Examination project

TravelGood Web service

by

Group 08

Gert Qin Hansen	**s093059**
Caspar Bak Ahrensberg	**s093258**
Thomas Hjorth Hansen	**s093272**
Jacob Bogelund Hansen	**s093277**
Andreas Foldager	**s093285**
Jasmina Pelivani	**s093289**

Technical University of Denmark
Department of Informatics and Mathematical Modelling
02267 Software Development of Web Services E12
Hubert Baumeister
3rd December 2012

Contents

Chapter 1

Introduction

This report deals with the implementation and design choices of a web service representing a travel agency. We will start off by giving a short introduction to the basic theory of web services and the motivation for using the technologies used to implement this project.

Since our web service makes use of other services, we will have a chapter on the coordination protocol, which describes the coordination between the client and TravelGood in order to make an itinerary for the travel agency. When implementing the web service, this protocol will be used for both RESTful and BPEL/SOAP implementation. Afterwards, the design decisions behind these implementations will be covered, and compared to each other.

Finally we will describe some of the advanced technologies available for web service development and the important role they would play for this project, if it was supposed to be an actual product for a customer.

1.1 Web Service Technologies

Service Orientation Architecture

Service orientation architecture (SOA) is a set of principles for organizing and developing software. Services are loosely coupled and are independent of each other. They each represent a unique functionality that is independent of the context they are used in. This means that they can be reused for many different applications.
Web services are based on this architecture. They provide services through the internet which always should be accessible for applications to use. For instance there might be a web service for verifying credit cards which all payment applications can use, as seen in our example solution.

Web service description

In order to understand a web service and discover all its possibilities, it needs to be described in some way. For BEPL, WSDL (Web Service Description Language) files are used. These files contain among other things information about the operations and how they can be called, including the parameters they use. For RESTful, WADL (Web Application Description Language) files are used. The files contain information on how resources are calculated and provided by services.

Web service discovery

Many software products today rely on web services. They can be provided by partners, customers or other companies. Web service discovery is designed to help find a suitable web service for a certain task and it increases the collaboration between services (for example a

bank providing credit information for an airline company). Basically web service discovery can be done in two ways; centralized using UDDI (Universal Description Discovery and Integration) or decentralized using WSIL (Web Service Description Language).
UDDI makes use of a centralized provider with information about the provided web services. These are about; the company, how to contact it, products, services, service behaviour and other technical information. Choosing the right web service can be done manually (usually at design time) or automatically based on the information at both design and run time. The service providers therefore heavily relies on the credibility of the central register and how often this is updated.
WSIL makes use of decentralized (local) information to describe web services. A WSIL document contains information specific to one or more web services and can link to other WSIL and UDDI registries used. The WSIL documents are based on XML and are locally managed by the provider of the web service itself.[1] [2]

Web service composition and coordination

The concept of web service composition is essential for programming languages for business processes. The language BPEL used in this project is just one example of this relationship. Services are called to achieve a business process requiring external resources. These processes can complete instantly or take several days. They are composed of services that are offered by a combination of external providers and internal providers. A large benefit when developing web services as business processes is the easy overview the graphical representation provides when developing in BEPL. This allows for an easier understanding of the process itself for people with limited experience with web services and software development in general.

Web service coordination is used to plan activities, that makes use of several actions from various web services. This coordination is a necessary agreement made to make sure that the actions of the web services together succeeds the overall activity.[3]

SOAP

Simple Object Access Protocol (SOAP) is an XML based protocol used by web services, for exchanging messages. These messages are transported by a HTTP or SMTP protocol. The benefits of using SOAP to exchange messages with web services are that among other things that the messages are easy to generate because they are based on XML. This also makes it easy to use and provide web services from any programming language.

RESTful services

RESTful services are focused more on resource management compared to SOAP that focus on the operations one can do on a service. The resources that the RESTful service represents can be accessed with the use of URIs, and these can be acted upon by use of HTTP verbs. As a result of using HTTP, the expected response from the service is defined by HTTP statuses, describing the outcome of the request on the server. By using HTTP verbs one ensures that RESTful service have a uniform interface in which different MIME-types can be transferred through a request. This allows for an open response, which for instance could be XML, JSON, plain text or another data format.

One aspect that greatly varies from SOAP service is that a RESTful service is stateless, meaning no data of a session is stored on the server. As a result of this, the client must provide all information needed. Due to the service being stateless, business processes must be handled in another way than how BPEL does it. The stateless approach is handled by providing a link to the caller where the resources will be placed once the service has finished calculating. This ensures that all clients follow a specific procedure to get their information.

Chapter 2

Coordination Protocol

The coordination protocol seen on fig. 2 follows the process of fulfilling an itinerary when using our system, i.e. from creation until cancellation or fulfilment. So if we take it from the initial step, the client will have to create the initial itinerary, and when created he will be able to update the itinerary with flights and hotels.

When the itinerary that the client created, is in the **Planning phase** state, he may get flights or hotels corresponding to his wishes and add any of these to the current itinerary. Otherwise he can simply cancel the itinerary, which in turn will free it from the system, meaning he cannot retrieve it again. Last but not least he will also be able to book the whole itinerary, which will go the **Booking phase** state.

The **Booking phase** state is a pseudo-state, meaning that the client cannot interact with the itinerary when he is located in this state. It will simply be used to verify if all flights and hotels added to the itinerary can be booked or not. This also means that two things can happen when located in this state. Either the booking succeeds, meaning we end in the **Booking Successful** state, or it fails allowing for another branch, where a successful cancelling will return the client to the **Planning phase** state and a failed cancel will go to the **Cancel failed** state.

In case it ends in the **Booking Successful** state, the client can try and cancel it, before the booking will be complete (Final state), in which no action is possible for the client.

If a booking fails, it will as fast as possible try to cancel the itinerary, in case some flights or hotels got booked.

If the cancellation succeeds, it will simply return to the **Planning phase** state. However in case the cancellation fail, we will end up in the **Cancel failed** state in which the client can only wait for the itinerary to be deleted one day before departure.

At any time during the process, the client may retrieve his itinerary in case it is located in a state that allows user-interaction, i.e. in the states: **Planning phase**, **Booking successful** and **Cancel Failed**. The pseudo-state, **Booking phase** does not support this, as it is not be visible to the client.

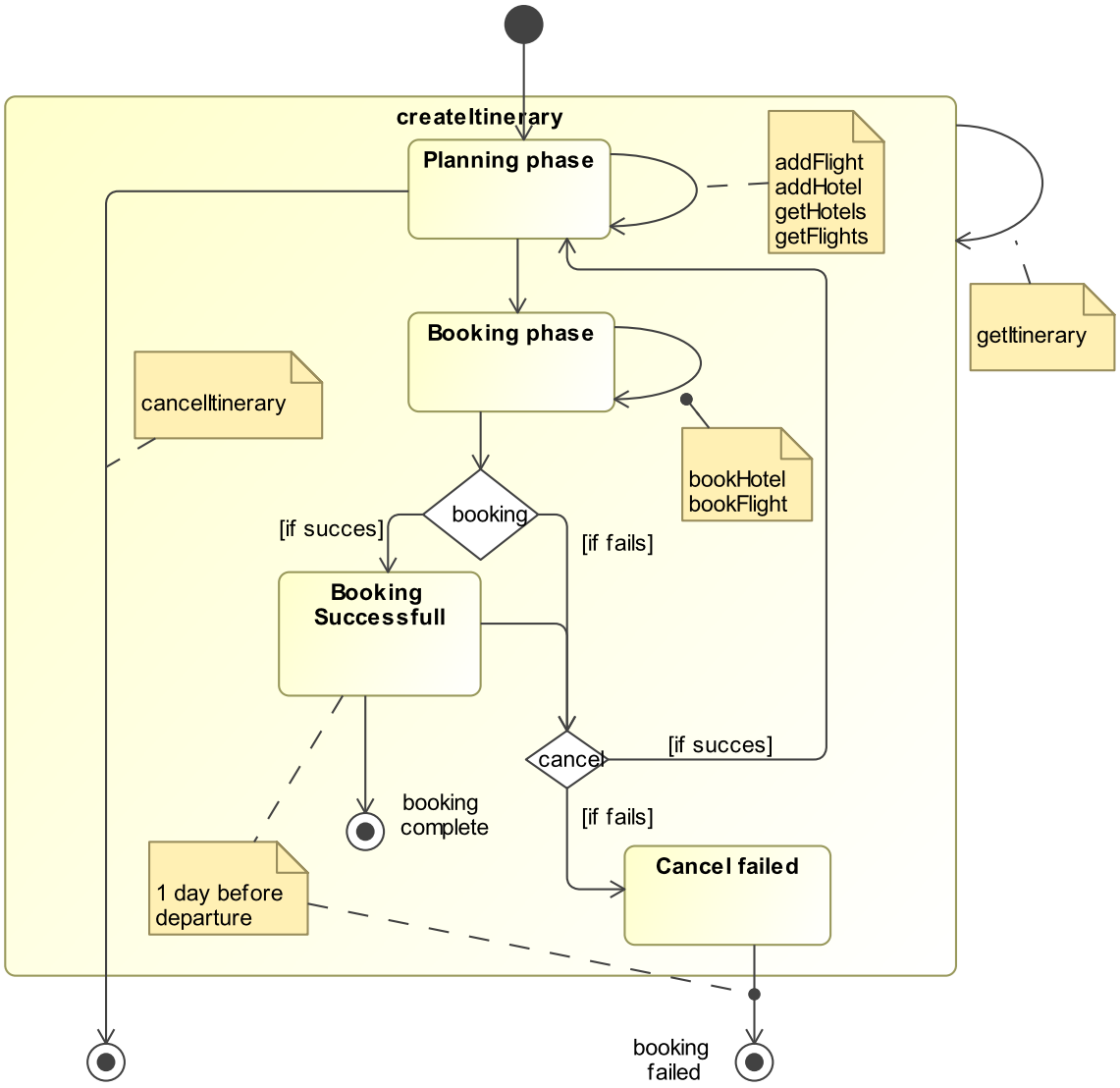

Figure 2.1: The coordination protocol for the TravelAgency interpreted from the given description

Chapter 3

Web Service Implementations

3.1 Data Structures

This section will contain a description of the choices that has been made for the two services, i.e. SOAP/BPEL and RESTfull, regarding data structures and how the internal classes relate to each other.

Common data types

One issue that should be taken care of, was how to keep track of the status of every booking made, for both flights and hotels. And in order to reduce the amount of traffic on LameDuck and NiceView a decision of storing it locally was made, so the travel agency would not have to contact the external services every time a request for a booking arrived. This was solved by adding a wrapper around both hotels and flights and their respective status that could be either **confirmed**, **unconfirmed** or **cancelled**. This can be seen in fig. 3.1 and is used in both RESTful and BPEL.

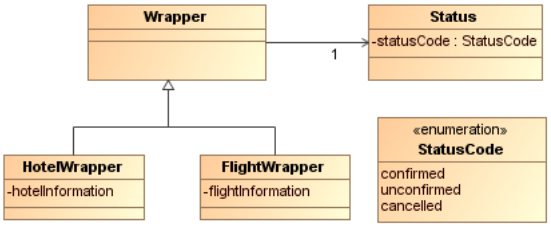

Figure 3.1: The classes that encapsulate a flight or hotel with a booking status for the RESTful solution

SOAP/BPEL

Since we are using information from external web services, we also use their complex types and therefore we need to import them. Besides these and the wrappers described in the section about common data types, we have added two more complex types, which represent a list of wrappers, *hotelStatus* and *flightStatus*, respectively. Because a customer can add multiple flights and hotels to his itinerary, we need to be able to store an unbounded amount of these wrappers and therefore we have made the lists.

RESTful

Here we provide a glimpse of the internal classes that holds the service together and how they are structured, as seen in fig. 3.2. At the top we will have a customer, containing a collection of itineraries, i.e. the bookings, as well as his credit card information. The collection of itineraries will be contained within a HashMap, as this gives a constant lookup time. And due to every itinerary having a unique identifier, within the customer owning the itinerary, no conflicts should arise. If we look into the itinerary, we can see that it will contain two collections, one for the hotels and another for the flights. These will be represented by a simple list, as we most likely will have to return the whole itinerary and not a single flight or hotel, and run through all bookings.

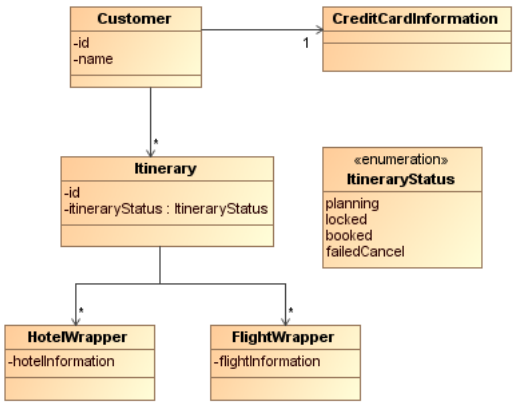

Figure 3.2: The information needed in order to make itineraries with the RESTful solution

By having chosen these data structures we are ensured that our solution is somewhat optimized for the tasks that the RESTful service will have to comply with.

3.2 Common design decisions for SOAP implementations

We have chosen the RPC/Literal binding-style for the operations, for all SOAP implementations, as this is a WS-I (Web Service Interoperability) standard binding with a simple structure. The literal encoding provides the simplest layout for the response messages and can thus be directly interpreted by the process without any conversions. The RPC style was chosen as we did not have the need for customized operation elements, which documents style allows, to execute the operations.

3.3 NiceView

The NiceView service provides operations for finding bookings and hotels. The service consists of three operations, getHotels, bookHotel, and cancelHotel.

Data Types

For the WSDL-file describing this service, we have four complex types: addressType, bookingType, hotelType and returnHotelList. To see a overview of the types, see figure 3.3.

addressType Describes an address.

hotelType Used to describe hotels, and consists of fields describing the name of the hotel, the price for a single night, whether credit card information should be validated before booking, and an addressType (type for describing the address of a hotel).

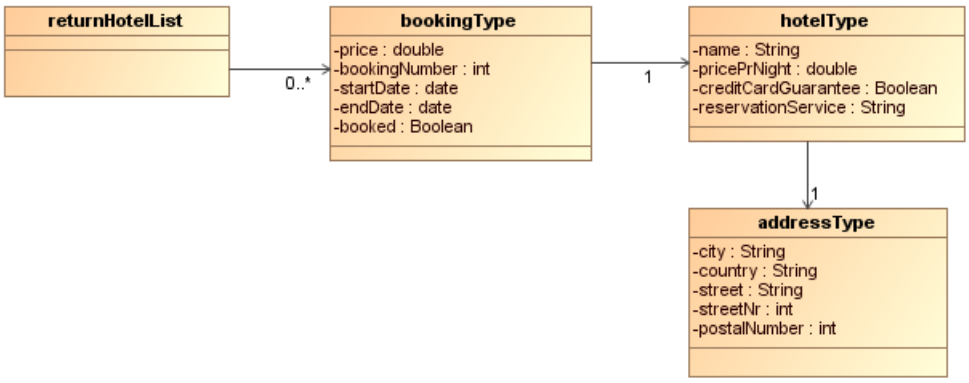

Figure 3.3: The data types defined for the NiceView service.

bookingType Describes the possibilities for a booking. This type contains fields describing the booking number, the hotel, the full price of the stay, the start- and the endDate. A boolean flag tells whether the booking has been completed.

returnHotelList A wrapper enabling the finalized service to return a list of booking-types.

Furthermore the service uses the data types of the bank service, *FastMoney*. Here we especially mean the **CreditCardInfoType** representing the credit card information of a client. This type is used when a client books a hotel.

Operations

getHotels A client specifies a city in which, he want to search for hotels, and furthermore the client inputs the duration of the stay by an arrival date and a departure date. The operation returns returnHotelList containing all available hotels. For each hotel found according to the input the service creates an object of **bookingType** making it possible for the client to book the hotel. Initially the boolean flag telling whether the booking is active, is set to false, meaning that the client has not booked the hotel.

bookHotel Each **bookingType** returned holds a unique booking number used for identification. The client can use the booking numbers received from *getHotels* as input for the *bookHotel* operation in order to book the stay at the hotel. Furthermore this operation also take credit card information as type of **CreditCardInfoType** as input, and the operation returns a boolean saying whether the booking was successful. If the booking is not successful, a **BookingFault** is thrown.

cancelHotel This operation takes as input a booking number and tries to cancel the booking. A boolean telling whether the cancellation was successful is returned. If the cancellation was not successful, a **CancelFault** is thrown.

This way the client is able to first search for hotels, and then based on the output of the *getHotels* operation, he can book and, if they are booked, cancel hotels.

Improvements

Looking at the **hotelType** describing the hotels, we see that we have a variable for the price per night. We only have one variable for this, meaning each hotel can only have one price. In reality hotels can have multiple price classes, and the price can also vary according to seasons, however this is left out for simplicities sake.

We also see, when a client uses the *getHotels* operation, we create multiple objects of type **bookingType**, which is stored globally, so the client can book the hotels using the *bookHotel* operation. This is not optimal, as we should make sure to delete some of these objects, when the potential stay at the hotel is completed. First of all it might not be necessary to keep old information of old bookings. Second of all, when a client searches for hotels in some context, we create an object of **bookingType** for each hotel in the context, and the client will probably only book one of the hotels. For this implementation, we have a list describing all the objects of booking type, and this list grows every time, a client uses the *getHotel* operation, but we never delete objects from this list. It would be smart to have a way of deleting old bookings.

3.4 LameDuck

LameDuck is a web-service designed in order to provide a client with flights and booking these accordingly. It has been deployed using SOAP and java, and if we look into how its WSDL file is constructed, we can first take a look at the data types the service has defined.

Data Types

In order to support the operations the web-service is supposed to implement, we were in need of three data types, as seen Figure 3.4.

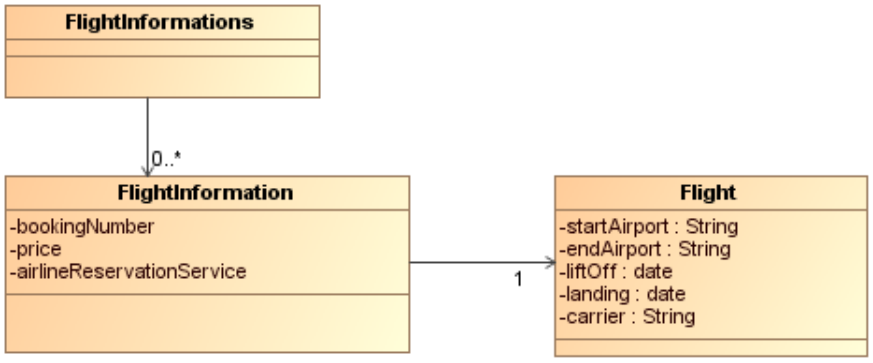

Figure 3.4: The data type defined in the LameDuck web-service

And the purpose of these data types are as follow.

Flight Describes individual flights with every detail needed, i.e. which cities it is going between and what date it is flying.

FlightInformation Is used to link extra information on top of a flight, i.e. its price, a booking number and what reservation service that was used. The purpose of this object is that it allows for multiple prices on a single flight, which could be used to separate economy class with business class.

FlightInformations Is simply a wrapper type, that allows us to send several **FlightInformation** at once from the web-service as a response.

Besides the data types described above, the web-service will make use of the data types provided by the bank service, *FastMoney*. More precisely it will use the **CreditCardInfoType**, found in the bank service, and it will pass this along to all clients that is going to use the services provided by *LameDuck*.

In order for the web-service to keep its flight stored, we have a *HashMap*, in which a combination of the start and final destination combined with the airport act as the key, and all flights for these airport are stored in an *ArrayList* contained in the *HashMap*. This is not the best solution in case we have a lot of flights, but it works well with the small amount of flights currently in place. Otherwise we have another *HashMap* containing all bookings made by a customer, identified by his telephone number, in which all his booking is located within an *ArrayList*.

Operations

LameDuck supports three operations, that can be used by external clients, and these are as follows.

getFlights Will deliver a list of flights going from the given start airport to the destination airport, on the specific date provided by the client. The flights that can be located based on the input, will be returned inside a **FlightInformations** type.

bookFlight Will try and book a flight, in case a valid booking number and credit card information is provided to this operation. In case the credit card information is not valid, a fault is thrown for the client to catch containing a string describing what went wrong. However in case the booking succeeds, the client will receive **true**.

cancelFlight Will try and cancel a booking, for any given booking number. The cancellation will try and refund half the price, given as input, to the credit card, given as input. In case the cancellations for whatever reason fails, a fault is thrown stating that something went wrong. Otherwise, in case of the cancellation succeeding, **true** is returned to the client.

In the current state, *LameDuck*, has support for five flights, used for testing purposes. Two of these flights will make the service fail in different ways, i.e. when one tries to book or cancel a flight. The one in which the booking fails has the price set to -999, while the one that fails the cancellation has a booking number of -999.

3.5 SOAP/BPEL

This section describes the TravelGood service implemented as a BPEL process.
First, we will look at a list of process operations and their usage. After they are defined, we will look at the used types, BPEL process implementation and take a closer look at how it works.

Design decisions (WSDL)

The TravelGood service have one port type 'itineraryPortType' which consists of the following nine operations:

createItinerary This operation just starts the process.

getItinerary This operation returns the current state of the customer's itinerary. All the added flights and hotels are returned in separated lists and each item (flight or hotel) has a status attached which can be either 'Unconfirmed' (not booked), 'Confirmed' (booked) or 'Cancelled' (booked and later cancelled).

getAvailableFlights This operation takes a start and end destination and a travel date, and then returns information about all flights satisfying these input values as a list.

getAvailableHotels This operation takes a city and an arrival and departure date, and then returns information about all hotels available for these input values also as a list.

addPlannedFlight The customer can add a flight to the itinerary (in the planning phase) using this operation.

addPlannedHotel In the same way the customer can add hotels to the itinerary using this operation.

bookAll This operation tries to book all the planned flights and hotels, and returns success (true) if all the bookings were successful and fail (false) otherwise.

cancelPlanning If the customer is in a part of the planning phase where cancellation is allowed, it can be done with this operation. Note that this terminates the process.

cancelBooking If the booking was successful the customer can (try to) cancel all the bookings using this operation.

All of the operations are request/response messages, so the customer is always informed if the requests have been handled. The operations which do not return specific output (addPlannedFlight, bookAll etc.) just return a boolean as status of the operation.

Web services can of course be used by multiple customers. For every customer a process instance is created.
To make sure that the operations the customer makes correspond to the right process, we need some kind of identification of the process instance. For this, a correlation set 'itineraryCorrelation' is defined. Here we can use the customer's id to identify the right process, so a customer cannot interfere with other customer's processes. But the same customer could plan several itineraries and thus only using the customer id will not be sufficient. We therefore also give each itinerary an id and then make the correlation set consist of both of these id's. So every operation has to give a customer id and an itinerary id and then we know which process the operation corresponds to. (If all the itineraries were given unique id's across customers, the correlation set could just consist of these id's).

Business process

The overall BPEL process implemented is illustrated in figure 3.5. It will be based on the coordination protocol described in section 2.
To create an instance of the process, the *createItinerary()* operation is called. This is also where the correlation set is instantiated. All the operations of the TravelGood web service will be using this correlation set. This ensures that operations invoked by the same user only is related to his business process.
Overall, the process is divided into three phases, a planning, a booking and a cancellation phase. These will be described in the sections below. What these phases have in common is that they are surrounded by a while loop, which makes it possible to make a new planning at any time for the same itinerary (if no cancellation has occurred).
The while-loop is contained in a scope which has an event handler. This handler is used to take care of the operations which the user can use, at any time in the process, without affecting the main process. In this process only one of such operations is provided, *getItinerary()* - the user can at all time see what he has in the itinerary. The process flow (how the phases are entered) is controlled by messages and a few global boolean variables - *isPlanning*, *isBooked* and *isCancelled*.

isPlanning Is used to control the while loop in the planning phase. It is updated to true before the planning begins. When the planning is done the variable is updated to false and it leaves the while loop.

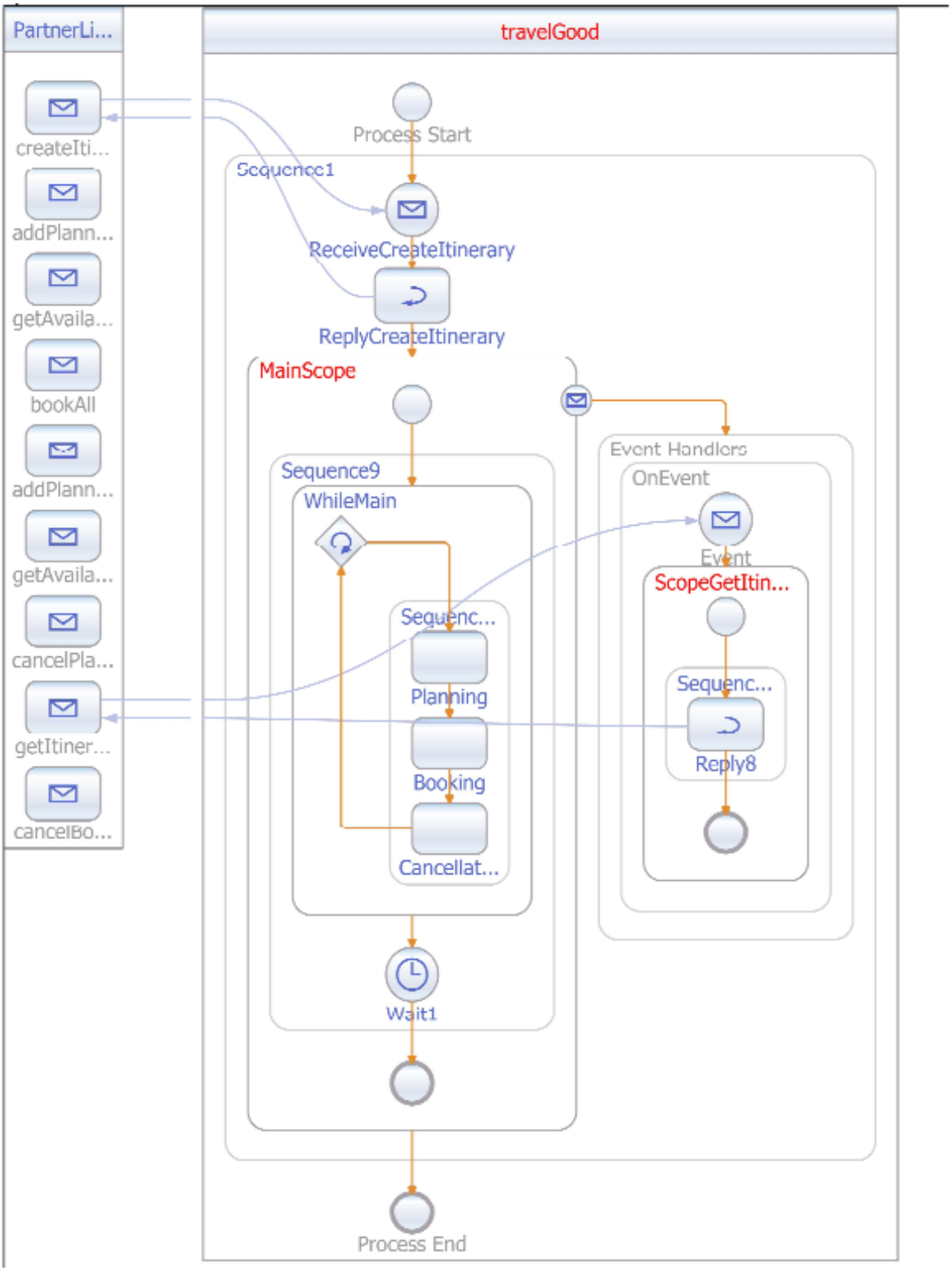

Figure 3.5: An overview of the BPEL process where all assigns and the planning, booking and cancellation phase is omitted (Shown with an empty box).

isAllBooked If all the bookings succeeds in the booking phase this variable is set to true, otherwise to false. It is only possible to enter the cancel phase if this variable is true.

cancelSucced If any cancellation fails, this variable is updated to false. This will make the execution leave the main while loop surrounding the planning, booking and cancellation phase, and results in no further operations can be executed.

Planning phase

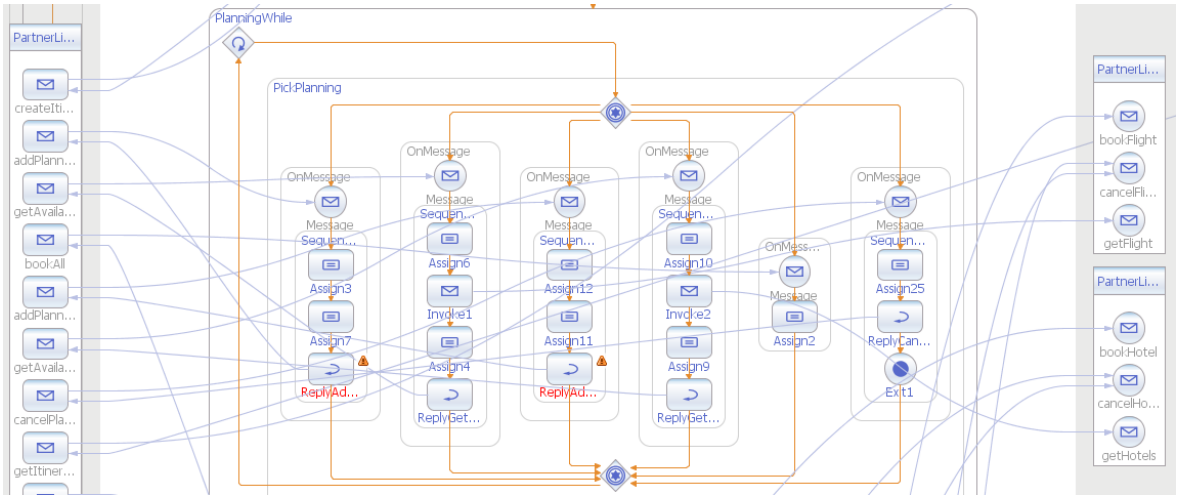

Figure 3.6: The planning phase of the BPEL process

In the planning phase it should be possible to request flight- and hotel information, and add any number of hotels and flights to the itinerary in any order. The customer can also choose to book all the planned items (hotels and flights) or cancel the planning.

This is implemented by pick-activity inside a while loop. The pick-activity waits for one of the following operations: *getAvailableFlights()*, *addPlannedFlight()*, *getAvaliableHotels()*, *addPlannedHotel()*, *bookAll()* and *cancelPlanning()*. The operations *getFlights()* and *getAvaliableHotels()* invoke the external web services, LameDuck and NiceView in order to use their functions *getFlights()* and *getHotels()* respectivly. *addPlannedFlight()* and *addPlannedHotel()* just adds the new item to a list variable (either listOfHotels or listOfFlights) in the process with the status **Unconfirmed**.
By calling the *bookAll()* operation, the global boolean variable *isPlanning* is updated to false, meaning that the process steps out of the while loop and thereby the planning phase of the business process is done.

The operation *cancelPlanning()* simply terminates the process and hereby the whole itinerary is cancelled.

Booking phase

In the booking phase, all of the planned hotels and flights should be booked. Referring to the coordination protocol 2 this phase can result in a series of different states for the overall business process. The booking can fail, which means the other bookings made should be cancelled and the customer should be compensated.

The booking of the hotels and the flights are done in parallel by a flow activity. This is to reduce the waiting time of the booking phase. This can easily be done in parallel because the booking of hotels and flights are independent of each other. The actual booking just invokes

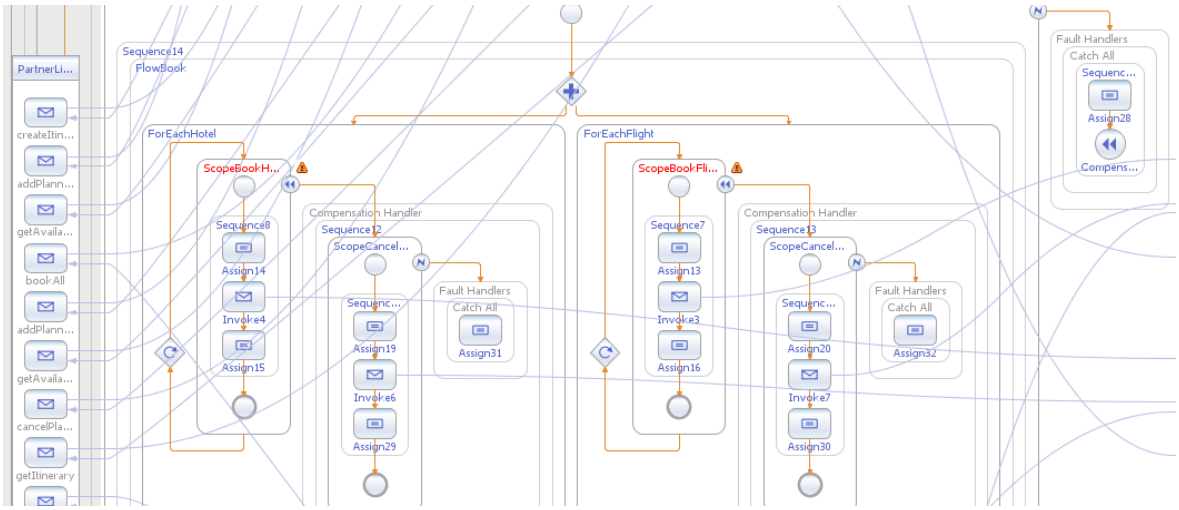

Figure 3.7: The booking phase of the BPEL process

the external web service's booking operations. When a booking is done its status is updated to **Confirmed**.

Since the booking phase changes data in the external web services, a way of keeping the integrity of this data is needed. This is done by a compensation handler. A fault handler containing a compensation element is attached to the outermost scope of the booking phase. This is because a fault in any booking should start the compensation of all completed bookings. The global boolean variable isAllBooked is in this case updated to false. The hotel and flight booking scopes got their own compensation handler that calls the relevant cancel operations of the external web services to undo the bookings. The status of all the cancelled items is afterwards updated to **Cancelled**.

If the cancellation fails, the compensation handlers of the cancellation scopes got fault handlers that catch the fault and simply proceed with the next cancellation. In this case, a global boolean variable, *cancelSucceed*, is updated to false, which indicates a cancellation has failed. This results in the process leaving the main while loop and getting to a wait activity, forcing the process to wait until a defined date. Based on our coordination protocol, this date is one day before the departure date of the itinerary. The departure date for the itinerary could be determined by having a date variable in the BPEL process, which would be updated every time an item with an earlier departure date than the currently stored was added to the itinerary.

We have had trouble with the BPEL date type, so we weren't able to either initialize nor store our departure dates. So, in our implementation, we wait for a constant duration.

Cancellation phase

After the booking phase, the customer can only see the itinerary or cancel it. Therefore after the booking phase the process gets to the cancellation phase.
Here, the customer has the opportunity to cancel the itinerary, meaning cancelling all the hotels and flights. The cancellation is done in parallel using a flow activity. The flow activity is inside a pick element with an *onAlarmEvent* branch. This makes it possible to cancel the bookings until the alarm condition is reached and the process terminates. Like explained for the wait activity, the alarm event should trigger one day before departure, but once again the alarm is triggered after a constant duration.

When a booking is cancelled its status is updated to **Cancelled**. If all the cancellations of the bookings succeed, the process returns to the planning phase. Since a cancellation can

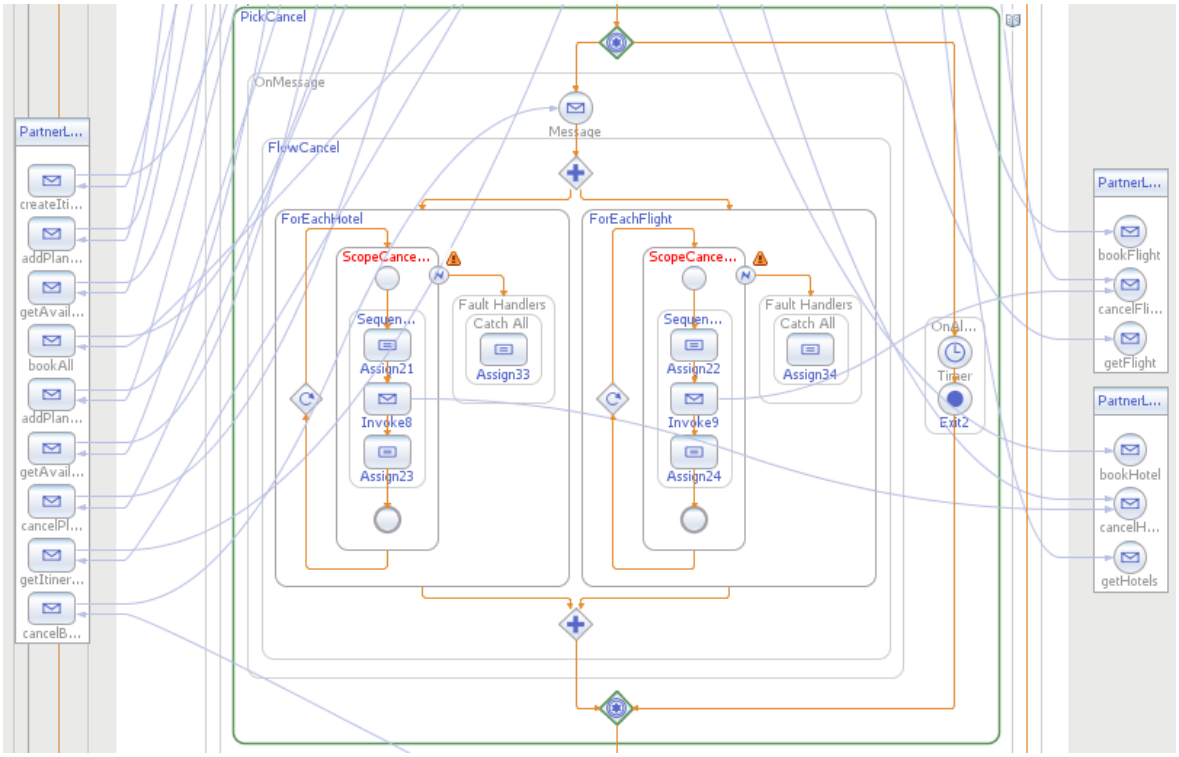

Figure 3.8: The cancellation phase of the BPEL process

fail, there is a fault handler attached to each cancellation, catching the fault. (The fault handlers are attached to the inner scope of the loops, which can be seen on 3.8.) In this case the failing item is skipped, but the rest of the bookings are still to be cancelled. When the cancellation is through, the process proceeds to the wait activity outside the main while loop.

Tests

Beside the five required JUnit tests, we have created two extra JUnit tests which can be found in `ExtraTests.java`.

The first test, tests that a cancellation fail is correctly handled if the fail occurs while the bookings are cancelled due to a booking fail.

the second test, tests the flow of the process - that a customer can book an itinerary and then later cancel it, add another flight and book it all again.

Extensions

Because we have used flow activities to do the bookings of the flights and hotels in parallel, the process can get a race condition problem when a booking fails, and the booked items have to be cancelled. Let us say a flight booking fails while a hotel is about to be booked, then our process will call the compensate handler which only will compensate all finished bookings. But the hotel about to be booked has not finished its booking process yet, but will neither stop it. It will therefore end up being booked and "Confirmed", where all items either should be "Unconfirmed" or "Cancelled". This problem could be solved if the booking process and the compensate handler had a lock. The compensate handler can this way only start when no bookings are in process.

Another problem with the business process is that the itinerary is only stored as local variables inside the process. This means that all the data of the itinerary is lost when the process is terminated. In case the customer has complaints about the itinerary later, the

company does not have any information left about the itinerary. If the process crashes early due to an error all data will also be lost and the customers will be unhappy.

A solution to this problem would be to store the itinerary information in a database instead of in local variables. The process would otherwise be identical and work in exactly the same way, but the data is now not completely gone when the process is terminated.

3.6 RESTful Implementation

This section describes the TravelGood web-service implemented using the RESTful approach. First we will describe the resources that is provided to the client from the service. Second we will describe in detail the choices we had when mapping HTTP verbs onto the resources available on the service. Lastly we will describe how the business logic for the process has been implemented.

Resources

In order to support the business process, we have three resources that need to be represented in the RESTful, and these are **Flights**, **Hotels** and the **Itineraries** themselves. These are the only objects that is being used within the service, and it was therefore an easy choice of having these three types represented as a resource. The reason for not having them represented as a single resource, was to get a separation of the unique resources stored in the services.

Besides these trivial resources, we have chosen to represent the **Booking** as an individual resource. This was done in order to allow for easy booking of a selected itinerary, as well as having a separated resource representing the booking state, defined in the business logic.

Finally we have chosen to have a **Status** resource, which will support cancellation and providing status on which state in the business process the itinerary is located in. This was created to separate cancelling from the itinerary, as the **Itinerary** resources should be accessible in all states of the business process, whilst cancellation is only allowed in a sub-part of the business process.

One last resource which has been created is a **Reset** resources, which only will be used for testing purposes. It will simply delete all itineraries made and give us a fresh start on the RESTful web-service, meaning we get consisting test results. This should probably be removed before the service was to go live.

Mapping to HTTP verbs

If we look at each individual resource found on the web-service and what action is possible to perform upon these resources and how these have been mapped to HTTP verbs we have:

Flights The **Flights** resource, is responsible for getting flights and adding any desired flights to a specific itinerary.

- **get flights** Has been mapped to *GET*, as the client will potentially get a list of flights as a result.
- **add flight** Has been mapped to *PUT*, as the client updates an already existing itinerary with the flight he wishes to add.

Hotels The **Hotels** resource, is responsible for getting hotels and adding any desired hotel to a specific itinerary.

- **get hotels** Has been mapped to *GET*, for the simple reason that the client in the end will receive a list of hotels, and no changes is made on the web-service in the process.

add hotel Mapped to *PUT*, since it will update the selected itinerary with the hotel given as argument.

Itinerary The **Itinerary** resource, is responsible for creating new itineraries and retrieving them after creations.

create itinerary Has been mapped to *POST*, as we will attempt to create a new itinerary for a customer. And since the customer is not posting any specific details for the itinerary during the process, one cannot use *PUT*, as no updates occur during the process.

get itinerary Has been mapped to *GET*, as the itinerary will get returned to the customer, whilst no changes will happen on the web-service.

Booking The **Booking** resources sole purpose is to book the itinerary selected by the customer, and it is not part of any other resource, since it is only supposed to be used in one state of the business logic.

book itinerary Is mapped to *PUT*, since it will try to update the status of the booking, by booking all hotels and flights added to the itinerary.

Status The **Status** resource is responsible for cancellation of an itinerary, and providing the customer with a status on the itinerary.

cancel itinerary Is mapped to *PUT*, since it in most cases will not delete the itinerary, but simply update the status which is assigned to the itinerary including all flights and hotels assigned to it. Due to the fact that it only in rare cases actually deletes the itinerary completely from the system, we have chosen not to use *DELETE*.

get status Is mapped to *GET*, since it will return the current status for the selected itinerary.

Implementation of Business Logic

In order for the business logic to be implemented we have introduced links, containing information on the resources a customer is allowed to access after a specific call has been processed and the control returned. These links contain all relevant information needed in order to access a specific resource. The links will contain the following information:

URI The exact URI for the resource, in which the customer can access the next step in the business logic. Will be unique depending on the customer and itinerary that the customer is working on at the current moment.

Relation A static name, that is the same for any resource of the same type in the system. This can be used to locate a specific link between several others.

Media Type Describes what media type is allowed by accessing the URI given in the link.

Relations In the web-service we have defined what links should be placed upon a response according to what state the itinerary is currently placed in. For our web-service we have provided several **Relations**, one for each resource located in the web-service. These all share the base url *http://travelagency.ws/relations/* by either *booking*, *status*, *flights*, *hotels* or *self*, depending on the resource the customer wants access to. Where they all correspond to the resource with the same name, except *self* that is a relation to the **Itinerary** resource. We chose to call this *self* as it is a relation to the overall resource which we consider to be the **Itinerary**. So if we relate these to the different states that can occur throughout the

business process, i.e. **Planning Phase**, **Booking Successful**, **Booking Complete** and **Cancel failed**, a relation table can be constructed as seen in table 3.1.
And the links given should be provided whenever one ends in the acceptable state(s).

Link Package	Acceptable State(s)	Link Required
Planning Links	**Planning Phase**	*booking*, *flights*, *hotels*, *status*, *self*
Booking Links	**Booked Successful**	*status*, *self*
Locked Links	**Booking Complete**, **Cancel Failed**	*self*

Table 3.1: Resource and State relations

Media Type The media type, which will be used throughout the whole service is *application/ itineraryprocess+xml*, so we have an increased chance of getting the right format at any time a client makes a request to the service.

URI For the web-service developed, we have defined our base URI for all resource to be:

http://localhost:8080/TravelAgencyREST/resources/TravelAgency/

where we for all resources will follow it by first the *customer id* and next the *itinerary id*. This means that every itinerary has a unique URI, that the customer has to use in order to access his itineraries. And simply by given the above URI with both *customer id* and *itinerary id*, one gets access to a specific itinerary. Otherwise one can follow it with, *Booking*, *Flights*, *Hotels* and *Status* depending on what resource the client wants access to. For the *Hotels* and *Flights* we use *QueryParams* in order to get the intended values for *getFlights* and *getHotels*.

By having these links, we can with ease guide the client through the business process, however the client can easily disrupt the process, as he can simply call another resource which is out of scope of the current business process.

Representations

In order to use the links with the responses that the user will receive, we need **Representations** that encapsulates an object with several links describing the next actions one can take in the business process. The representations we have chosen to have in our web-service is shown in Figure 3.9.

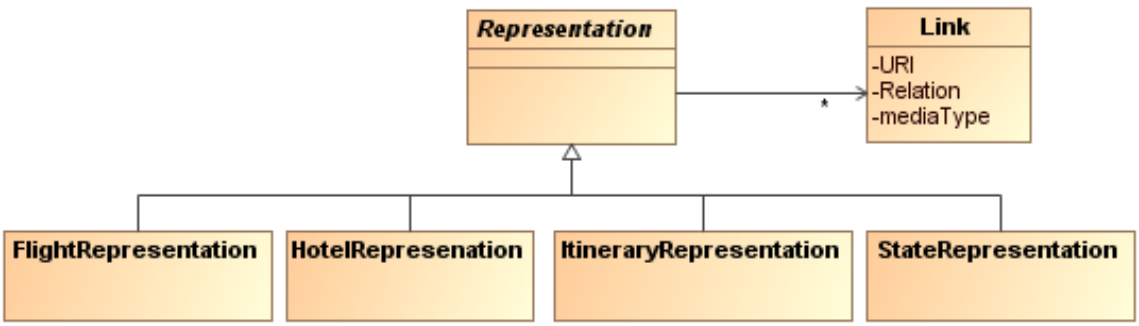

Figure 3.9: The representation available in the web-service

As seen we have four representation, in which three of these is encapsulating objects occurring on the web-service, that has the possibility to be sent to the customer as a response to a request.

FlightRepresentation Is responsible for holding a list of flights, that a customer has requested through the **Flight** resource, with links to the next step in the business logic that can be performed. We chose to have this representation, since we need a method of providing flights to the customer with the respective links.

HotelRepresentation Is responsible for holding a list of possible lists of hotels acquired from the **Hotel** resource, including links for further action that can be taken. It has been created in order to provide hotels to the customer with respective links.

ItineraryRepresentation Hold the full itinerary requested from the **Itinerary** resource, as a response to the customer with the links needed for the current state in which the itinerary is currently at. This has been created in order to get a possibility for sending the itinerary to the customer, so he can get an overview of what he has booked/planned.

At last we have a representation, that is not encapsulating any object, namely the **StateRepresentation**.

StateRepresenation Is used whenever there is no object to be returned to the customer, but the links is still needed, meaning we use this representation as a dummy for the state. However it is also responsible for telling the customer what succeeded or what failed during the request. It was created as we needed a method in which methods that do not return any object could have links returned in order to find the next actions it could take. This meant that we found an opportunity for providing a status update, since we needed to provide links.

Chapter 4

Web Service Discovery

Web service discovery is the task of finding a suitable service for a certain task. The first part of this is to find the correct provider. For a flight plan one could try finding it through the local airport or the company one wishes to fly with. When having decided on a provider a method for determining what resources are available is needed. For this two approaches exists.
The first approach is **UDDI** which is a centralized register where any company can get their services registered. Customers of these services can now access them by searching for the company in the register and then get a list of all services provided by the provider. In the case of the flight plan the customer would search for either provider in the register and then from here find the service that would return a flight plan or a combination of services that would provide enough data to build plan from it. See figure 4.1.
The second approach is **WSIL** which is a localized way of describing your services. A main inspection file is to be placed at the root of your services (called *inspection.wsil*) and this should then either describe every service provided, or point to other inspection files from which information about the services can be gathered. In the case of the flight plan, the initial inspection file would have to be evaluated and the service which provides the most optimal return of resource, should be chosen. See figure 4.1.
Both methods can be used in unison if a provider first creates an entry in the UDDI register and then appends the link to that entry in the WSIL file with the link-tag.

4.1 Our implementation

All files are named *inspection.wsil*.

TravelGood

This file is placed at the root of the RESTful implementation of TravelGood, at the same level as the index page. It contains links to the inspection files of both NiceView and LameDuck.

NiceView

Placed at the root of the SOAP implementation of NiceView, alongside the index file.

LameDuck

Placed at the root of the SOAP implementation of LameDuck, alongside the index file.

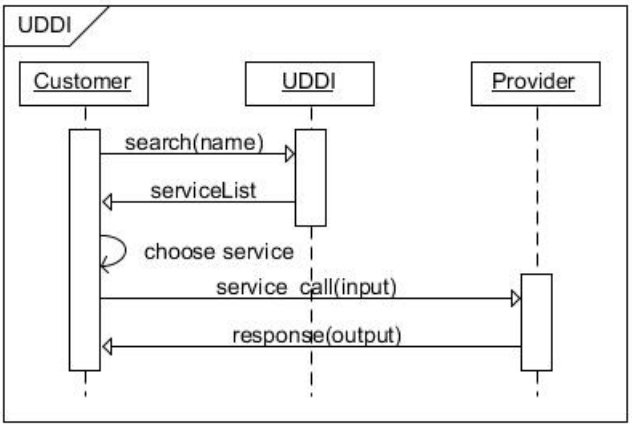

Figure 4.1: Communication process when obtaining and using a service with UDDI

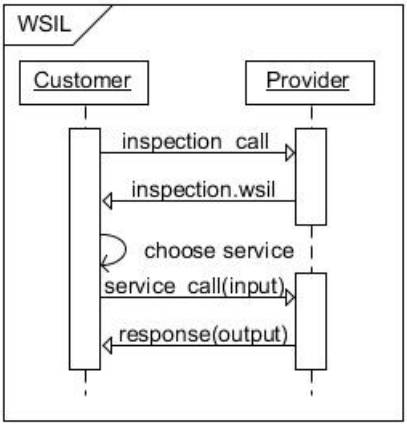

Figure 4.2: Communication process when obtaining and using a service with WSIL

Chapter 5

Comparison of RESTful and SOAP/BPEL Web Services

We will in the following compare the experiences made with RESTful versus SOAP/BPEL based upon the implementations we have created. Some general thoughts we have about the developed product is that it is mainly based on implementing a business process, which SOAP/BPEL is better suited for. Therefore we are also more positive towards BPEL in the following discussion than towards RESTful.

5.1 Implementation

The biggest difference between SOAP/BPEL and RESTful is how one implements business logic. We experienced that for SOAP/BPEL it was an easy task to keep track of the current state in the business process, and the client was forced to only do actions allowed by the given state. RESTful on the other hand has no states and can therefore not force the client to only use permitted actions on a specific state. A RESTful implementation can try to enforce this by providing the resources that can be accessed in the next step, however the client can easily break this restriction. This can also be seen by the handling of time events, as SOAP/BPEL support this by default, where RESTful on the other hand relies on using sleeping threads. An experience made during this project, was that we found SOAP/BPEL better at handling errors, meaning one could easily get an overview of whether the correct fault procedure had been implemented. In a RESTful service, this can be hard to detect as the fault procedure is written in plain code. A SOAP/BPEL process is also excellent at providing specific faults in case a critical error occurs, while RESTful will have to explain in plain text what a specific fault was caused by. A major advantages, from our point of view, of using SOAP/BPEL is that it provides XML schemas meaning that the classes it provides are visible for the client, which again leads to that one can get a quick overview of the possibilities a service provides. The RESTful service does not provide this, and the client must know how the objects look in advance, which is a big disadvantage in case the client has no knowledge of the resources on the web-service. A point that talks towards the use of RESTful services was the easier implementation, code-wise. This probably stems from the fact that we are more used to program in a text-based fashion, opposed to using drag-and-drop as seen in SOAP/BPEL.

5.2 Understanding the implementation

In regard to understanding the implementation, we found the SOAP/BPEL implementation to be the easiest, due to its graphical representation. In SOAP/BPEL one can easily see the whole process on the screen at once. The RESTful, does not provide this, since the different

steps in the business logic is divided into several resources and thus one can easily lose the overview of the full process. This also means that it is easy to lose track, if all of the business process has been implemented in RESTful and there only exists one correct sequence for the process to follow. Even though it is easy to get an overview of the business process in SOAP/BPEL, it is quite hard to know what the individual steps is implementing. In RESTful on the other hand it is easy to get a quick overview of the individual steps in the process.

5.3 Changed requirements

We feel that a SOAP/BPEL service will handle with grace any changes, as one can simply drag-and-drop the features in or out of the processes in order to make it fit the new requirements. A RESTful service on the other hand might need several classes rewritten or new resources introduced, in order to comply with the new requirements. So overall we fell that SOAP/BPEL is more suited for requirements changes.

5.4 Scalability

Regarding scalability we find RESTful the best, as the messages being sent between the service and client contain less information, as SOAP has the need for sending a header with each message that is sent between the client and service. This means that a RESTful service has need of less bandwidth, meaning that the service may complete more request than a SOAP/BPEL service. However since RESTful is using HTTP, all calls are synchronous, opposed to SOAP/BPEL that support both synchronous and asynchronous meaning it is better for long term scalability, as request can be put on a queue, meaning less threads is needed for SOAP/BPEL in case asynchronous call are used.

Chapter 6

Advanced Web Service Technology

6.1 WS-Addressing

When we have a SOAP-message, we cannot see who the sender of the message is and neither can we see the intended receiver. This can cause problems. To address this issue there is WS-addressing which is contained in the header of the SOAP message. Information contained in such header can include:

- Unique identifier of the message as an URI.
- Recipient of the message.
- An action value indicating the semantics of the message.
- Relationship to other messages as URIs.
- Sender of the message.
- A reply endpoint used for recipients of replies.
- A fault endpoint, for which faults are sent to.

So from the header of the SOAP message, one can see where the message comes from and who the intended receiver of the message is. Furthermore, one can see relationships to former messages, and what the purpose of the message is, eg. *bookHotel*, from the action value.

In connection to Addressing, we can see recipient and sender of the messages. When a client sends a request to the agency, we would like to know who we should reply to. Therefore Addressing can be used in accordance with our project.

6.2 WS-Reliable Messaging

For SOAP messages, we cannot be sure that a message arrives to the receiver, as messages can be lost. And furthermore, we cannot be certain that messages arrive in the order that they are sent. Reliable messaging should ensure that messages are received (which can be done by having an acknowledging messages), and that they arrive in the given order.

An example of a ReliableMessaging protocol is this horizontal protocol. When we send a message, we wait for some acknowledgement by the receiver, if we wait for too long without receiving the acknowledgement, we resend the message. All this resending and acknowledging of messages should be handled by a ReliableMessagingLayer, so the whole process is transparent to the application.

Different types of delivery assurances exists. For AtMostOnce, messages are not resent if lost, meaning each message can at most be received once by the recipient, however we cannot

be certain that messages will arrive. For AtLeastOnce, we resend messages if we do not receive an acknowledgement, meaning we ensure that each message will be received by the recipient, however the same message might arrive multiple times. A third alternative is ExactlyOnce, which means that all messages will be received exactly once by the receiver, meaning messages will be resend if they are lost, however we need a higher level of administrative effort in order to determine which messages have been received and which are lost, so as not to send a received message twice.

Get-operations are less reliant on the ReliableMessaging protocol as you always expect to get an output. For the book- and cancel-operations it is nice for the client to know whether a booking or cancellation was successful and ReliableMessaging is therefore a great way of ensuring this.

6.3 WS-Security

It is easy for an outside person to read and change a SOAP message, which is not preferable and can cause serious damage in some cases. To avoid this we have different types of security. We have authentication, meaning that we can authenticate that the identifier of the sender of a message is in fact the original sender of the message. We also have Privacy/Confidentiality, which means that a third person should not be able to read the content of the message, even if he/she gets a hold of it. Furthermore we got Integrity, meaning we ensure that the message has not been changed since the message was sent from its original point.

We know different techniques for ensuring Authentication, Privacy and Integrity through encryption and digital signatures. Such techniques can be used on a message-level basis to create the necessary security for the messages.

For our project, there are some messages that contain sensitive data that could be a security concern. When we have messages sending credit card information the confidentiality should be enforced, so an outsider would not be able to gain this information even if the message should be intercepted. This way we can also assume that if the credit card information is correct, then it is a legitimate booking of a hotel. We assume people are aware of keeping their credit card information secret. So the credit card information also serves the purpose of authenticating the message.

6.4 WS-Policy

WS-policy is a specification of the rules for a web service to follow. A web service can use WS-policy in order to have the constraints and rules of its policy specified in XML. Examples of these policies are: Safety policies, for the intermediaries and endpoints.

A WS-policy is similar to a boolean expression and has no standard assertions. Assertions of policies are defined in their own name space.

We mentioned some important security aspects for our project in 6.3. These aspects should be specified in the policy for our service. Another policy is the demand that all responds must happen within five seconds of the request being sent.

Chapter 7

Conclusion

Developing a web service with two different implementations of the same goal has been an interesting task. The Travel Agency web service was a classical business process which prove to be a perfect match for a BPEL implementation, since BPEL is made for coordinating business processes. It was possible to make a good RESTful implementation, however in similar projects in the future we will prefer to work with a BPEL implementation. Even though RESTful had its advantages in some situations, BEPL gave a better result overall.

We had the feeling that netbeans was not mature enough for projects with a considerable size. Debugging was cumbersome with too many standard faults for a large range of fault scenarios. Furthermore we did not find the BPEL community as helpful (smaller in size) as we are used to from other technologies.

Starting out with creating a coordination protocol has been very helpful. This has been ensuring that the goal of both of the implementations have been meet.

A key point of choosing BPEL was that it is easier to understand the implementation for, for example, a manager with business knowledge and limited programming skills. This is true in smaller projects, however we discovered that even in products with as low complexity as in this project, the understandability decreases significantly.

We acknowledge that if this project should be able to have the demanded quality needed by such business processes, more advanced WS-technologies are needed. Especially for security.

Chapter 8

Who Did What

The group work has been equally distributed amongst the group members. Every one has worked together as authors and contributed to the whole report as well as the programming part. The next section gives an overview of the members' main responsibilities.

8.1 Distribution

Thomas
(1.1) Web service composition and coordination, (3.5) Business process, Planning phase, (5.1) Implementation, (5.2) Understanding the implementation, (7) Conclusion
Program: planning.ws

Jacob
(3.3) NiceView, (6.1) WS-Addressing, (6.2) WS-Reliable Messaging, (6.4) WS-Policy.
Program and tests: NiceView

Caspar
(1.1) Web service discovery, (3.2) Common design decisions for SOAP implementations, (4) Web Service Discovery, (6.3) WS-Security
WSIL-files.

Andreas
(1.1) RESTful services,(3.1) Common data types, (3.1) RESTful, (3.4) LameDuck, (3.6) RESTful Implementation, (5.3) Changed requirements, (5.4) Scalability
Program and test: LameDuck and RESTful TravelGood

Jasmina
(1) Introduction, (1.1) Web service description, (3.1) SOAP/BPEL data types, (3.5) Booking phase, Cancellation phase
Test files: RequiredTests.java, ExtraTests.java

Gert
(1.1) Service Orientation Architecture, (1.1) SOAP, (3.5) Design decisions, (3.5) Extensions, (3.5) Tests, (2) Coordination protocol
Program: travelGood.bpel

Bibliography

[1] Web Service Discovery Mechanisms: Looking for a Needle in a Haystack
http://mmlabold.ceid.upatras.gr/people/sakkopoulos/conf/ht04.pdf
John Garofalakis1, Yannis Panagis1, Evangelos Sakkopoulos, Athanasios

[2] Web Services Inspection Language (WSIL)
http://publib.boulder.ibm.com/infocenter/radhelp/v8/index.jsp?topic=/org.eclipse.jst.ws.doc.user/concepts/cwsil.html

[3] Web Services Coordination (WS-Coordination)
http://jotm.objectweb.org/related/ws-coor.pdf
Felipe Cabrera, Microsoft <cabrera@microsoft.com>, George Copeland, Microsoft <gcope@microsoft.com>, Tom Freund, IBM <tjfreund@uk.ibm.com>, Johannes Klein, Microsoft <joklein@microsoft.com>, David Langworthy, Microsoft <dlan@microsoft.com>, David Orchard, BEA Systems <dorchard@bea.com>, John Shewchuk, Microsoft <johnshew@microsoft.com>, Tony Storey, IBM <tony_storey@uk.ibm.com>