It enabled us to have a good overview of this topic through the example of the TravelGood services. Moreover, it has been challenging our management skills, as it was necessary to properly manage a whole group in order to meet both the requirements and the deadline of this project. Although some mistakes could have been made in the group management process, our group has been working well together; as each member did the work he or she was requested to do. Besides, the mistakes we had to deal with

can only be a good thing regarding future experiences and more particularly future projects we will be assigned on.

8 WHO DID WHAT

- Diego:
  - LameDuck: Creation and implementation of the files AirlineSchema.xsd, LameDuck.wsdl and LameDuck.java. Tests to prove the correctness of LameDuck.
  - TravelGood in REST: Implementation of the files: BookedFlightEntity.java, FlightRepresentation.java, Itineraries.java and LoginResource.java (the last two are web resoures). Implementation of the methods of ItineraryResource.java (main resource for the itinerary): addFlight and cancelItinerary. The test TestItineraryResource.java.
  - TravelGood in BPEL: Implementation of the messages, operations of cancelFlight in travelGoodWsdl.wsdl. Part of the process to cancel a booked itinerary in travelGoodBpelModule.bpel. Operation getItinerary. SmallTest file.
  - Report: section 3.2.2, AirlineServices: LameDuck. sections 6.4 and 6.2 of Advanced Web Services. Section 8, Who did what.
- Miguel:
  - NiceView: Creation and implementation of the files hotelschema.xsd, NiceView.wsdl and NiceView.java. Tests to prove the correctness of NiceView.
  - TravelGood in REST: Implementation of the files: BookedHotelEntity.java, ItineraryEntity.java, StatusRepresentation.java. Implementation of the methods of ItineraryResource.java (main resource for the itinerary): getItinerary, addHotel and bookItinerary. The file Constants.java.
  - TravelGood in BPEL: Part of the module travelGoodBpelModule.bpel to end the process one day before the itinerary starts. The auxiliary module CalculatorDate (the .wsdl file and the .java file) and modifications to calculate the duration of the process (once booked). Testing of the alarms
  - Report: section 3.2.1, Hotel Services: NiceView. Sections 6.3 and 6.1 of Advanced Web Services.
- Alina:
  - TravelGood in REST: Implementation of the files: FlightEntity.java, FlightInfoEntity.java and ItineraryRepresentation.java. The resource FlightResource.java.
  - TravelGood in BPEL: The following parts of travelGoodBpelModule.bpel: getFlights, addFlights, bookItinerary part flight. Implementation of the Unit test of each operation to check whether they are working properly.Requested tests. Initial version of travelGoodWsdl.wsdl.

  - Report: section 2 Coordination Protocol, section 3.1 Data Structures used, section 5 compararision RESTful and SOAP BPEL Web Services.

- Pierre-Emmanuel:
  - TravelGood in REST: Implementation of the files: HotelRepresentation.java and HotelInfoEntity.java. Implementation of the file HotelResource.java (resource). Implementation of the method getHotel and contribution to addHotel.
  - TravelGood in BPEL: Implementation of the messages, operations of addHotel in travelGoodWsdl.wsdl. Hotel part of bookItinerary method.
  - Report: Section 1.1, General Introduction. Section 3.4 of Web Service Implementation. Text part of Section 4, Web Service Discovery. Section 7, Conclusion.

- Soukaina:
  - TravelGood in REST: Required tests for the RESTFUL implementation.
  - TravelGood in BPEL: The following parts of travelGoodBpelModule.bpel: createItinerary, getHotels, addHotels.Implementation of the Unit test of each operation to check whether they are working properly.
  - Report: Section 1.2, Introduction to Web services. Section 3.3 of Web Service Implementation. Section 4,Web Service Discovery.