

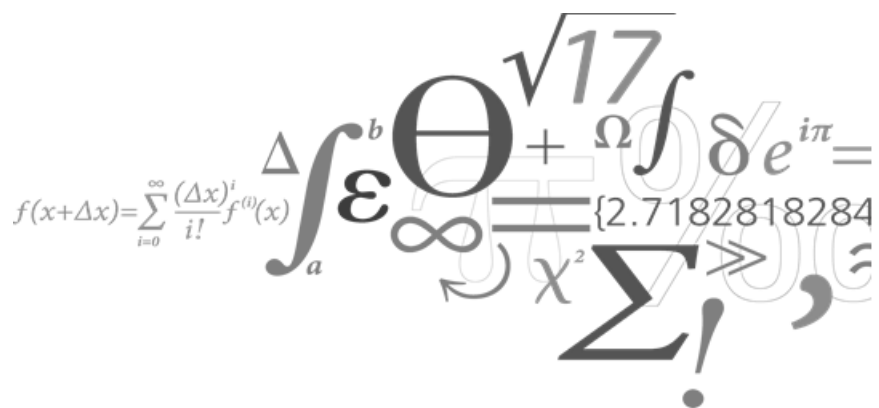
02267 SOFTWARE DEVELOPEMENT OF WEB SERVICES

---

## TRAVEL GOOD WEB SERVICE

---

<b>Group 11:</b>	Paulina Bien	s131107
	Monica Coman	s130919
	Oğuz Demir	s114503
	Audrius Grigaliunas	s141759
	Cæcilie Bach Kjærulf	s093063
	Johannes Sanders	s142249

A collection of various mathematical symbols and expressions arranged in a cluster. Visible symbols include:  $f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$ ,  $\Delta \int_a^b \epsilon \Theta$ ,  $\sqrt{17}$ ,  $\Omega \int \delta e^{i\pi} =$ ,  $\infty$ ,  $\chi^2$ ,  $\Sigma$ ,  $!$ ,  $>>$ ,  $\{2.7182818284$ , and  $\approx$ .

# 1 Introduction

This section should introduce the project and the material covered in the report.

## 1.a Introduction to Web services

What exactly is a web service? It is a software function, that supports machine-to-machine interactions over a network. Mostly web services are used for connecting existing software stubs into one usable service, but there is a broad variety of web services usage. With the help of web services we can solve the interoperability problem when linking or exchanging data. Web services describe a way of integrating web-based applications using common standards like XML, SOAP, WSDL and UDDI. It is also broadly used for business to communicate with each other and with clients. It does not provide a graphical user interface for the user, but it shares business logic, data and processes through a programmatic interface, therefore, the usage of Web Services has a very important role in business processing.

Web Services Discovery - in order to make a web service usable, service provider needs to make it accessible to potential users. It can be located through a public or private business registry, or a WSIL document. Discovery is managed by UDDI (Universal Description Discovery and Integration), where distributed registry of business and its services implementation is being set for potential clients. WSIL is an alternative to UDDI, which allows to go directly to the web service and ask for the possible services to use. Discovery works as the client is looking for one or more services and the services, and then as a result, which match the criteria is being sent as a response directly to the client. After discovery process completion, the service provider and client application is able to know the exact location of service (URI), its capabilities and how to interact with it.

Web Service Composition is being more and more used, because composite services have unique features, that an individual service cannot support. Semantic web is a more advanced form of the web, where all the contents have well defined meaning, because of composition. In web service composition services can automatically select, integrate and invoke different other web services in order to achieve user specified tasks. Composing existing web services to deliver new functionality should be used by all the companies, because it provides new functionality with the ease of use. For example, LameDuck, NiceView and TravelGood together was made as a composition, when planning, booking and payment connects with each other and is being used as one dependent application. Composition extends the notion of service discovery by enabling automatic composition of services to meet the requirements of a given task description. It can be either static (where requirements are pre-defined) or dynamic (requirements are given on the spot). It connects a variety of different services in a composite application, so many services can be used in one application and one web service is dependent on another.

WS-Coordination is a Web Services specification which defines a framework for protocols that coordinate the actions of distributed applications. Coordination is one of the major parts in business processing. Such coordination protocols are used to support a number of applications, including those that need to reach a consistent agreement on the outcome of distributed transactions. The framework defined in this specification enables an application

service to create a context needed to propagate an activity to other services and to register for coordination protocols.

RESTful Services (Representational state transfer) is an abstraction of the architecture of the web. It is an architectural style consisting of a coordinated set of architectural constraints applied to components, data elements within a distributed system. If applied to a web service, REST improves desirable properties such as performance, scalability on a distributed system. Data, functionality and resources are accessed using Uniform Resource Identifiers. REST manipulates resources using a fixed set of operations: PUT, GET, POST and DELETE, where PUT creates a new resource, which can be deleted by DELETE, GET retrieves the current state of the resource and POST commits a new resource state.

Service orientation is a design paradigm to build computer software in the form of services. The idea behind this is to automate the business logic as a distributed system. System orientation is a set of design principles which carries out the separation of concerns in the software. Orientation promotes loose coupling between software components so that it can be reused. The idea of service orientation is to do a service that is easy to use.

Basic service technologies have components such as XML, SOAP, WSDL, UDDI. First of all, web services communicate using SOAP messages. They are being exchanged using XML style documents. There are also two types of SOAP interaction styles: document and RPC style. The main difference between RPC and document style is that RPC style pattern is used, when the client uses web service as a single application with data inside. Request and response messages map directly to the input and output parameters. For example single input for converting temperature from celsius to fahrenheit, contains simple input and output. However when document style is used, client uses web service as a long running business process. Here input parameters are a complete unit of information (using an instance of another web service as an element), such as transaction, where credit card information must be verified during the process. And there main web service calls another web services to use another service helping functionality.

WSDL (Web Service Description Language) is an XML based document which contains a set of definitions that describe the functionality offered by a web service. It has its own structure where types, messages, operations, port types, bindings and services are defined. That document defines services as collections of networks endpoints, or ports. In WSDL, it is possible and important to be able to describe the communication between devices using web services in a structured manner.

XML (eXtensible Markup Language) is a markup language that defines a set of rules for encoding documents which is readable for both: machines and people developing a code. The goals of XML emphasize simplicity, generality and usability across the Internet, but the main usage of XML in web services is message exchanging in SOAP.

## 2 Coordination Protocol

We assume that the clients connecting to *TravelGood* are stand-alone applications calling our web services. This process starts by sending a customer/client reference identification through the *CreateItinerary* operation, which will create a unique itinerary for the client. In

the interaction with the client, the state itinerary changes and this is represented in the state diagram below (see figure 6).

In the state diagram several operation that can change the state of the itinerary are defined. Most of them, such as *AddToItinerary*, *GetHotels/Flights*, *BookItinerary*, *CancelItinerary* are self explanatory and well-defined in the project description. However, we should mention that *TerminateItinerary*, although possible from all states (unbooked, booked, cancelled) has different meanings. The user is able to terminate the itinerary by calling the *CancelItinerary* operation in the unbooked state. Concurrently, in the booked and cancelled states, it might happen that the itinerary is terminated whenever the date of a flight or a hotel has passed.

As soon as an itinerary is terminated, it is removed from *TravelGood*'s record therefore its status can not be called for anymore.

The coordination process between *TravelGood*, *LameDuck* and *NiceView* are similar for both our REST and BPEL implementations. Communication with the client happens only via the *TravelGood* web service which will afterwards concurrently connect to the external web services (*LameDuck* and *NiceView*). Moreover, both *LameDuck* and *NiceView* call operations from the *FastMoney* web service. All the connections between client the and services are portrayed in figure 2.

As it can be seen as mentioned above, the client only interacts with the *TravelGood* service which will afterwards start a new itinerary and return its identifier. In BPEL, *TravelGood* creates a GUID identification for further communication whereas in REST, a unique string identifier is used. After the itinerary has been created, the client can request available flights and hotels from *TravelGood*, which will in turn request the info from the *LameDuck* and *NiceView* services. The result of *GetFlights* and *GetHotels* operations will be an itinerary which holds two arrays, one for the list of flights and one for the list of hotels.

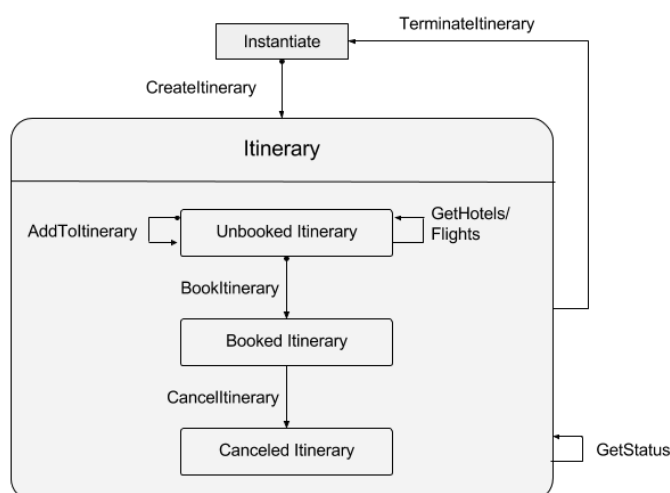


Figure 1: State Diagram for the coordination protocol between the client and *TravelGood*.

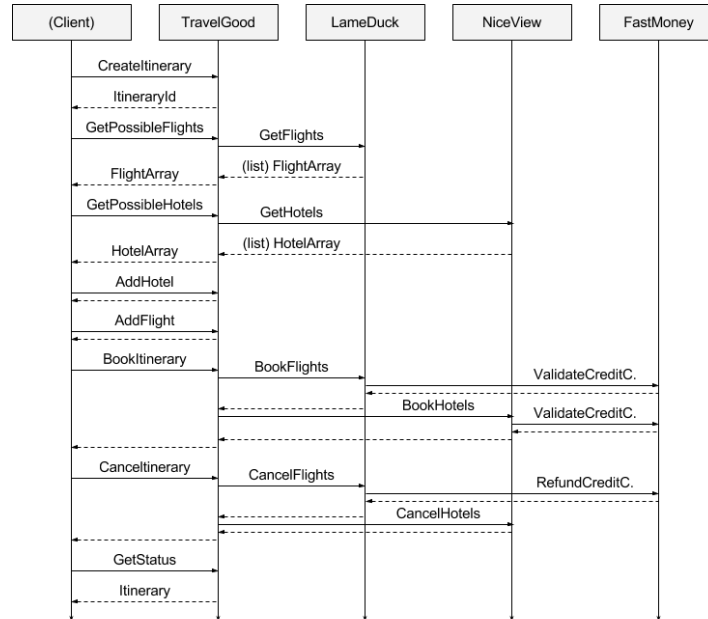


Figure 2: Sequence Diagram for TravelGood, LameDuck and NiceView.

During the booking (*BookItinerary*) process, the client application is required to also send the bank information along with the itinerary id. This operation can return a fault if booking fails and all exceptions are handled properly (see 3. Web Service Implementations). Similarly, the cancel itinerary operation (*CancelItinerary*) also requires a booking number and credit card info and can return an error if some parts of the cancellation fail.

### 3 Web Service Implementations

#### 3.a Data Structures

The data structures used within the *TravelGood* RESTful and SOAP/BPEL are depicted in figure 3 and figure 4. Although describing the exact same process, the two implementations of *TravelGood* are different and therefore use somewhat different data structures. More details about what are the main differences follow in the discussion below.

*Itinerary* (*ItineraryInfoType*) is the main data object that will be manipulated by *TravelGood* operations. It contains two arrays of *FlightInfo* (*FlightInfoType*) and *HotelInfo* (*HotelInfoType*) data objects that will be populated by the user during the planning phase when calling the *addFlightToItinerary()* and *addHotelToItinerary()* operations. Although the *FlightInfo* and *FlightInfoType* classes contain the same attributes (as is the case for *HotelInfo* and *HotelInfoType*), different data structures are created separately for both REST and SOAP/BPEL services since the first one requires classes to be annotated with *XMLRootElement* whereas the latter works with auto-generated classes from the WSDL complex types declarations. Same principle applies for the *CreditCard* (*CreditCardInfoType*) data object required for communication with *FastMoney* web service.

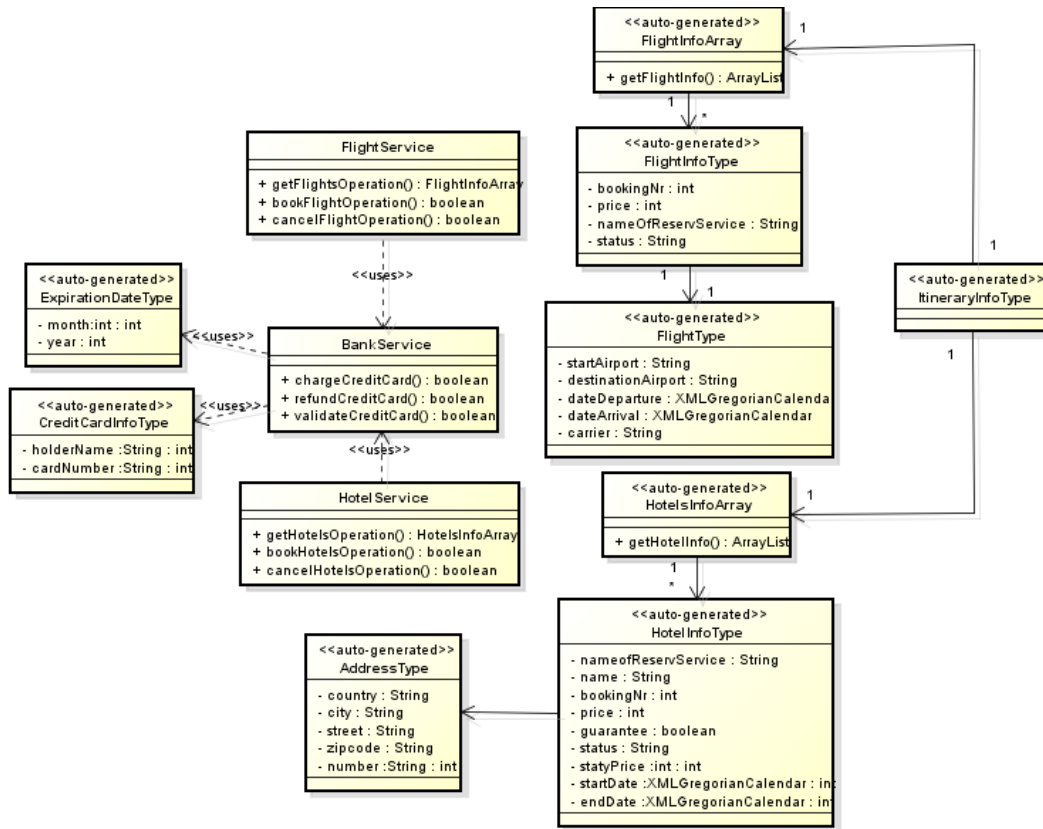
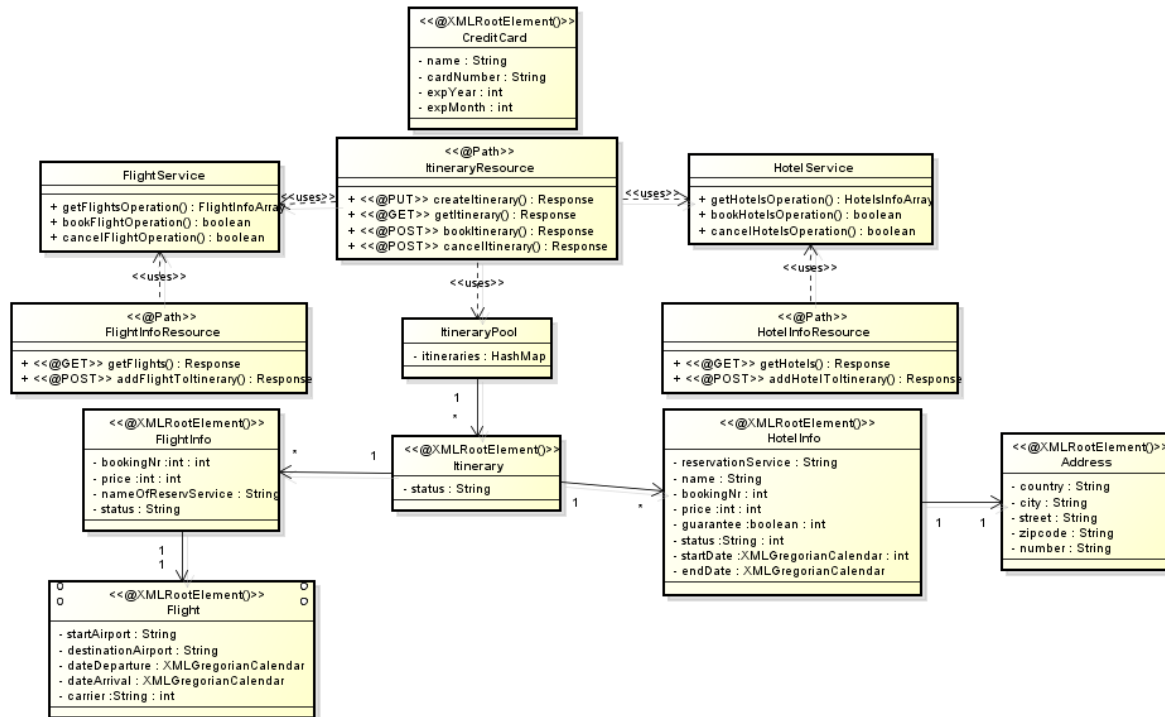


Figure 3: Class diagram of *TravelGood* BPEL/SOAP.

Information about flights and hotels is stored, maintained and manipulated by the external web services, *LameDuck* and *NiceView*. Therefore, when calling their operations, both implementations of *TravelGood*'s operations need to make sure that the right input types are used. In BPEL, this consistency is easily ensured by having an XML Schema Document enclosing the declaration of all complex types necessary for *LameDuck*, *NiceView* and *TravelGood*. In REST, however, this is not as straightforward since *TravelGood* uses *XMLRootElement*s such as *FlightInfo*, *HotelInfo*, *CreditCard* which need to be transformed to *FlightInfoType*, *HotelInfoType* and *CreditCardType* in order to be understood by the external web services. This transformation is ensured inside the *FlightService* and *HotelService* classes (see figure 4).

In addition, inside *LameDuck* and *NiceView* services, each flight and hotel data is represented with *Flight* (*FlightType*) and *Hotel* (*HotelType*) classes where their instances contain static data (name, address, start airport, destination, etc.) of the entity they are representing. When a user requests a flight information, we create the instances of classes *FlightInfo* (*FlightInfoType*) and *HotelInfo* (*HotelInfoType*) that contain additional information (such as booking number, price, status) on top of *Flight* and *Hotel*. With these additional information, the flight or hotel can be booked and the bookings can be tracked.


Figure 4: Class diagram of *TravelGood* REST.

Last but not least, REST uses resources ( *Path*) on top of all the data objects discussed above which can be accessed by the user via URLs in order to execute different operations (create, book, cancel, etc). This is not the case in BPEL, when only complex types are used for all client-WS or WS-WS interactions. This matter will be discussed more in-depth in the following sections.

### 3.b Lame Duck and Nice View Web Services

As described in the problem description for this project we created web services for both *NiceView* and *LameDuck*. We implemented both web services as simple web services without databases. Objects containing information about available flights/hotels are stored in static arrays. In addition, we assume that all of the resources are unlimited which means that is it always possible to book every item which is listed in a list of available flights/hotels returned to the client. However, booking numbers are locally unique. That means when creating a bookable flight info, the booking number of the flight will be unique across the flight web service and this will remove the risk of booking number collisions which can cause bigger problems. The same steps regarding booking number are performed when creating a bookable hotel info. Booking numbers are represented as integer values. The next booking number is calculated using increase-and-set method. There could have been a better approach such as using GUID as a booking number, however using integers simplifies the process.

The main structure of both services is the same. There are three packages:

- `ws.flight/ws.hotel` – packages which contain classes with implementation of web services' functionality (see below) as well as classes with the bank service operations,
- `ws.flight.builder/ws.hotel.builder` – in order to simplify the process of creating objects a builder pattern has been implemented,
- `ws.flight.query/ws.hotel.query` – packages which contain query engine that returns available hotel/flight information based on the query values.

In addition, there is a debugging package for the flight service called `ws.flight.debugger` which contains a class used to print out operations together with parameters on the server side in order to get a better understanding of what is being called. It was particularly useful during implementation and verification of BPEL part of the project.

Moreover, a reference to the service *FastMoney* has been added to both *LameDuck* and *NiceView* to set up the following interactions with the bank:

- charge credit card,
- validate credit card,
- refund money if booking is cancelled.

The functionality offered by both *LameDuck* and *NiceView*, as specified in the problem description, is:

- getting list of available flights/hotels according to the query parameters (returning array of flights/hotels),
- booking flight/hotel,
- cancelling booking of flight/hotel.

The Web services have been generated from WSDL files and the used binding style is document/literal. The reason for using this structure is to keep the service simple and describing exactly what is communicated in the XML according to the predefined schema.

### 3.c SOAP/BPEL Web Service

BPEL (the business process execution language) is an entirely Web Services based language that offers flexible coupling between several systems. Moreover, Web Services are based on XML and they provide means of communication over a network in an open and flexible way. The objective all of these technologies is to automate process executions across system and the client. BPEL specification was written by Microsoft, IBM, and BEA and it consists of web services definition language (WSDL) files defining the underlying Web Services and BPEL files defining the entire business process. All BPEL processes have only one variable variable (which, in our case, would be the *itinerary*).

BPEL process indicates the order in which Web services should be invoked. There was used `<receive>` and `<reply>` blocks in order to interact with the client and initiate the logic for the operations he performed. There was also used `<assign>` blocks to convert one complex types to another in order to `<invoke>` simple web services, that have been used



in intermediate steps. There was also used `<flow>` mechanism to run in parallel during booking and cancelling itinerary to speed up the process (instead of sequential operations). For exception handling and tractionality for core operations, there were used `<scope>` and `<compensationhandler>` as well as `<faulthandler>` in each scope. Details of the BPEL process are being discussed below in further sections. The BPEL process starts when a client requests to create an itinerary using *createItineraryOperation* which expects a user id as input and returns itinerary id as output. The uniqueness of an itinerary was enforced by the BPEL process itself using correlation sets. Whenever a user requests to create an itinerary, we create a GUID as itinerary id and correlate the whole business process using that id meaning that each subsequent operation calls will contain the itinerary id in their input in order to figure out which BPEL process they are on.

After itinerary is created, we step into *planning phase* which contains two different scopes namely *booking scope* and *waiting scope*. In the booking scope, client can get possible flight/s/hotels according to his/her needs and adds them into his/her itinerary. The scope terminates when the user calls *bookItineraryOperation* with a credit card info. After the itinerary is booked, we enter into waiting scope where the process waits until the first date of the booked flight/hotel in the itinerary and terminates the whole process when the time exceeds.

In the whole planning phase, the client can call *getItineraryOperation* in order to see the current status of the itinerary or the client can call *cancelItineraryOperation* that will terminate the whole process if the itinerary is not booked, or it will wait until the first date of the flight/hotel in the itinerary before it terminates the whole process.

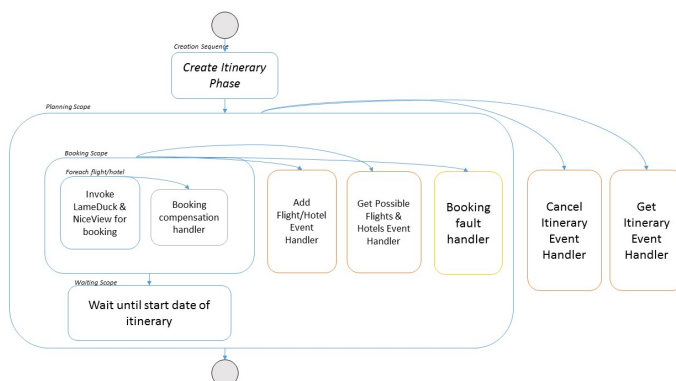


Figure 5: Abstract representation of TravelGood BPEL implementation.

## Create Sequence

BPEL uses a sequence approach to web services with a visual interface to go along with it. To start our BPEL process, the client sends a request to the *createItineraryOperation()* taking a string representing the ID of the client as input and returning another string representing the ID of the created itinerary as a result. Using this ID, the correlation set of the BPEL process is initiated and all the subsequent operation calls will contain the itinerary ID as an input. In this way the BPEL engine is able to understand the current execution order of the process and can easily distinguish between different processes. Before returning the ID of the created

itinerary, empty lists are assigned to *flightInfoArray* and *hotelInfoArray* variables, part of the *itineraryInfo* variable in the create sequence and the value of the current booked variable is set to false. After the unique identifier is returned to the client, the creation sequence is finished and the process enters the planning scope.

## Planning Scope

At any time in the planning scope the customer can cancel the itinerary and get the current state of itinerary.

### *Cancel Itinerary Event Handler*

According to the requirements, clients have the possibility to cancel planning phase as well as booking. A *cancelItineraryOperation* expects the itinerary ID and a credit card information as an input and returns true or false, depending on if the cancellation is successful or not. At the beginning of a scope associated with *CancelItinerary* event, a variable called *cancelItineraryOutput* is defined with a boolean true value assigned to it. This variable is used to determine if the cancellation was successful or not at the end of the scope.

There are two possible actions - client cancels either during planning or booking - so there are two If activities *IfBooked* and *IfNotBooked*. To determine which activity should be used, the process checks value of the variable *booked* (true or false). Variable *booked* is set to false when the process starts and is set to true after booking phase finished successfully.

#### *IfBooked activity*

The implementation of this activity is based on two parallel flows, one for cancelling flights and the second one for cancelling hotels. Both tasks are independent since they use two different external web services. By using two separate *ForEach* activities, the process goes through all of the items in *flightInfoArray* and *hotelsInfoArray* respectively, it is through all of the items in the itinerary.

The flow in each of *ForEach* activity is as follows: take current element from an array and assign it to the *currentFlightInfo/ currentHotelInfo* variable, assign necessary values to the input variable for the external web service (booking number and credit card information), invoke *LameDuck/NiceView* web service which are added as PartnerLinks, make a logical AND operation of variables *cancelItineraryOutput* and value returned from the web service and assign the result back to *cancelItineraryOutput*, if the cancellation was successful then change the status of the current flight/hotel in an itinerary to CANCELLED.

Since cancelling may fail, there is a FaultHandler associated with a *CancelFlightScope/ CancelHotelScope*. It is a *catchAll* element which means that it catches all faults which occur in this scope. It handles exception which may be returned by the invoke activity and it assigns a logical false to *cancelItineraryOutput* variable. It ensures that a logical false is returned whenever an invocation of cancel operation fails.

After *IfBooked* activity is finished, it replies with the value in *cancelItineraryOutput* which is either true or false. True means that all of the booked items has been cancelled and false means that at least one of the booking has not been cancelled.

### IfNotBooked activity

In case a client cancels a planning phase, a BPEL process replies with boolean true immediately. After that the BPEL process ends.

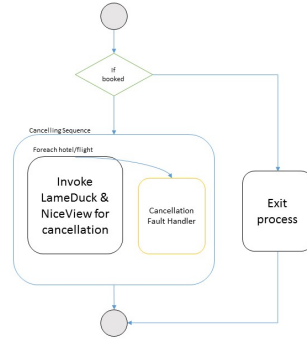


Figure 6: Abstract representation of TravelGood cancel itinerary event handler.

### ***Get Itinerary Event Handler***

In the planning scope, the client is able to get the status of the itinerary anytime he wants. The event *getItinerary* has an underlying scope in the event handler, named *GetItineraryEventScope*, which assigns the itinerary info that we keep in the BPEL process to the *getItineraryOutput* which contains *flightInfoArray* and *hotelInfoArray*.

### **Booking Scope**

At any time in the booking scope, the customer can get a list of possible flights and hotels for a destination and time as well as add flights and hotels to an itinerary. Hence, the events handled in this phase are:

- getting list of possible flights and hotels (*getPossibleFlights/getPossibleHotels*)
- adding specific flight or hotel to itinerary (*addFlight/addHotel*)

### ***Get Possible Flights & Hotels Event Handler***

To get a list of possible flights one has to call the *getFlight* operation from *LameDuck* service. The *getFlight* operation expects itinerary ID, departure airport, destination airport and departure date as input and returns an array (*flightInfoArray*) with possible flights as output. The event *getPossibleFlights* has an underlying scope in the event handler named *GetFlightsEventScope*. The scope performs the following actions to return a list of possible flights to the client:

First it assigns the input for the *getPossibleFlights* to the input for the *getFlightsOperation* from the external service *LameDuck*. *LameDuck* is added as PartnerLink in BPEL. Secondly the *getFlights* operation from the *LameDuck* service is invoked. The list of flights from the *LameDuck* service is then assigned to the *getPossibleFlights* output in the *TravelGood* web

service. Hence, the output is a list of flights that satisfies the requirements specified by the client.

To get a list of possible hotels one has to call the *gethotel* operation from *NiceView* service. The *getHotel* operation expects itinerary ID, arrival date, departure date and city as input and returns an array (*hotelInfoArray*) as output. The event *getPossibleHotels* has an underlying scope in the event handler named *GetHotelsEventScope*. The scope performs the same actions as the scope for *GetPossibleFlights*, but in this case it assigns to the input for the *getHotel* operation from the external service *NiceView* and it assigns the response to the *getPossibleHotels*. As *LameDuck*, *NiceView* is added as a *PartnerLink* in BPEL.

### **Add Flight/Hotel Event Handler**

The *addFlight* event expects the id of the itinerary and a flight info that is going to be added to itinerary and it returns *FlightInfoArray* and *HotelInfoArray* as output that shows the current itinerary. The scope for the event *addFlight* is named *AddFlightEventScope*. The scope performs the following actions:

It finds the number of flights already added to the itinerary and assigns the value to the *numberOfFlights* variable. The flight that should be added to the itinerary is then appended to the end of the list of flights in the itinerary (*flightInfoArray*), by using the *numberOfFlights* variable. The scope then returns the updates *flightInfoArray* as well as *hotelInfoArray*.

The *addHotel* event expects itinerary ID and a *hotelInfoType* as input and it returns *FlightInfoArray* and *HotelInfoArray* as output. The scope for the event *addHotel* is named *AddHotelEventScope*. The scope performs the same actions as the scope for *addFlight*, but instead it finds the number of hotels already added to the itinerary and assigns the value to the *numberOfHotels* variable. The hotel that should be added to the itinerary is then appended to the end of the list of hotels in the itinerary (*hotelInfoArray*).

### **Booking operation**

The BPEL process enters the booking phase when the *bookItinerary()* operation is called by the client. The sequence starts by retrieving the information received from the client and required for the booking operation: an *itinerary* and *credit card information*. After this, the following flow of activities in *BookingFlow* is triggered:

- the process iterates through both the flights and the hotels arrays and enters a new scope for every iteration (*BookFlightScope* / *BookHotelScope*),
- in each of these inner scopes the following sequence of activities is executed:
  - assign the values of the flight/hotel object from the received itinerary to the *currentFlight*/*currentHotel* variable,
  - assign the values of the *currentFlight*/*currentHotel* variable to an input variable that will be used to invoke the *bookFlight()*/*bookHotel()* operations from *LameDuck*/*NiceView* external services (these are added as *PartnerLinks* in BPEL and can be called at any time throughout the process),
  - invoke the booking operation from the external web services,
  - assign the result of the invoke activity to the result of the *bookItinerary()* operation,

- change the status of the flight/hotel object in the current iteration to "CONFIRMED".

Both inner scopes discussed above can terminate with a fault since both *bookFlight()* and *bookHotel()* operations of *LameDuck* and *NiceView* services return a fault if errors occurred during booking; in BPEL, these faults are handled using *FaultHandlers*. We set the fault handler in the booking scope so that whenever a fault is thrown in a booking invocation, we will catch it in the scope and by using *Compensate* activity, we start to compensate all the successfully finished subscopes.

In addition, whenever a fault occurs, the booking operation needs to compensate for the flights/hotels that have already been booked and revert everything (refund, if necessary). In BPEL, compensation is achieved using *CompensationHandlers*, therefore, each of the inner scopes have their own compensation handler executing the following sequence of activities:

- assign the values of the *currentFlight/currentHotel* variable to an input variable that will be used to invoke the *cancelFlight()/cancelHotel()* operations from *LameDuck/NiceView* external services,
- invoke the cancelling operation from the external web services. Since cancellation logic for flights includes only refunding half of the price that is paid by booking, when we are invoking cancelling operation in compensation handler, we call with the double amount of the booking price to ensure that it is refunded the whole amount,
- change the status of the flight/hotel object in the current iteration to "CANCELLED".

After the sequences of activities from the inner scopes have finished successfully, the value *true* will be assigned to the global variable *booked* (variable created when the process starts) and a reply is sent back to the client.

The process is however not terminated yet, it enters a waiting state according to the logic described in the **Waiting Scope** section.

### Waiting Scope

The wait logic is used after the an itinerary has been booked, to keep the instance alive until the earliest itinerary item has passed. To do this we first loop through the *flightInfoArray* and the *hotelInfoArray* and determine the earliest item in the itinerary. Concurrently we start a wait until this time has passed and when it has passed then the BPEL process will end.

## 3.d RESTful Web Service

*TravelGood* RESTful service implements three types of resources:

- ItineraryResource
- FlightInfoResource
- HotelInfoResource

The reason for dividing resources into three types was that there are two types of resources provided by the external web services (flights and hotels) and one type of resource provided by the *TravelGood* service (itinerary). The API reference for *TravelGood* service is presented in table 1.

URI Path	Resource Class	HTTP Methods
/users/userid/itinerary/itineraryid	ItineraryResource	PUT, GET
/users/userid/itinerary/itineraryid/book	ItineraryResource	POST
/users/userid/itinerary/itineraryid/cancel	ItineraryResource	POST
/users/userid/itinerary/itineraryid/flights	FlightInfoResource	GET
/users/userid/itinerary/itineraryid/flights/add	FlightInfoResource	POST
/users/userid/itinerary/itineraryid/hotels	HotelInfoResource	GET
/users/userid/itinerary/itineraryid/hotels/add	HotelInfoResource	POST

Table 1: API reference for *TravelGood* RESTful service

## Create phase

In BPEL implementation, itinerary is associated with a process. In RESTful implementation all itineraries are stored in the *ItineraryPool* class. There, a static list represented as a HashMap contains all of the created itineraries. The key for each itinerary in the list is stored as *userid-itineraryid* and in this way the uniqueness of an itinerary is ensured.

The process starts with creating an itinerary. A client initiates the process by sending a PUT request with user ID and itinerary ID as parameters. As opposed to BPEL, in RESTful implementation the id of an itinerary is generated on the client's side. This means that the service does not create and return an itinerary id, instead, it just takes the value which has been sent in a request. We assume that every user has a unique id assigned and uses this id when sending requests to the web service. After the itinerary is created and added to the list of itineraries, a response with the information "itinerary successfully created" is returned to the client.

## Planning phase

### *Exploring available flights and hotels*

The planning phase starts with getting a list of available flights or hotels. A client sends a GET request to the web service with query parameters specified:

- date, startAirport, endAirport for getting list of flights
- city, arrivalDate and departureDate for getting list of hotels

If the itinerary for given path parameters (*userid*, *itineraryid*) does not exist then the method returns a response object with status NOT\_FOUND. If the itinerary is already booked or cancelled then the method returns a response object with status NOT\_ACCEPTABLE. Otherwise, a response object with the list of available flights/hotels is returned to the client. That

is the way we ensure that the client can see the list of available flights/hotels only if he has a valid itinerary which is not booked or cancelled.

### ***Adding flights and hotels to itinerary***

In order to add a flight or a hotel to the itinerary, client has to send either a *flightInfo* object or a *hotelInfo* object in the body of a POST request (see table 1).

Same as for getting the list of flights/hotels, if the itinerary for the given path parameters does not exist then the method returns a response object with status NOT\_FOUND. If the itinerary is already booked or cancelled then the method returns a response object with status NOT\_ACCEPTABLE. Otherwise, a response object with the information that the flight/hotel has been successfully added to the itinerary is returned to the client.

## **Booking phase**

Another operation in the Itinerary Resource class is the *bookItinerary()*. A client sends a POST request with user ID, itinerary ID as path parameters and credit card information as request entity. An itinerary can only get booked if the status of the itinerary is unconfirmed. Hence, when receiving the POST request from a client, the booking operation starts by checking if the itinerary does not exist or status of the itinerary is already booked (CONFIRMED) or is already cancelled (CANCELLED). If that is the case, then a response with the information "itinerary not found", "itinerary booked already" or "itinerary cancelled already" is returned to the client. If the status is unconfirmed (UNCONFIRMED), it goes through every flight and hotel added to the itinerary and books them. If for some reason a booking fails, it catches a fault and starts cancelling everything that has been booked so far. The response with the information "Not all bookings were booked" is returned to the client. If all bookings succeed, it sets the status of the itinerary to CONFIRMED and a response with the information "itinerary successfully booked" is returned to the client.

## **Cancellation phase**

In order to cancel an itinerary, the user sends a POST request to the *TravelGood* REST service with the user ID, itinerary ID and credit card information as parameters. Both the user and the itinerary ID's are sent as parameters in the path of the request whereas the credit card information is sent as an object in the body of the request.

After a POST request has been received, the *getItinerary()* operation will look for the itinerary resource in the *ItineraryPool*. If the requested resource is not found then a response with status NOT\_FOUND is returned to the user, otherwise the service will continue with the cancel operation based on the status of the itinerary:

- If itinerary is CONFIRMED
  - The service will iterate through all the flights and hotels of the booking and try to cancel them by calling the external *LameDuck/NiceView* services.
  - If any faults happen during the cancellation of one booking (either flight or hotel) the operation still continues and in the end the user will get a response with status



"Not all bookings were cancelled" and a link for accessing the itinerary and check exactly the status of each booking

- If cancellation was completed successfully, the user will get back a response with status "itinerary successfully cancelled" and a link for accessing the itinerary and check exactly the status of each booking
- If itinerary is CANCELLED
  - The user will receive a response with the message "itinerary cancelled already"
- If itinerary is UNCONFIRMED
  - The itinerary will be deleted from the *ItineraryPool* and the user will get a response with the status "itinerary terminated" and a link to create a new itinerary

## Get Status of Itinerary

During every state of an itinerary (UNCONFIRMED, CONFIRMED, CANCELLED), the operation *getItinerary()* can be called. This operation is possible by making a GET request on the *ItineraryResource* with a user ID (*userid*) and itinerary ID (*itineraryid*) as parameters. If no itinerary is found in the *ItineraryPool* a response with the status NOT\_FOUND is sent. If an itinerary is found it will check that all the flights/hotels are still valid (their start date has not passed). If either a flight or a hotel has a date in the past a response with status BAD\_REQUEST and message "itinerary terminated" otherwise the itinerary resource will be sent to the client along with all its data and relevant links.

## Next steps added as links

During the entire business process the customer is able to see what operations/calls can be done next. A set of links with the possible next steps (URI) is returned to the client. What links are returned is shown in table 2.

URI Path	create-Itinerary	getHotels/ getFlights	addHotel	addFlight	book	cancel	get-Itinerary
... (PUT)		x					x
... (GET)		x			x	x	
.../book		x*				x	x
.../cancel	x**						x
.../flights		x		x	x	x	x
.../flights/add		x			x	x	x
.../hotels		x	x		x	x	x
.../hotels/add		x			x	x	x

Table 2: Links returned to client for *TravelGood* RESTful service

\*Only if booking fails, \*\*Only if cancelled in planning phase



The different URI paths shown in the first column are all preceded by `/users/userid/itinerary/itineraryid` and all possible operations(steps) from the business process are shown in the first row. The x's indicates whether the step is added as a link after the URI has been called.

For example, the next possible steps after calling `/users/userid/itinerary/itineraryid/flights` are: getting list of hotels or flights, add flight to itinerary, book itinerary, cancel itinerary and get itinerary. Also, after booking, the customer can cancel the itinerary, get the itinerary and if the booking fails, get list of hotels and flights.

In the whole REST implementation, we decided to use xml to present our resources to external clients. Our motivation was to keep the clients informed about the types of the return values. This might introduce longer messages than JSON representation but at least more clear and concise representation of the entities they represent. We also thought that if the clients start using browser based applications, we would think about migrating to JSON based representation since Javascript has built-in support for JSON objects.

## 4 Web Service Discovery

The Web-Service Inspection Language (WSIL) files are a standard that show where one can find the links to the web services' WSDL files. It is meant as an entry much like a phone book but then for web services. These WSIL files describe the current web services that you are accessing and also contains links to other web services this services uses to fulfil its tasks. The WSIL specification is based on an XML format developed by IBM and Microsoft ??.

The *TravelGood* WSIL file is locally managed on same web server as the BPEL application but has to be deployed separately. The document describes where the WSDL file for the BPEL application can be found in the service part of this file, as a link both the *LameDuck* and the *NiceView* services are also described with their respected WSIL files.

The WSIL files for both *NiceView* and *LameDuck* are very similar and also locally managed. These files are deployed along with the web services themselves and describe in the service part what the web services are and where their WSDL files can be found. Moreover, they also have a link to the *FastMoney* service since they use this to validate their credit card information and do payments.

## 5 Comparison RESTfull and SOAP/BPEL Web Services

### Ease of implementation

After implementing the same business logic using SOAP based web services and RESTful web services, we can say that SOAP based web services has a steep learning curve. We had to spend significant amount of time in order to build the process from scratch. Lack of tools for BPEL development is also affected our implementation speed. We have used OpenESB during the implementation and it has a lot of bugs that we had to deal with it. One other

point is that debugging a BPEL process is not easy. We have created a simple debugger for simple flight web service in order to track what is going on behind the scenes and which third party web services are called during process runs. But at the end, we managed to fulfill the requirements and able to pass all the required tests.

The compensation handling and fault handling in BPEL engines ease the pain of implementing transactionality and error handling from scratch. Although it was hard to adapt to their usage and include them into our solution.

In RESTful web service implementation, the main challenge was to handle resource/url mapping in a nice and elegant way. So whenever the user reads the url, it should be clear for him to understand what it does/which resource it maps to. JAX-RS was particularly easy to use and since we are experienced with java programming language, the implementation stage went smooth and we didn't hit a lot of issues. At the end, we can conclude that it was easier to use RESTful web services for this business process than to use SOAP based web-services.

### **Ease of understanding the implementation**

BPEL has a big advantage such that you can create a business process by simply dragging/-dropping blocks into one another. So you can see how process starts, iterates and terminates by just looking at the BPEL file. The only way to understand the implementation of RESTful services is to read the code behind. That makes it more time-consuming and harder to track the process.

### **Ease of changing the business process later**

BPEL also has advantage on this point. Since it is easier to build the process using activities and blocks in the designer editor, it is also easy to change the process later without breaking any other parts of the system. We have actually seen this in action when we needed to modify our BPEL implementation couple of time to adjust to our needs, it worked smoothly. RESTful services are harder to change the process, since more coding and testing needs to be involved in order to adjust to current needs.

### **Scalability**

BPEL engines are designed to be scalable and run thousands of business processes concurrently. In order to make RESTful web services scalable, there should be design decisions such as if the resources in the web service are read-intensive or write-intensive. Then the underlying database/cache strategy can be chosen in order to reflect the needs. In short, BPEL engines by default are more scalable than RESTful services.

## 6 Advanced Web Service Technology

### WS-Addressing

In SOAP based message exchange, messages don't contain the sender information, ws-addressing protocol is created to overcome this issue. Moreover, you can override the replies and send to another endpoint than the caller, you can create message relationships to group messages together.

For our case, we could have used ws-addressing protocol to group the requests coming from TravelGood to LameDuck/NiceView. By grouping, these two web services could extract the knowledge of which bookings are added/queried for an itinerary, which can then be used to gain knowledge of user interests. Moreover, we can use ws-addressing to see which clients are using TravelGood web services so that we can track the usage and get statistics from there. Another benefit of ws-addressing is the ability to reply to messages/faults to some other endpoints, for instance when a fault occurs during booking. So, when the fault occurs it should be returned back to TravelGood but if the booking succeeds, it should return the response directly to the user-endpoint.

### WS-ReliableMessaging

WS-ReliableMessaging ensures that messages are retransmitted until the receipt is acknowledged. This is an important property for our web service implementations.

The messaging between TravelGood and NiceView/LameDuck, as well as the messaging between NiceView/LameDuck and FastMoney, should be reliable since we are doing transactions that involve payment and bookings. The client should be ensured that the payment messages and responses are received from both the client and server side, and the end-user shouldn't end up in the situation of paying double (or more) for his itinerary.

### WS-Security

When SOAP messages are using HTTP, then they can be easily read and changed by a third party. HTTPS itself, is not enough to secure a SOAP message since the receiver still cannot understand if a message is changed or not, even though a third party cannot read the message, he can still change it.

WS-Security becomes handy for our business process, since we are operating on sensitive data (such as credit card info) and we do not want any third parties to read/change that data. In particular we do not want any third party to read any data that is being transferred between the components in our business process.

### WS-Policy

WS-Policy define rules that web services can impose for allowing access to their operations. These rules can be related either to the capabilities of the web service, what operations can

the web service provide, or to the requirements of the web service, what does the web service need in order to execute those operations. Policies can be applied to different levels of the WSDL depending on whether the rules refer only to an input/output/fault message, to an operation, to certain endpoints (port/port type/binding) or to the entire service.

A good example of a policy valuable for our business process would be a UsernameToken policy on all TravelGood operations. This type of restriction ensures that only authenticated users can create itineraries and operate on them (add flights/hotels, book/cancel them). In addition, imposing a UsernameToken with a hashed password as input to all operations will allow TravelGood to identify which itineraries belong to a specific user so that users are authorized to make changes only to their own itineraries. ??

## 7 Conclusion

The opportunity to work on the *TravelGood* project provided us with an experience and understanding of the web services implementation processes. To sum up the report, we will discuss issues we encountered during the implementation as well as share some thoughts on what we actually did to deliver a good quality project and what we learnt from implementing all four web services.

While working on the *TravelGood* project we identified some obstacles which extended the time needed to successfully implement the required web services. First of all, the control version system we used (Git) does not deal well with BPEL files hence we usually had to merge changes manually which was definitely time consuming. Because of that, it was hard to divide tasks among all of the group members while working on BPEL since it was better to work on one version at a time to avoid version conflicts. In terms of group work, RESTful implementation process was definitely a lot easier to split up and implement in parallel.

Although BPEL Editor interface in OpenESB is easy to use, it gets much harder and less user-friendly once the process becomes more complex. Arrows which connect elements make the process diagram hard to read. We came up with the conclusion that, in the case of this project, the interface started to get very confusing at some point. Moreover, in the beginning of the project we did not spend enough time on designing the data structures (complex types and elements) required by the SOAP services. Due to that, we had to refactor the WSDL files of all of the web services several times which resulted in an additional work regarding merging of the versions. This sometimes also caused code redundancy. However, we already took this experience as a starting learning point for the REST implementation when we thoroughly discussed the necessary resources before starting the implementation.

We are also aware of what we could have done better since now we already have an experience in working with web services:

- give the designing process a lot of thought, in particular pay attention to identifying needed data structures in order to avoid code redundancy,
- simplify the BPEL process by making the best use of elements available in the design interface of the OpenESB environment,

- become acquainted with required tests as soon as possible, preferably already at the designing stage,
- define all data types in one document from the start (e.g. and .xsd file) so that common complex data types can be easily referenced and reused among web services

On the other hand, we did our best to produce good quality code. For instance, we used the builder pattern for generating the flight and hotel objects, we created utility classes to accommodate static reusable variables (see *StringUtils* in *TravelGoodREST*), we provided special methods for commonly used operations (see *TestUtils* in *TravelGoodTest* project/bpel package) and, in general, tried to avoid code redundancy.

In addition, we touched most of the elements and activities available in the BPEL framework in order to provide the whole functionality of our business process (e.g. flow activities, compensation handlers, fault handlers).

Last but not least, we ensured that all our tests are properly written such that assertions are not wrong (e.g. comparing two itinerary objects instead of the expected itinerary status) and the expected result is received.

## 8 Who did what