Chapter 5
Beyond CRUD

Irum Rauf and Ivan Porres

Abstract REST web services offer interfaces to create, retrieve, update and delete information from a database (also called CRUD interfaces). However, REST web services can also be used to create rich services that offer more than simple CRUD operations and still follow the REST architectural style. In such a case it is important to create and publish behavioral service interfaces that developers can understand in order to use the service correctly. In this chapter we explain how to use models to design rich REST services. We use UML class diagrams and protocol state machines to model the structural and behavioral features of rich services. The design models are then implemented in Django Web Framework. We also show how to use the behavioral interfaces to implement a service monitor.

Introduction

The interface of a web service advertises the operations that can be invoked on it. A web service developer looking for a particular service finds the service over the web and integrates it with other services by invoking the advertised operations and providing it the required parameters.

Many RESTful web services present simple interfaces to create, retrieve, update and delete information from a database (also called CRUD interfaces). However, REST is not limited to simple CRUD applications. It is possible to create web services exhibiting a rich application state that still follow the REST architectural style, e.g., flight and hotel reservation systems, stock trading services etc. In such cases, it is important to create and publish behavioral service interfaces so other developers can understand how to use a service correctly. A behavioral interface

I. Rauf (✉)
Department of Information Technologies ICT, Abo Akademi University,
Joukahainengatan 3-5 A, FI-20520 ABO, Finland
e-mail: irauf@abo.fi

E. Wilde and C. Pautasso (eds.), *REST: From Research to Practice*,
DOI 10.1007/978-1-4419-8303-9_5,

of a web service provides information about the order of invocation and about any special conditions under which interface methods can be invoked and their expected effect.

A REST interface should offer features of *addressability*, *connectedness*, *uniform interface* and *statelessness*. In order to provide these interface features for beyond CRUD REST applications along with behavioral interface specifications, we present a design methodology that caters to the REST design philosophy earlier in the development cycle (Porres and Rauf 2011). The design approach addresses modeling of REST features using UML (Unified Modeling Language) (OMG UML 2009), thus creating web services that are RESTful by construction. In this chapter, we overview the design methodology presented in Porres and Rauf (2011) and then detail how the design approach is implemented in Django Web Framework. The service monitor implemented in Django Web Framework checks the correctness of a service with regard to its design.

Modeling the RESTful way

Models represent a system in graphical notations that are easier to understand and communicate between system developers and with other stake-holders of the system. We use UML to model the structural and behavioral features of REST web service. UML is a standard modeling notation and is well-accepted by industry. It provides representation of the system in an abstract manner from different perspectives and also serves as part of the specification document (Mens and Gorp 2005).

The objective of this modeling activity is to represent a REST web service with UML models that provide features of a REST interface, i.e., *addressability*, *connectedness*, *uniform interface* and *statelessness*. Using these design models, we can create a web service that will exhibit REST features thus making it RESTful by construction.

The starting point of the modeling activity is an informal web service specification in natural language. This specification is used to model structural features as a conceptual resource model and behavioral features as a behavioral model of the web service. Both the models are built in parallel and refined iteratively.

REST web services expose their functionality through resources. We model these resources in our conceptual resource model. The conceptual resource model is represented by a UML class diagram and tackles the *addressability* and *connectivity* requirements of a REST interface. The behavioral specifications of an interface are represented with UML Protocol state machine. A protocol state machine contains a number of states with state invariants and transitions. Each transition is triggered by a method. In a RESTful interface, resources do not have different access methods, instead the standard HTTP methods are used. Our approach uses four HTTP methods, i.e., GET, PUT, POST, and DELETE, for retrieving and updating data in a resource. The behavioral model tackles with the *uniform interface* and *statelessness* features of REST style.

In the next two sections, we show how these models are developed. We use as example an imaginary hotel room booking (HRB) service. The service allows a client to book a room, pay for the reservation, and cancel it. It is a simplified pedagogical example, but it shows how to design a REST interface for a service with a complex application state.

Conceptual Resource Model

A RESTful web service is data-centric and exposes its functionality through resources. Each resource has a representation in the form of data attributes. These resources form part of the static structure of the web service. We represent this static structure as a conceptual resource model using UML class diagram. A UML class diagram represents classes and associations between them. An association defines a relationship between two classes by which one class knows about the other class (OMG UML 2009).

As a starting step we analyze the natural language specifications of the service and identify the resources. Any important information in a service interface is exposed as a resource. Each resource is shown as a class in the class diagram. Identifying resources can be an iterative process and as we analyze and design the behavioral model of a web service, we can add or remove the resources in its conceptual resource model. As a general practice, the number of resources can be increased to reduce the complexity of a service interface. Every piece of information that needs to be retrieved or manipulated by the users of the service is modeled as a resource.

Figure 5.1 shows the conceptual resource model of the HRB RESTful service.We have broken our HRB service into six (non-collection) resources, i.e, (*booking, room, payment, pwaiting, pconfirmation and cancel*). A user interested in retrieving certain information can invoke a GET method on that resource and get representation of resource as a response. For example, if a user is interested in knowing whether a booking is canceled or a certain payment is confirmed, she would invoke GET method on *cancel* or *pconfirmation* resource, respectively.

A resource can also be a collection resource that contains a group of other resources. A collection resource is identified from the specifications and stereotyped as *<<collection>>* in the conceptual model. In Fig. 5.1, *bookings* and *rooms* represent collection resources with the stereotype *collection* and are linked to child resources, *booking* and *room*, respectively. A collection resource has a cardinality of more than 1 on the association end of a child resource. A GET method on a collection resource returns a list of all the child resources it contains. For example, a GET method on *bookings* will give a list of all the *booking* resources that it contains.

The attributes that form representation of a resource are represented as attributes of a class. These class attributes would appear in the resource representation, i.e. an XML document or a JSON serialized object.

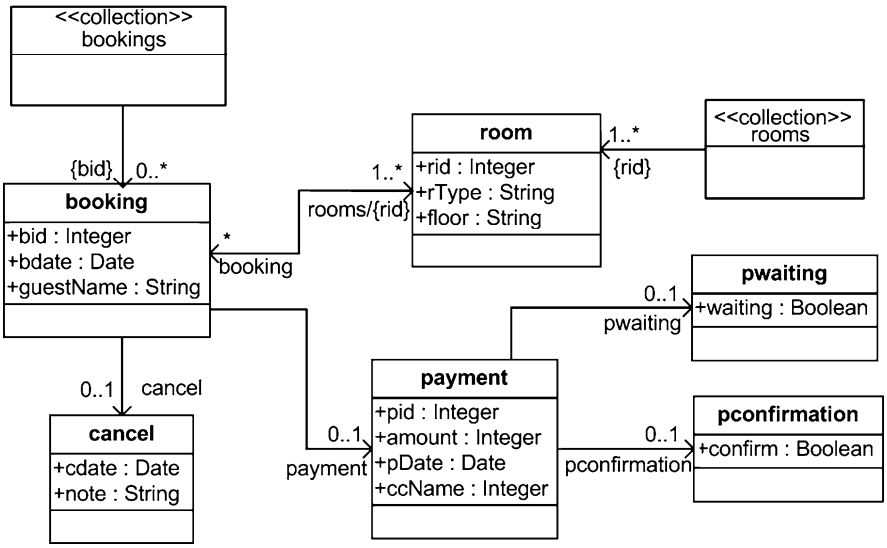

Fig. 5.1 Conceptual model for HRB RESTful web service

Figure 5.1 shows representation of resources in the HRB service. For example, *room* resource contains three attributes i.e. *rid, rType and floor*. Room ID(*rid*) and floor(*floor*) are integer values and room type(*rType*) is a string value. Attributes are modeled as a public attribute as the representation of a resource is available for manipulation.

Classes are connected via associations and each association is marked with role names on association ends. These associations show connection between resources and their multiplicity shows number of resources that can be related to the resource on the other end of the resource. These associations provide addressability and connectivity features to web service interface as explained in the next section.

Addressability and Connectedness

The associations between classes in the conceptual model provide information on the connection between the resources. The association direction shows the navigation direction and the role names on the association ends show the relative navigation path. Collection resources can be used as the starting point of the navigation paths to address each resource. Starting from a collection resource, we can access other resources by navigating the successive associations. For example, in Fig. 5.1, payment resource of a particular booking with id {*bid*} is retrieved by visiting the path /*bookings*/{*bid*}/*payment*/. Paths visiting the same association more than once are not valid. In our example, the valid paths are listed below.

```
/bookings/{bid}/
/bookings/{bid}/cancel/
/bookings/{bid}/payment/
/bookings/{bid}/rooms/{rid}/
/bookings/{bid}/payment/pconfirmation/
/bookings/{bid}/payment/pwaiting/
/rooms/{rid}/
/rooms/{rid}/booking/
/rooms/{rid}/booking/cancel/
/rooms/{rid}/booking/payment/
/rooms/{rid}/payment/pconfirmation/
/rooms/{rid}/payment/pwaiting/
```

The REST style requires that all resources should be addressable and connected. Thus, we require that our resource model should not contain an isolated resource. Each resource can be reached from at least one collection resource by navigating one or more associations.

Uniform Interface

A UML class diagram allows us to define a number of operations for each class. Since a RESTful web service provides uniform interface for all resources, all resources would only have from one to four method names GET, POST, PUT, and DELETE. Thus, we do not show operation information in the conceptual resource model. However, by constraining the allowed transition triggers in behavioral model to the standard HTTP method we comply with the uniform interface requirement.

Behavioral Service Model

The purpose of the behavioral model is to describe the behavioral interface specifications of a RESTful web service. It shows the sequence under which operations should be invoked, the conditions under which they can be invoked and the expected results.

We use a UML protocol state machine with state invariants to describe the allowed operations in a web service. A UML protocol state machine is suitable for representing the behavior of a web service as it provides interface specifications that give information about conditions under which methods can be invoked and their expected output.

A UML protocol state machine contains mainly states and transitions. We require that each state has a state invariant that is defined as a boolean expression. We then say that a state is active if and only if its state invariant evaluates to true. A state may contain other states and is called a composite state. In such a case, the actual

state invariant of the contained state is given by the conjunction of the state invariant specific for the contained state and the state invariants of all the states that contain it. These state invariants within a composite state should be mutually exclusive. That is, only one state within a region of a composite state can be active at a time.

A transition is an arc from one or more source state(s) to one or more target state(s) labeled with a method name and a guard. If the source states are active, the guard is true and the method is invoked, then the transition may be fired and as a consequence the target state(s) become active. When no guard is shown in the transition it is assumed to be true.

Since we are describing RESTful web interfaces, the only allowed operations are GET, POST, PUT, and DELETE on resources.

The GET method retrieves representation of a resource and it should not have side effects, i.e., not cause a change in the state of the system. Due to the addressability requirement, it is possible to always invoke a GET method over a resource. For example, *GET(/bookings/{bookingId}/payment/)* and *GET(/bookings/{bookingId}/cancel/)* represent GET requests on resources *payment* and *cancel*, respectively. Whenever a GET method is invoked on a resource, it gives the representation of resource as a response if the resource is present, else a response code of 404 is sent back. In practice, the access to resources may be restricted by an authentication and access control mechanism.

The transition triggers can only be defined as POST, PUT, or DELETE operations over resources described in the conceptual model. The POST, PUT, and DELETE methods can have side effects, i.e., they can cause a change in the state of the system.

Our behavioral model shows different states of a RESTful web service and gives information on what HTTP methods on a particular resource can be invoked from a certain state. According to Fig. 5.2, the protocol state machine of HRB service is initiated by the HTTP POST method on the *bookings* resource. The client can make payment for a booking by invoking a PUT method on *payment* resource only if the name of the credit card is same as the name of the guest. The booking service invokes a third party credit card payment service(CCService) from the *paid* state as an internal action. If the CCService is asynchronous, then the booking service invokes a PUT on *pwaiting* resource and the transaction enters a *wait* state. It then invokes a PUT on *pconfirmation* resource when response is received from the CCService. If the CCService is synchronous, the booking service invokes a PUT on pconfirmation resource from the *paid* state when it receives response from the CCService. The case of synchronous and asynchronous services is explained in "Synchronous and Asynchronous Web Services." A booking can be canceled from the composite state *reserve_and_pay* and simple state *pconfirmation_info*. A booking cannot be canceled if it is waiting for the payment confirmation from a third-party service. A booking can be deleted only if it is canceled. Note that all the information needed to process the request on a resource are contained in the invoked method and URL.

A GET method can be invoked on every resource as it is free of any side-effect. However, a closer look at the behavioral model also exposes information about the allowed side-effect methods on a resource. For example, Fig. 5.2 shows that

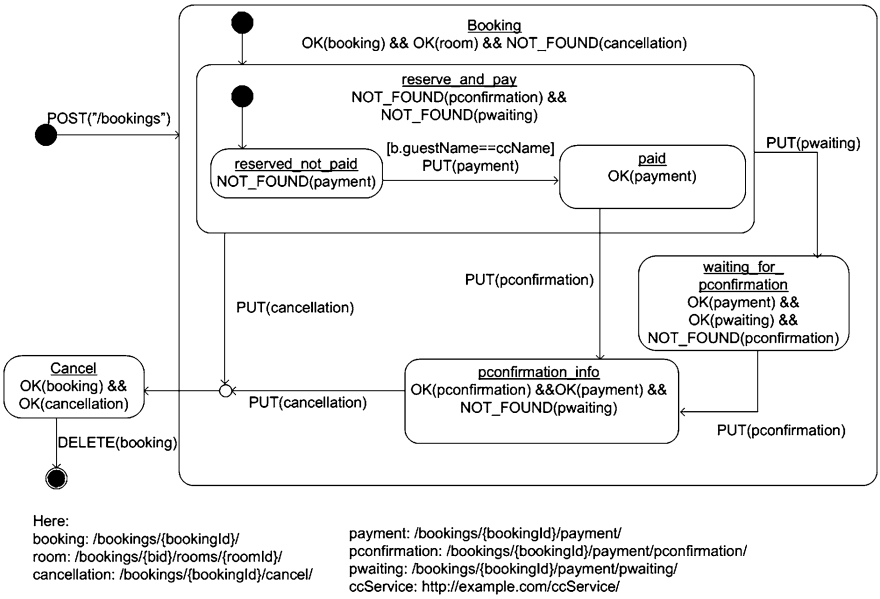

Fig. 5.2 Behavioral model for HRB RESTful web service

only a POST (side-effect) method can be invoked on collection resource *bookings*, similarly allowed (side-effect) method on resource *booking*, *payment*, *pwaiting*, *pconfirmation* and *cancel* is PUT. On *booking* resource, a DELETE method can also be invoked.

The guards and postconditions on transitions are defined only using GET requests on request on resources and the request parameters that include values parsed out of the request URI. A guard condition on the transition specifies the condition required to invoke an HTTP method on a resource. For example, consider guard *[b.guestName==ccName]* for the method *PUT(payment)* in Fig. 5.2, where *b* refers to the relative navigation path to resource *booking*. This guard specifies that the PUT method on *payment* resource can be invoked only if the guestName in resource representation of *booking* for booking Id {*bookingId*} matches the name of the credit card provided by the client.

State Invariants Using Resources

State invariants show the current state of an application during the lifecycle of an object. We are representing behavioral interface of a REST web service using protocol state machines. REST invocations do not contain any state or session information, so defining state invariants for REST application states is not obvious.

We address this problem by performing GET requests on different resources and using their representations and response codes to form boolean expressions.

When we invoke an HTTP GET method on a resource, it returns its representation along with the HTTP response code. This response code tells whether the request went well or bad. If the HTTP response code is 200, this means that the request was successful and the referred resource exists. Otherwise, if the response code is 404, this implies that URI could not be mapped to any resource and the referred resource does not exist. We do not treat this 404 code as an error but as an important determinant of protocol state.

We use a boolean function *OK(r)* to express that the response code of HTTP GET method on a resource *r* is 200. Similarly, the boolean function *NOT_FOUND(r)* is true when the response code of HTTP GET method on resource *r* is 404. These boolean functions on the resources along with the attributes that represent a resource are used to define a state invariant in our RESTful behavioral model.

For example, consider the state invariant for the state *reserved_not_paid* in Fig. 5.2. *NOT_FOUND(payment)* checks the response code for the HTTP GET method on the resource *payment*. It evaluates to true if response code of GET method on *payment* for a particular booking ID({bid}) is 404. For the HRB service to be in state *reserved_not_paid*, the state invariant of this simple state is conjuncted with the state invariants of all the states that contain it, i.e., $NOT\_FOUND(payment) \&\& NOT\_FOUND(pconfirmation) \&\& NOT\_FOUND(pwaiting) \&\& OK(booking) \&\& OK(room) \&\& NOT\_FOUND(cancellation)$.

Synchronous and Asynchronous Web Services

Interaction between web services can be either synchronous or asynchronous. This interaction is distinguished in the manner request and response are handled. When a client invokes a synchronous services, it suspends further processing until it gets a response from the service. On the other hand, when a client invokes an asynchronous service it does not wait for the response and continues with its processing. The asynchronous service can respond later in time. The client receives this response and continues with its processing.

We have modeled the scenario for both the synchronous and asynchronous third party service in Fig. 5.2. In case of interaction with an asynchronous service, we create a *waiting* state in our state machine. In Fig. 5.2, a third party credit card payment service is invoked when a PUT is invoked on the payment resource. This would invoke CCService as an internal action. If CCService is an asynchronous service, then it may take a long time to process the credit card and confirm the payment back to the client. Thus, the system goes into a wait state for the particular booking with booking ID {bookingId} and resumes processing of other transactions. When a response on payment confirmation is given by the third party service, the processing for this booking is resumed.

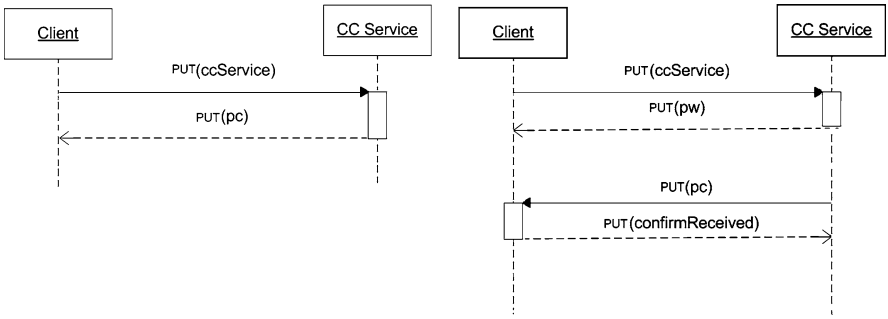

Fig. 5.3 (*Left*) Interaction with Synchronous CC Service. (*Right*) Interaction with Asynchronous CC Service

For synchronous service, there is no need for a *waiting* state since the service does not take long to respond and system can continue with its processing after receiving the response. This is shown in Fig. 5.2 by a direct transition from *paid* state to *pconfirmation_info* state with PUT(pconfirmation) as a trigger.

The two scenarios showing the request and response behavior in synchronous and asynchronous services is shown in Fig. 5.3. The left side shows the scenario in which credit card(CC) verification service is synchronous and on the right hand side show interaction with an asynchronous CC verification service. It may be worth pointing out that the agent PUTing the payment (the client) must also be able to act as a server in order to receive a PUT payment confirmation. As an alternative, the CCService might return 202(Accepted) response with location. This would require the client to poll for confirmation.

Stateless State Machines

We have used state machines to model the stateless behavior of REST web service. Using a state machine to model a stateless interface may seem an oxymoron. In the context of a RESTful service, statelessness is interpreted as the absence of hidden information kept by the service between different service requests. In that sense, a RESTful web service should exhibit a stateless protocol. Also, there is no sense of session or sequence of request in a true RESTful service.

On the other hand, state machines have a notion of active state configuration, that is, what states are active at a certain point of time. If an implementation of an interface described using a state machine would have to keep the active state configuration between different requests, then this would break the statelessness requirement of the RESTful service.

It is notable that the behavioral modeling described above, does not actually require that a service implementation keeps any additional protocol state. In our

approach a state is active if its invariant evaluates to true, but the invariants are defined using addressable application resources. Therefore, an implementation of a service can determine the active state configuration by querying the application state. There is no need to keep any additional protocol state.

Determining what is the active state configuration of the interface state machine every time that a service implementation has to fulfill a request may be a slow task in the case of complex interfaces with many states. However, in practice it is not necessary to explore all states in the state machine but only the source states of the transitions that can be triggered based on the current request. We show in the next section how we can do that by computing the precondition (and postcondition) of each method request.

Service Preconditions and Postconditions

In this section, we show how to extract the contract information from a UML protocol state machine with state invariants. The contract contains the precondition and postcondition for each method that triggers a transition in the behavioral model.

The precondition of a method states under what conditions a method can be triggered. We say that the precondition of a method *m* is satisfied when the state invariants of all the source states of transition *t* are true along with its guard condition.

In a similar manner, if a method *m* triggers a transition *t* in a behavioral model, then its post-condition is satisfied when the state invariants of all the target states of transition *t* are true along with the postcondition annotated on the transition *t*.

In order to shorten the description of the contract we use path variables to represent the address of a resource. First, the precondition for a method that triggers a transition in the behavioral model is presented. The precondition of a method *m* is given by taking into account all the transitions that are triggered by *m*. If it is a simple transition, then the state invariant of its source state is conjuncted with the guard of the transition. In case the transition is a trigger to more than one transition, with true guards, and all the transitions have different source states, then the precondition is given by taking a disjunction of state invariants of all the different source states. This implies that the method can trigger a transition whenever it is in one of its source states.

A transition can occur from one state to another if the method that triggers this transition is invoked and its precondition is true. For the transition to be successful, the postcondition of the transition should also be true after the method is invoked. This is specified by the *implication* operator that relates a precondition of a transition with its postcondition.

A postcondition for a method is extracted from the protocol state machine by manipulating the state invariants of the target states of transitions and the post-conditions on transitions. The post-condition of a fork transition, with true

postcondition, specifies that the state invariants of all its target states are true and for a self-transition, its post-condition ensures that the same state invariants are true that were true before invoking the HTTP method.

For the details and formal definitions of generating preconditions and postconditions for different elements in a UML protocol state machine of a class readers are referred to Porres and Rauf (2010).

The postcondition of a transition will be evaluated only if the precondition for that transition is true. We define as *pre_OK(r)* the function that gives boolean value of *OK(r)* on resource *r* before invoking the trigger method. Similarly, *pre_b.guestName* and *pre_NOT_FOUND(r)* give the representation of *booking* and boolean value of *NOT_FOUND(r)* before invoking the trigger method, respectively.

The excerpt below from the list of high-level contracts generated from Fig. 5.2 shows the contracts generated for the HTTP method PUT on *payment* resource.

```
PATH
 b: bookings/{bid}/
 r: bookings/{bid}/rooms/
 p: bookings/{bid}/payment/
 pc: bookings/{bid}/payment/pconfirmation/
 pw: bookings/{bid}/payment/pwaiting/
 c: bookings/{bid}/cancel/

PUT {bookings/{bid}/payment/}
 precondition
  ((OK(b) && OK(r) && NOT_FOUND(c)) &&
  (NOT_FOUND(pc) && NOT_FOUND(pw)) && NOT_FOUND(p) &&
     [b.guestName == ccName])

 postcondition
  ((pre_OK(b) && pre_OK(r) && pre_NOT_FOUND(c)) &&
  (pre_NOT_FOUND(pc) && pre_NOT_FOUND(pw)) && pre_NOT_FOUND
      (p) && [pre_b.guestName == ccName]) ==>  ((OK(b) && OK
      (r) && NOT_FOUND(c)) &&
  (NOT_FOUND(pc) && NOT_FOUND (pw))&& OK(p))
```

The conceptual model as shown in Fig. 5.1 and behavioral model as show in Fig. 5.2 are implemented as REST web services. This is explained further in the next section.

Implementation of a Service Using the Django Framework

Django is a web framework that makes it easy to develop web applications and web services in Python. At a glance, Django can be understood with its three basic files that support separation of concerns, i.e. models.py, urls.py and views.py where models.py contains descriptions of database tables, views.py contains the business logic, and urls.py specifies which URIs map to which view. For a

```
from django.db import models

class room(models.Model):
    rType = models.CharField(max_length=200)
    floor = models.IntegerField()

class guest(models.Model):
    fName = models.CharField(max_length=200)
    phone = models.IntegerField()
    email =  models.CharField(max_length=200)

class booking(models.Model):
    bDate =  models.DateTimeField()
    cancel = models.BooleanField(default=False)
    cancel_note = models.CharField(max_length=500)
    room = models.ForeignKey(room)
    gName = models.CharField(max_length=500)

class payment(models.Model):
    amount = models.FloatField()
    pDate = models.DateTimeField()
    confirm = models.BooleanField(default=False)
    waiting = models.BooleanField(default=False)
    p_try = models.IntegerField(default = 0)
    ccName = models.CharField(max_length=500)
    booking = models.ForeignKey(booking)
```

Listing 5.1 Implementation of Database Models for HRB Service

detailed working of Django Framework, readers are encouraged to read Django Documentation (Django Software Foundation 2010) and Django Book (Holovaty and Kaplan-Moss 2010).

The design approach we have used to design REST web services in this chapter can be easily implemented in Django. In this section, we show how this implementation is done. We carry forward the example of HRB service demonstrated above and show its implementation procedure.

The main steps in our implementation phase are:

- Implement database tables in models.py
- Create views for each resource and its transitions in views.py
- Map relative URIs from resource model to respective views in urls.py.

As a first step, the database tables are specified in models.py. The database tables we have created are shown in Listing 5.1.

In the second step, for each resource, shown in the conceptual resource model, a view is defined. The information on *allowed* and *not-allowed* methods is retrieved from behavioral model. The incoming request to the view is verified against the allowed methods and redirected to the view that supports the request method for the resource.

```
def booking_payment(request, bid):
    if not request.method in ["GET", "PUT"]:
        return HttpResponseNotAllowed(["GET", "PUT"])
    if request.method == "GET":
        bid = bid
        return booking_payment_get(request, bid)
    if request.method == "PUT":
        bid = bid
        amnt = request.POST.get('amnt')
        ccName = request.POST.get('ccName')
        return booking_payment_put(request, bid, amnt, ccName
            )

def booking_payment_get(request, bid):
       p = payment.objects.filter(booking=bid)
       if p:
           json = serializers.serialize("json", p)
           return HttpResponse(json, mimetype="application/
               json")
       else:
           return None

def booking_payment_put(request, bid, amnt, ccName):
       b = booking_detail_get_local(bid)
       r = room_detail_get_local(bid)
       c = booking_cancel_get_local(bid)
       p = booking_payment_get_local(bid)
       pc = booking_pconfirmation_get_local(bid)
       if not p:
          pre_p = False
       else:
          pre_p = True
       deserialized = serializers.deserialize("json", b)
       b_detail = list(deserialized)[0].object
       a = []
       for field in ["bDate", "cancel", "cancel_note", "room"
           , "gName"]:
           new_val = getattr(b_detail, field, None )
           a.append(new_val)
       if  b and r and not p and not pc and not c and a[4]==
           ccName:
           now = datetime.datetime.now()
           cc = ccName
           a = amnt
```

Listing 5.2 Payment View

The first view *booking_payment*(*request*, *bid*) in Listing 5.2 shows implementation of *payment* resource. The behavioral model in Fig. 5.2 shows that the allowed methods for this resource are GET and PUT. These two methods are listed in the

```
            p = payment(confirm=False, pDate=now, waiting=
                False, amount=a, p_try=0, ccName = cc,
                booking_id=bid)
            p.save()
        b = booking_detail_get_local(bid)
        r = room_detail_get_local(bid)
        c = booking_cancel_get_local(bid)
        pc = booking_pconfirmation_get_local(bid)
        post_p = booking_payment_get_local(bid)
        if  b and r and not pre_p and post_p and not pc and
            not c:
            response = HttpResponse("created")
            response.status_code = 201
            return response
        else:
            response = HttpResponse("not created")
            response.status_code = 406
            return response
```

Listing 5.2 (continued)

```
urlpatterns = patterns('',
                       (r'^bookings/$', collection_bookings),
                       (r'^bookings/(\d{1,3})/$',
                           booking_detail),
                       (r'^rooms/$', collection_rooms) ,
                       (r'^bookings/(\d{1,3})/rooms/$',
                           room_detail),
                       (r'^bookings/(\d{1,3})/payment/$',
                           booking_payment),
                       (r'^bookings/(\d{1,3})/payment/waiting
                           /$', booking_waiting),
                       (r'^bookings/(\d{1,3})/payment/
                           pconfirmation/$',
                           booking_pconfirmation),
                       (r'^bookings/(\d{1,3})/cancel/$',
                           booking_cancel),
)
```

Listing 5.3 Relative URIs and views mapping for HRB Service

list of allowed methods in *booking_payment* view and each incoming request to this view is first verified to be one of these methods, otherwise an HTTP response of method not allowed is given.

In the third step, the relative URIs shown in the conceptual resource model are mapped to the respective views. Every resource in our conceptual model is addressable. We can get the relative URI for each resource directly from Fig. 5.1 that is then mapped to the respective views as show in Listing 5.3.

Users can use cURL to invoke URIs specifying the methods they want to invoke on the service. cURL is a command line tool that is a capable HTTP client and

supports most of HTTP methods, authentication mechanisms, headers etc. (cURL 2010). For invoking a POST method on *payment* resource with *amnt* value, on local server, the following command can be used on cURL:

$$curl - X\ PUT - d\ amnt = 115 - d\ ccName = '' Thomas'' http : //127.0.0.1 : 8000/\ bookings/3/payment/$$

Now lets look in detail on the implementation of views. A separate view is implemented for each of the allowed methods on each resource. Once a view related to a specific URL is called, it further redirects the control to the view that corresponds to the invoked HTTP method.

As an example, we are only looking into the *payment* resource and its allowed methods in Listing 5.2. The allowed methods on *payment* resource are GET and PUT as specified in Fig. 5.2. When the client invokes */bookings/13/payment/*, control is passed to *booking_payment* view. This view verifies the input method and if the request method is neither GET nor PUT, an HTTP not allowed response is given. If the method is GET or PUT on *payment*, the client is redirected to *booking_payment_get*(*request*, *bid*) view or *booking_payment_put*(*request*, *bid*, *amnt*, *ccName*) view, respectively.

The GET view, i.e., *booking_payment_get*(*request*, *bid*), queries the database, retrieves the payment information for booking id 13 and returns it as a JSON object. If there is no booking record with id 13, then a response code of 404 is returned.

The PUT view creates the specific resource and returns a successful HTTP response method. When the client invokes */bookings/13/payment/* with PUT method, the control goes to *booking_payment_put*(*request*, *bid*, *amnt*, *ccName*) view and a payment record is entered for booking with id 13.

However, if a payment record is already present for this booking id, then the operation of inserting additional record in payment table should not be executed. Such rich behavioral specifications are present in the behavioral model and earlier in "Service Preconditions and Postconditions" we saw how preconditions and postconditions of methods can be generated from this model. We now detail how these behavioral specifications are inserted for methods in Django Web Framework.

The pre-condition of a method is extracted from the state machine by manipulating the state invariants of all the source states and guard on the transition. Likewise, a post-condition is extracted by manipulating the state invariants of all the target states and post condition on the transition.

When a method with side-effects, i.e. PUT, POST or DELETE is called on a resource, we need to extract the current state of different resources to check whether the conditions to invoke the method are satisfied. In a similar fashion, we have to check the status of different resources to ensure that desired effect is created before returning the client a success message. By current state we mean the presence or absence of a resource or values of its attributes at the time of invoking certain method.

In Django, we extract the current state of resources by calling the view that maps to GET request on the resource. However, to take advantage of relative URI mechanism and to reduce the number of HTTP calls, local GET views are

```
def booking_payment_get_local(bid):
        p = payment.objects.filter(booking=bid)
        if p:
            data = serializers.serialize("json", p)
            return data
        else:
            return None
```

Listing 5.4 Excerpt of Local GET View on 'payment' for HRB Service

implemented for each resource. The local GET views retrieve information from the database and return them as normal objects rather than as HTTP response objects. An implementation of local GET view on *payment* resource is shown in Listing 5.4.

The pre and post conditions are asserted in each of the views that correspond to the methods that trigger a transition in state machine. Listing 5.2 shows how pre and post conditions are asserted for PUT method on *payment*. Information of the resources that form the state invariant of source states and guard condition is stored in different variables. These variables are combined as a boolean expression and asserted as an *if* condition before performing the desired task. Similarly, before giving a success response to the client, a local get is performed on the resources that make the state invariant of target states and transition's post conditions. Only if the expected behavior is observed, a success response is given to the client.

Implementation of a Service Monitor

A service monitor can be used to continuously verify the functionality of an implemented web service. This monitoring mechanism can keep a check on the behavior of both the client and the provider. The client is checked for invocation to the service under right conditions and the provider of the service is constraint to provide the implementation as specified.

The monitoring mechanism can be implemented in Django by using the rich behavioral information present in our state machine. The service monitor is implemented as a service proxy. It listens for requests from the client, verifies the conditions to invoke the method and then forward it to the actual service implementation.

The behavioral model provides a rich behavioral interface that can be published with the service as a specification. This gives information about the conditions in which a method should be invoked on its interface and also about its expected conditions. This specification of a service interface can be used to build a proxy interface to test the functionality of that service and to invoke the service in right conditions.

```
def booking_payment_get(request, bid):
    print "booking payment get"
    req = urllib2.Request('http://127.0.0.1:8000/bookings/%s/
        payment/' % bid)
    try:
        response = urllib2.urlopen(req)
        the_page = response.read()
        return HttpResponse(the_page)
    except:
        return HttpResponse(status=404)
```

Listing 5.5 Excerpt of GET view in Proxy Interface

```
def booking_payment_put(request, bid, amnt, ccName):
   b = booking_detail_get(request, bid)
   r = room_detail_get(request, bid)
   c = booking_cancel_get(request,bid)
   p = booking_payment_get(request, bid)
   pc = booking_pconfirmation_get(request, bid)
   pw = booking_pconfirmation_get(request, bid)
   if not p.status_code == 200:
       pre_p = False
   else:
       pre_p = True
   if  b.status_code = 200 and r.status_code == 200 and p.
       status_code == 404 and pc.status_code == 404 and pw.
       status_code == 404 and c.status_code == 404:
       values ={'amnt': 33, 'ccName': 'Thomas'}
       mydata = urllib.urlencode(values)
       opener = urllib2.build_opener(urllib2.HTTPHandler)
       request = urllib2.Request('http://127.0.0.1:8000/
           bookings/%s/payment/' % bid, data=mydata)
       request.add_header('Content-Type', 'your/contenttype')
       request.get_method = lambda: 'PUT'
       url = opener.open(request)
   else:
       return = HttpResponse(status=404)
   post_p = booking_payment_get(request, bid)
   if  b.status_code = 200 and r.status_code == 200 and pc.
       status_code == 404 and pw.status_code == 404 and c.
       status_code == 404 and not pre_p and post_p.status_code
        == 200:
       return HttpResponse(the_page,status=201)
   else:
       return HttpResponse("not created",status=406)
```

Listing 5.6 PUT Method on Payment in the Proxy Interface

In this section, we show how we have implemented a proxy interface for HRB service detailed above. In proxy interface, a method is implemented for each of the methods that are invoked on the REST web service interface using urllib2. urllib2 is a Python module that is used to fetch URLs (urllib2 extensible library for opening URLs 2010). In a proxy interface for HRB service, a GET method on payment resource is implemented as shown in Listing 5.5.

Each GET view returns an HTTP response object. When a POST, PUT or DELETE method is implemented in the proxy interface, it manipulates the status codes of the HTTP response objects and asserts them as method pre and post conditions. An excerpt of HRB proxy interface that shows a PUT method on the payment resource is given as shown in Listing 5.6.

Conclusions

RESTful web services can be used in rich services that go beyond simple operations of creating, retrieving, updating, and deleting data from the database. These rich services should also offer interface that would exhibit REST features of uniform interface, addressability, connectedness, and statelessness. In this chapter, we discuss the design methodology that creates RESTful web services by construction. The approach uses UML class diagram and state machine diagram to represent the structural and behavioral features of a REST web service. The conceptual resource model that represents the structural feature adds addressability and connectivity features to the designed interface. The uniform interface feature is offered by constraining the invocation methods in the state machine to HTTP methods. In addition, to provide the feature of statelessness in our interface we use a state machine for behavioral modeling. This oxymoron is addressed by taking advantage of the fact that state invariants can be defined using query method on resources and the information contained in their response codes.

The rich behavioral specifications present in the behavioral model show the order of method invocations and the conditions under which these methods can be invoked along with the expected conditions. We use this behavioral model to generate contracts in the form of preconditions and postconditions for methods of an interface.

The design approach is implemented in Django web framework and the contracts generated from the behavioral model are asserted as contracts in the implemented interface.

A proxy interface is also implemented in Django as a service monitor. This service monitor can also be implemented for services that are already implemented and only provide their behavioral specifications in natural language or in any other form. A service monitor can continuously verify functionality of a service and reports if a service user violates a precondition or the implementation does not provide the expected behavior.

References

cURL. 2010. http://curl.haxx.se/.

urllib2 extensible library for opening URLs. *Python Documentation*, 2010. http://docs.python.org/library/urllib2.html.

Django Software Foundation. Django Documentation. *Online Documentation of Django 1.2*, 2010. http://docs.djangoproject.com/en/1.2/.

A. Holovaty and J. Kaplan-Moss. The Django Book. *Online version of The Django Book*, 2010. http://docs.djangoproject.com/en/1.2/.

T. Mens and P. V. Gorp. A Taxonomy of Model Transformation. *Proceedings of the International Workshop on Graph and Model Transformation*, 2005.

I. Porres and I. Rauf. From uml Protocol Statemachins to Class Contracts. *Procceedings of the International Conference on Software Test, Verification and Validation(ICST 2010)*, 2010.

I. Porres and I. Rauf. Modeling Behavioral RESTful Web Service Interfaces in UML. *Accepted for Publication in 26th Annual ACM Symposium on Applied Computing Track on Service Oriented Architectures and Programming (SAC 2011)*, 2011.

OMG UML. 2.2 Superstructure Specification. *OMG ed*, 2009. http://www.omg.org/spec/UML/2.2/.