Chapter 6
Advanced SOA Patterns for Building Information Models

Abstract Two styles of Web services exist today: Simple Object Access Protocol (SOAP) and REST. Representational State Transfer (REST) is often preferred over the more heavyweight SOAP because REST does not leverage as much bandwidth. REST's decoupled architecture makes it a popular building style for cloud-based APIs, such as those provided by Amazon, Microsoft and Google. This chapter starts with providing technical information about RESTful Web services. Following this, RESTful design patterns for facilitating BIM-based software and Web service architectures are presented.

6.1 Introduction

As indicated in Isikdag and Underwood (2009) Web services can be defined as components and resources that can either be invoked over the Web or reached by standard Web protocols using messages. Exposing a Web service mostly involves enabling the older software (or the data layer of the legacy system) to receive and respond to web message requests for its functionality (Pulier and Taylor 2006). He (2003) indicated that two constraints exist for implementing the Web services: (i) interfaces must be based on Internet protocols such as HTTP, FTP and SMTP and (ii) except for binary data attachment, messages must be in XML. Two definitive characteristics of Web services are mentioned as loose coupling and network transparency (Pulier and Taylor 2006). As explained by the authors, in a traditional distributed environment computers are *tightly coupled*, i.e. each computer connects with others in the distributed environment through a combination of proprietary interfaces and network protocols. Web services in contrast, are *loosely coupled*, i.e. when a piece of software has been exposed as a Web service, it is relatively simple to move it to another computer. On the other hand, as Web services' consumers and providers send messages to each other using open Internet protocols, Web services offer total *network transparency* to those that employ them (Pulier and Taylor 2006). Network transparency refers to a Web service's capacity

U. Isikdag, *Enhanced Building Information Models*,
SpringerBriefs in Computer Science, DOI 10.1007/978-3-319-21825-0_6

to be active anywhere on a network or group of networks without having any impact on its ability to function. As each Web service has its own Universal Resource Indicator (URI), Web services have similar flexibility to web sites on the Internet. Two styles of Web services exist today: SOAP and REST. Simple Object Access Protocol (SOAP) is a protocol for exchanging XML formatted messages between the networks using Hypertext Transfer Protocol (HTTP). The SOAP protocol has two constraints: (i) except for binary data attachment, messages must be carried by the SOAP protocol and formatted by its rules and (ii) the description (exposed interface) of the service must be defined in the Web Services Description Language (WSDL). The terms REST and RESTful Web services have been coined after the PhD dissertation of Fielding (2000). As explained by Pautasso et al. (2008), REST was originally introduced as an architectural style for building large-scale distributed hypermedia systems. According to REST style, a Web service can be built upon resources (i.e. anything that is available digitally over the Web), their names (identified by uniform indicators, i.e. URIs) representations (i.e. metadata/data on the current state of the resource) and links between the representations. As mentioned by Techtarget (2015), REST is often preferred over the more heavyweight SOAP style because REST does not leverage as much bandwidth. REST is often used in mobile applications, social networking web sites, mashup tools and automated business processes. REST's decoupled architecture makes it a popular building style for cloud-based APIs, such as those provided by Amazon, Microsoft and Google. When Web services use the REST architecture, they are called RESTful Application Programming Interfaces (APIs) or REST APIs. REST decouples information consumers from producers and supports stateless existence where the server does not store the number or state of the clients consuming a Web service. The number of operations is limited with the methods provided by the HTTP protocol. GET, PUT, POST and DELETE are the most commonly used methods (i.e. the number of verbs to be used is limited). In fact, the hierarchical naming convention on REST APIs provides flexibility where each RESOURCE is identified by a universal resource identifier (URI) which is defined by nouns. REST also has downsides. In the world of REST, as mentioned by Techtarget (2015), there is no direct support for generating a client from server-side-generated metadata. SOAP is able to support this with WSDL. In fact, RESTful APIs have the advantage of exchanging plain messages, while SOAP uses XML envelopes, which make the message size larger and message transfer less efficient. Most of the Web services currently implemented have transformed from interfaces of the legacy systems.

6.2 REST in a Nutshell

In order to explain the RESTful architectures it would be good to start from a simple web page request. Once a request is made from a web browser by the user to view a page (such as www.google.com) a GET request is issued according to the

HTTP protocol. A GET request in the HTTP protocol is a request where you can pass a URI including some parameters. For instance,

http://www.somewebsite.org/showbuildingpart.php?building_id=1254&floor_id=12

is a valid GET request with parameters. In fact, the HTTP protocol offers just more than the GET request. Another commonly used request in the web browsers is the POST request. The POST request sends information to a URI but as opposed to the GET request does not include parameters as a part of the request. In a POST request, information that is required by the receiving end is sent within the HEAD or BODY of the message. In fact, the HTTP protocol capabilities are not limited to the GET and POST, but the HTTP protocol is able to send a PUT request to update the REPRESENTATION of a web RESOURCE and DELETE request to delete a resource. There are also other HTTP methods such as HEAD, TRACE, PATCH, CONNECT or OPTIONS, but these will not be elaborated here. The REST architectural principles indicate that the information on the Web can be managed using the HTTP methods, in a similar way that one can manage the Create/Read/Update/Delete (CRUD) operations for a data resource. In a RESTful architecture HTTP methods GET, POST, PUT and DELETE are used to make CRUD-like operations over the web RESOURCES. According to the REST architectural style a Web service can be built by utilizing …

- RESOURCES (i.e. anything that is available digitally over the Web),
- IDENTIFIERS (i.e. URIs),
- REPRESENTATIONS (i.e. current state of the resources).

In a RESTful architecture RESOURCES, REPRESENTATIONS and IDENTIFIERS can be described as below:

- RESOURCE → is a logical object identified by an IDENTIFIER.
- IDENTIFIER → A globally unique ID that points to the RESOURCE.
- REPRESENTATION: Physical source of information that is pointed by the IDENTIFIER.
- A RESOURCE can have multiple representations but a single IDENTIFIER which can only point to a single REPRESENTATION of a RESOURCE at a single point in time.

Once a RESOURCE is created in a RESTful architecture it is constant until it has got DELETED. The variability is on the REPRESENTATION, as the REPRESENTATION can be UPDATED using the HTTP protocol methods. The acronym REST stands for Representational State Transfer. In a RESTful architecture a REPRESENTATION existent in one STATE of a RESOURCE is TRANSFERRED to a web client and this causes the change in the STATE of the web client. It is vital to mention here that the SERVER is STATELESS in RESTful architectures, which means that the CLIENT STATE is not known or maintained by the SERVER, which provides great efficiency in RESTful architectures. Table 6.1 summarizes valid server-side RESOURCE REPRESENTATIONS by examples.

Table 6.1 Resource representations (server side)

Resource	Identifier	Representation	Time	Resource state
Building footprint	http://www.xyz.com/building/0001/footprint	footprint.jpg	$t = 0$	s = i
Building footprint	http://www.xyz.com/building/0001/footprint	footprint.csv	$t = 1$	s = ii
Building footprint	http://www.xyz.com/building/0001/footprint	footprint.dwg	$t = 2$	s = iii
Building facade	http://www.xyz.com/building/0001/facade	façade.jpg	$t = 3$	s = i
Building facade	http://www.xyz.com/building/0001/facade	façade.dwg	$t = 4$	s = ii
(Deleted) building facade			$t = 5$	s = null

An example sequence of client interaction in a RESTful architecture is provided in Table 6.2. As indicated in Table 6.2.

1. client states (s = 1, 3, 5, 7, 9, 11, 13, 15, 17, 19) are REQUEST states where the client makes an HTTP REQUEST to a RESTful Service,
   (a) to GET REPRESENTATION of a RESOURCE by an IDENTIFIER (i.e. URI)
      i. using HTTP GET (s = 1, 5, 9, 13, 17)
   (b) to UPDATE REPRESENTATION of a RESOURCE by an IDENTIFIER
      ii. using HTTP PUT (s = 3, 7, 15)
   (c) to CREATE a new RESOURCE by POSTING a REPRESENTATION of it with an IDENTIFIER
      iii. using HTTP POST (s = 11)
   (d) to DELETE a RESOURCE by an IDENTIFIER
      iv. using HTTP DELETE (s = 19)
2. client states (s = 2, 6, 10, 14, 18) are the VIEW states where the information transferred from the server is shown by the client.
3. client states (s = 4, 8, 12, 16, 20) are the ACKNOWLEDGEMENT states where the ACKNOWLEDGEMENT on method success or failure sent by the server are shown by the client.
4. in the HTTP REQUEST states the client view stays on the previous state until the request is fulfilled (e.g. in web modern browsers this is the reason that we continue to see a web page until the next page is loaded).

Table 6.2 Client-side interactions within a RESTful architecture

Client state	Client HTTP REQUEST	Client view	Data sent by client
$s = 1$	GET http://www.xyz.com/building/0001/footprint	Nothing	Nothing
$s = 2$	Nothing	footprint.jpg	Nothing
$s = 3$	PUT http://www.xyz.com/building/0001/footprint	footprint.jpg	footprint.csv
$s = 4$	Nothing	HTTP 200	Nothing
$s = 5$	GET http://www.xyz.com/building/0001/footprint	HTTP 200	Nothing
$s = 6$	Nothing	footprint.csv	Nothing
$s = 7$	PUT http://www.xyz.com/building/0001/footprint	footprint.csv	footprint.dwg
$s = 8$	Nothing	HTTP 200	Nothing
$s = 9$	GET http://www.xyz.com/building/0001/footprint	HTTP 200	Nothing
$s = 10$	Nothing	footprint.dwg	Nothing
$s = 11$	POST http://www.xyz.com/building/0001/facade	footprint.dwg	façade.jpg
$s = 12$	Nothing	HTTP 201	Nothing
$s = 13$	GET http://www.xyz.com/building/0001/facade	HTTP 201	Nothing
$s = 14$	Nothing	façade.jpg	Nothing
$s = 15$	PUT http://www.xyz.com/building/0001/facade	façade.jpg	façade.dwg
$s = 16$	Nothing	HTTP 200	Nothing
$s = 17$	GET http://www.xyz.com/building/0001/facade	HTTP 200	Nothing
$s = 18$	Nothing	façade.dwg	Nothing
$s = 19$	DELETE http://www.xyz.com/building/0001/facade	façade.dwg	Nothing
$s = 20$	Nothing	HTTP 200/ HTTP 204	Nothing

5. An ACKNOWLEDGEMENT of a successful GET request is usually not shown to the users, instead the client state is transferred to the VIEW state where users see the REPRESENTATION of the RESOURCE.
6. Regular HTTP clients such as web browsers do not show HTTP ACKNOWLEDGEMENTS, a RESTful service client is required to observe all states provided in Table 6.2.

As it can be noticed from Tables 6.1 and 6.2 the RESTful requests utilize a hierarchical URI structure. As explained by the RESTful API Tutorial (2015), in addition to utilizing the HTTP verbs appropriately, resource naming is arguably the most debated and the most important concept to grasp when creating an understandable, easily leveraged Web service API. When resources are named well, an API is intuitive and easy to use. Generally a RESTful API is a collection of URIs. A RESTful URI should refer to a resource which is a thing instead of referring to an action. URIs should follow a predictable, hierarchical structure to enhance usability;

hierarchical in the sense that data has structure relationships. This is not a REST rule or constraint, but it enhances the API. If we go back to our first example, URI called with an HTTP GET request

http://www.somewebsite.org/showbuildingpart.php?building_id=1254&floor_id=12

has its counterpart URI in a RESTful API which implies a hierarchical naming convention would be one of the following:

http://www.somewebsite.org/building_id/1254/floor_id/12
http://www.somewebsite.org/building/1254/12

REST is a key architectural style in enabling interaction with different layers of data. As one of the complex information models, BIM would benefit much from RESTful data interchange, RESTful interactions and RESTful APIs. The following sections will present advanced SOA/RESTful patterns to facilitate interaction with BIMs. The patterns presented in the following sections are RESTful, utilize REST for information interchange, the services presented in the patterns are RESTful APIs and it is assumed that a service consumer API is present and facilitates interaction between the service itself and the service client(s).

6.3 Generalized Design Pattern for BIM-Based SOA

The patterns explained in the following sections define software architectures that consist of several layers. Although the patterns are defined for different purposes, the software layers defined in these patterns have common characteristics. In order to prevent repetition of these characteristics for each pattern, the common characteristics of the architectural layers are illustrated in Fig. 6.1 and are explained below.

1. **Data Layer**: The patterns describe architectures where an extended BIM, a BIM or a BIM view resides in an object or an object relational database (which are shown with database symbols in the pattern illustrations). These databases are the persistence environments where the persistent versions of the BIMs reside. In some patterns the BIM instances are persisted in the form of a BIM file which is encoded as an ISO10303 P-42 ASCII file or an XML file (which are shown as file symbols in pattern illustrations). The databases reside in a database server or in cloud database server hardware. The file resides in a web server or a cloud web server. These data stores together form the first layer of the architecture.
2. **Database/File API Layer**: The database/file API forms the second layer of the architecture. The API acts as a database/file interface that will be used to query and interact with the object/object-relational databases or the file, based on requests coming from the run-time objects (i.e. the upper layer). For IFC BIMs the variations of SDAI-based APIs and XML APIs can be used in this layer. The database/file API can reside within the same hardware which the database

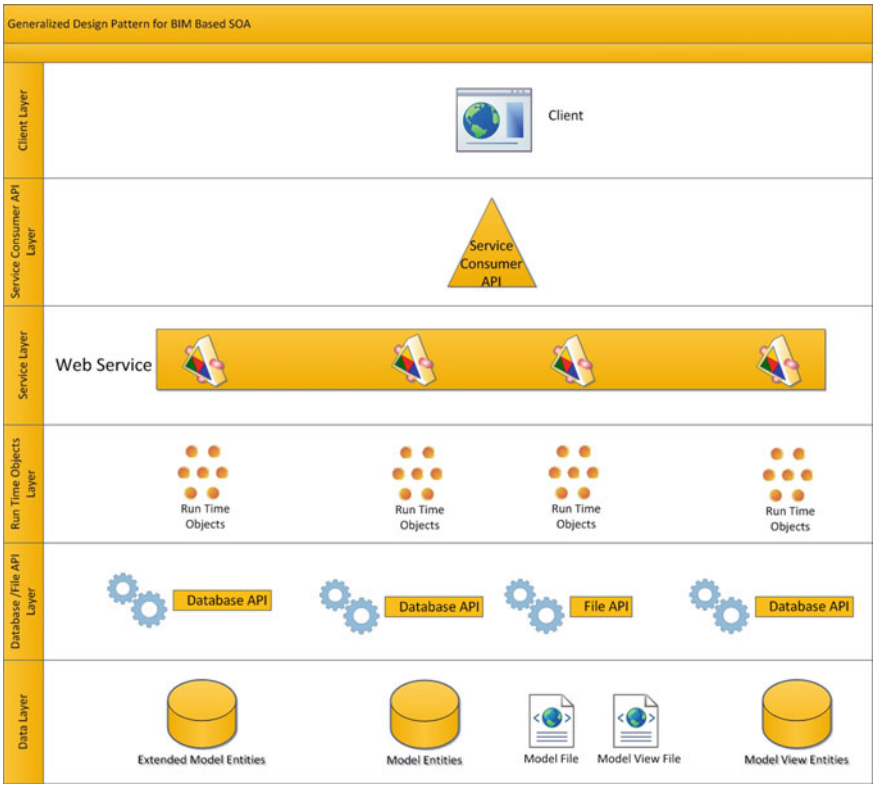

Fig. 6.1 Generalized design pattern diagram

operates (i.e. in the database/cloud database server) or which the file resides in, but it can also reside in a different server.

3. **Run Time Objects Layer**: The run time objects form the third layer of the architecture and maintain the (i) transient data objects that are generated as a result of service requests/interaction and (ii) transient objects of the business (or pattern) logic which the service designer chooses to utilize in this layer. The communication of this layer is bidirectional (i.e. would either be for information provision to the service or for updating the data layer based on information provided by the service) and the direction of communication depends on the type of REQUESTs that are directed to the Web service. The object container can be one of Common Language Infrastructure of the .NET cross-language cross-platform object model, EJB contained within the Java VM. The service designer can choose to distribute the business logic between the service objects and the run time objects. The run time objects layer will be tightly coupled with the database/file APIs and the service. The ratio of objects maintained in the run time objects layer to objects maintained in the service layer would be defined by

the choice of the software architect. Both the run time objects layer (i.e. the object container) and the service layer can be located in the same hardware or in different hardware components/servers, including real and virtual servers. The number of predicated requests/interactions between these layers per time frame (i.e. hour/day/week etc.) and the network bandwidth required for communication between these layers will be the main determinants for this decision.

4. **Service Layer**: This layer consists of service component(s) that provide the RESTful interface. This is the core service layer of the pattern. Each service component presented in this layer will be implemented as a REST API. This layer's main function is to provide an interface to handle HTTP GET/POST/PUT/DELETE REQUESTS, which means that this layer would form the endpoint for the service. All the requests from client side such as

   HTTP GET http://www.service.com/model
   HTTP POST http://www.service.com/model
   HTTP PUT http://www.service.com/model
   HTTP DELETE http://www.service.com/model

   will be handled by this layer. Based on architectural decisions, this layer can also contain server-side business logic along with service logic to handle HTTP REQUESTs. The service discovery and metadata provision mechanisms are also implemented in this layer. It is advised to maintain a dedicated real or virtual server as the hardware for this layer to operate smoothly.
5. **Service Consumer API**: The service consumer API forms the next layer of the architecture. The API is a general-purpose software component which is loosely coupled with the REST API. The consumer API can reside in an independent hardware, or on the client side. Unlike with the choice in run time objects layer, this layer does not contain the client-side business logic (such as software components for user experience or visualization support). The API here is solely designated to facilitate communication between the client and the REST API. This API acquires information from the REST API and this information is then transferred to the client. The API can also be used to transfer information from the client to the service side. Communication between the service consumer API and the REST API needs to be efficient and most of the bandwidth requirement of the overall architecture would be for this API. If an efficient communication mechanism between these two APIs cannot be established, it can generate a major bottleneck for the overall architecture.
6. **The Client**: The client is the software or a software component that is loosely coupled with the RESTful Web service. In other words, the client would exist and function perfectly without the existence of the Web service; similarly the Web service presented in the patterns would exist and function perfectly without the requirement of the existence of the client. The client can be a simple visualization interface working on a PC, a CAD or analysis software, a Web-based 3D visualization tool, a mobile device/tablet interface, an augmented

or virtual reality device user interface or any other user interface. The client provides the means to the user for interaction with the presented visualization of the model (which can either be in the form of a 3D model or a simple tabular data). The visualization of the information that is acquired from the Web service and the acquisition of the user input are the key functions of the client. Apart from its main functionality and based on user requirements, the client can provide voice or video communication with other clients. Following the generalized design pattern, the specialized patterns developed will be explained in two parts, first a problem definition will be provided, and then the structure of the service pattern and its role in solving the (early defined) problem will be presented.

6.4 REST Query Filter Pattern

The Problem In BIM-based collaborative environments there usually exist multiple stakeholders and multiple synchronous activities. Furthermore, a single user might require access to a set of selected elements either from the BIM (i.e. the model), an extended information model (i.e. extended BIM) or from a BIM view (i.e. model view). Multi-user environments with synchronously working devices can generate a lot of network traffic and require more bandwidth; furthermore, if the models are residing in a cloud database server, transferring the overall model (which might be large to several hundred MBs) for every transaction or user request will also be economically unfeasible.

The Solution As a solution to the above-mentioned problem, the REST query filter pattern describes a software architecture, where information from a set of BIMs, extended BIMs and BIM views can be transferred efficiently to the requesting parties over the Web using discrete and uncoupled query filter services for each data resource. The architecture consists of six layers.

1. **Data Layer**: This layer implements generalized design pattern data layer, which consists of an extended BIM, a BIM or a BIM view.
2. **Database API Layer**: The layer implements generalized design pattern database API layer.
3. **Run Time Objects Layer**: The layer implements generalized design pattern run time objects layer.
4. **Query Service Filter Layer**: This layer consists of discrete service components that provide RESTful interfaces for interacting with different data models of BIMs, extended BIMs and BIM views (Fig. 6.2). This service will focus on providing a filter to present similar building components (i.e. a set of columns, beams, fire alarms) or components of a group of building elements (first-floor elements, façade elements) or elements with time history (i.e. updated after last

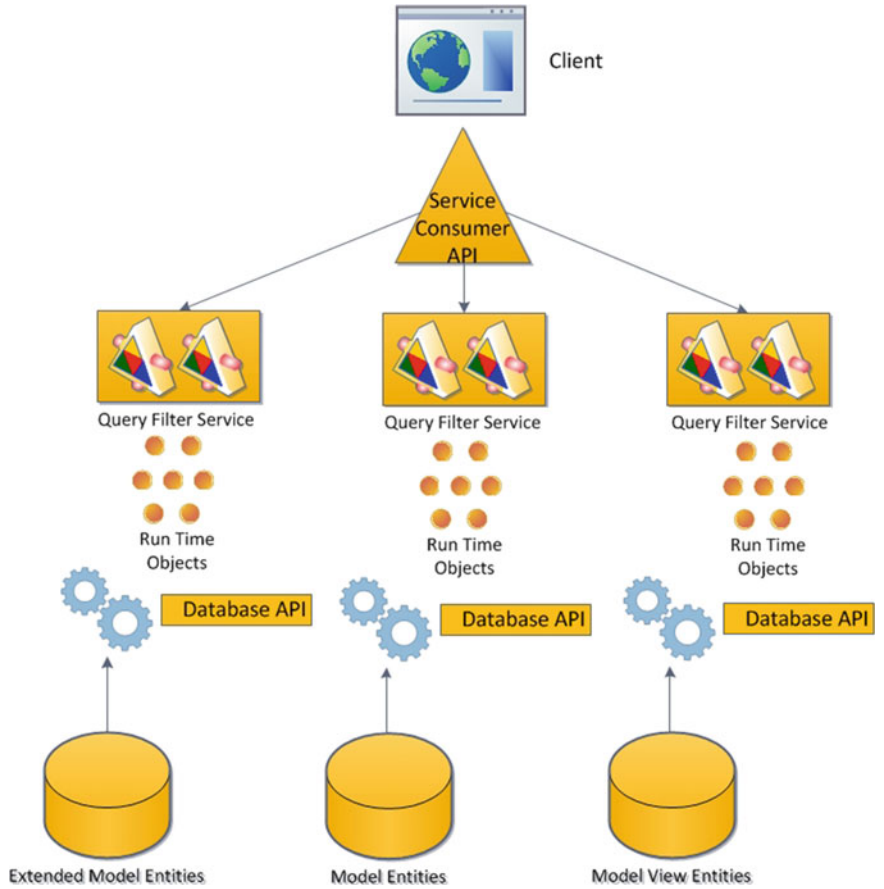

Fig. 6.2 REST query filter

year or updated in the last 100 days). An interaction with these API can include REQUESTS such as

HTTP GET http://www.service.com/model/filter/beams
HTTP GET http://www.service.com/modelview/filter/firstfloor
HTTP GET http://www.service.com/extendedmodel/filter/façade
HTTP GET http://www.service.com/model/filter/beams/updated/days/100
HTTP GET http://www.service.com/model/filter/walls/updated/years/1

5. **Service Consumer API**: This layer implements generalized design pattern service consumer API layer.
6. **The Client**: This layer implements the client description in generalized design pattern.

6.5 REST Façade Pattern

The Problem Today, design process in the construction industry requires tools for synchronous collaboration between the stakeholders. For example an architect, an engineer and a customer would have the need to work on the design documents (which are generated from the BIM) synchronously. In fact these parties are mostly located in different places, and it is difficult to make efficient synchronous use of design tools because of their location constraints. Thus there appears a need for collaborative use of information systems over the Web. In fact, as different stakeholders focus on different aspects of the process and need to interact with different parts of the data store, problems occur in reaching information from multiple model (data) sources such as the BIM itself, the extended model or the model view.

The Solution In order to propose a solution to the problem, REST façade pattern provides a single interface to multiple components of the data layer, which will act as a RESTful gateway to reach information in extended BIM, a BIM, a BIM view stored in databases or the model information residing in BIM files. The RESTful architecture provides loose coupling between the client and the provided façade. The architecture consists of six layers.

1. **Data Layer**: This layer implements generalized design pattern data layer, which consists of an extended BIM, a BIM, a BIM view or a BIM file which can be queried through the RESTful façade.
2. **Database/File API Layer**: The layer implements generalized design pattern database/file API layer.
3. **Run Time Objects Layer**: The layer implements generalized design pattern run time objects layer.
4. **REST Façade Layer**: The layer consists of a single service interface. The role of the interface is to provide a service to enable interaction with the multiple data sources through the Web (Fig. 6.3). The service will focus on queries targeted to multiple data sources such as acquiring the garden furniture information from the extended model while acquiring the outer installations of the building from the model (BIM) itself. Another query can be for exploring utilities inside the building together with utility elements in the garden (i.e. which are represented in the extended model). An interaction with the REST API of this service can include REQUESTS such as

   HTTP GET http://www.service.com/façade/outside_elements
   HTTP GET http://www.service.com/façade/all_utilities

5. **Service Consumer API**: This layer implements generalized design pattern service consumer API layer.
6. **The Client**: This layer implements the client description in generalized design pattern.

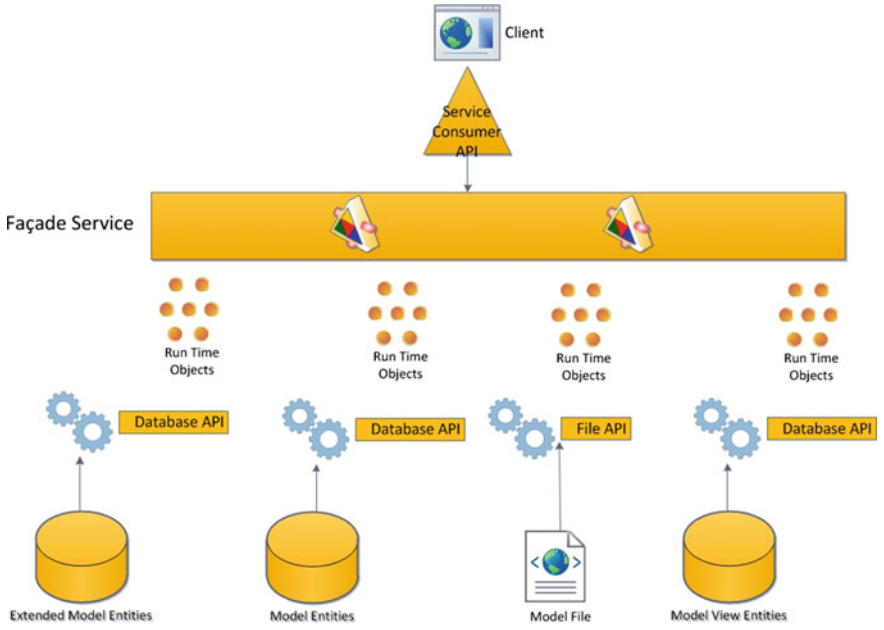

Fig. 6.3 REST façade

6.6 RESTful Real-Time View Generator Pattern

The Problem In multi-stakeholder processes of the building lifecycle, especially during the design and construction process, there is an ongoing need to reach the different elements of the model (related to the role of the stakeholder) on demand, by multiple stakeholders located in different places. The information requests sometimes are not fulfilled by early defined views as they do not cover the aspects for the required information, or transferring an overall model view(s) to the multiple stakeholders would require high bandwidth.

The Solution The RESTful real-time view generator pattern presents a solution to the problem by enabling the generation of on-demand model views that are tailor-made for the user requests. The pattern includes a service that will respond to requests of the users to generate the model views. The RESTful architecture provides loose coupling between the client and the provided service. The architecture consists of six layers (Fig. 6.4).

1. **Data Layer**: This layer implements generalized design pattern data layer, which consists of a BIM or a BIM file which can be queried through the RESTful real-time view generator.
2. **Database/File API Layer**: The layer implements generalized design pattern database/file API layer.

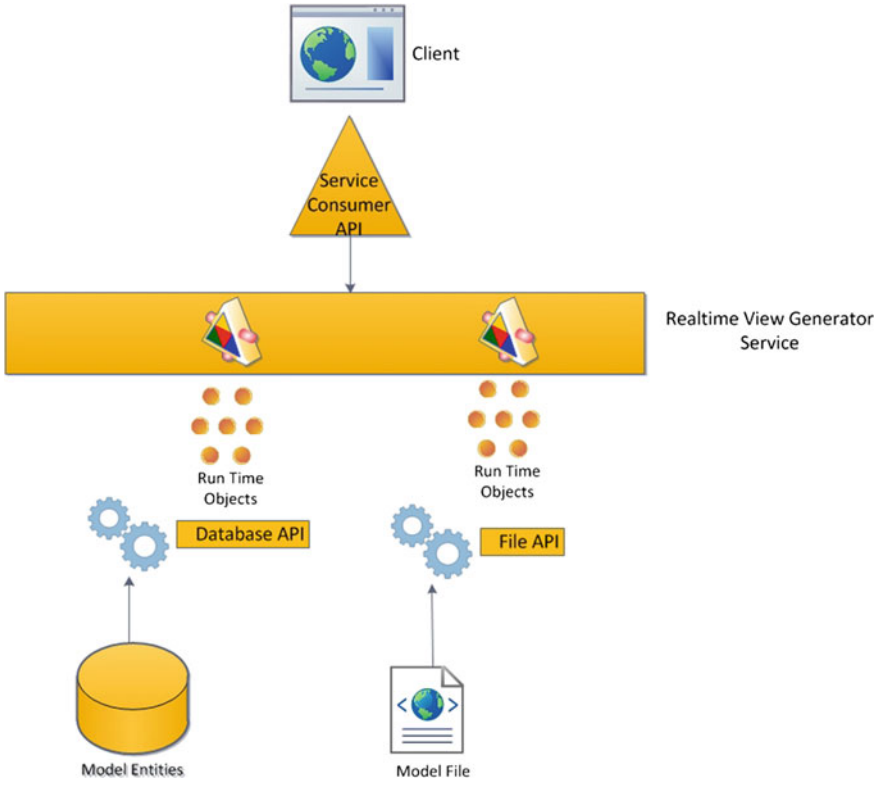

Fig. 6.4 RESTful real-time view generator

3. **Run Time Objects Layer**: The layer implements generalized design pattern run time objects layer.
4. **RESTful Real-Time View Generator Layer**: The layer consists of a single service interface. The role of the interface is to provide a service that will generate transient model views based on the requirements of the users. For example, a mechanical engineer would like to visualize the utility elements or HVAC elements of the second floor, or a civil engineer would like to check the details of columns on the first floor, or the architect would like to check the details of window elements. An interaction with the REST API of this service can include REQUESTS such as

   HTTP GET http://www.service.com/runtimeview/utilities/secondfloor
   HTTP GET http://www.service.com/runtimeview/columns/firstfloor
   HTTP GET http://www.service.com/runtimeview/windows
   HTTP PUT http://www.service.com/runtimeview/columns/firstfloor

   The service would also be able to handle the HTTP PUT/POST/DELETE REQUESTS; for instance, the last example PUT REQUEST can be used to

update the BIM with the as-built information provided from the construction site. In this case, the service (i.e. this layer) would interact with the run time objects layer to update the BIM with the latest changes that occur at the construction site.
5. **Service Consumer API**: This layer implements generalized design pattern service consumer API layer.
6. **The Client**: This layer implements the client description in generalized design pattern.

6.7 RESTful Memento Pattern

The Problem In the design and construction process the BIMs are updated frequently and the stakeholders have access to the latest version of the BIM. As this has been the key user requirement of the BIM-based construction management processes, most efforts focus on providing the most up-to-date version of the model. In fact, in many situations, specifically in the design process and less commonly in the construction process, there appears a need for examining the previous version of the model to compare the changes, and to notify what has changed and so on.

The Solution The RESTful memento pattern is focused on persisting the multiple state(s) of the BIM in the data layer and restoring an old version of the model when required by the user. In other words, the pattern focuses on enabling the backing up of the previous versions of the BIM in a persistence environment and restoring them. The memento service provided in this pattern would (i) generate a back-up copy of the model with the timestamp as a response to a user REQUEST (i.e. on demand) and would store this data in a model server database and (ii) would restore a model from the generated copies based on user REQUEST. The RESTful architecture provides loose coupling between the client and the provided façade. The architecture consists of six layers (Fig. 6.5).

1. **Data Layer**: This layer implements generalized design pattern data layer, which consists of a BIM that resides in a model server database.
2. **Database API Layer**: The layer implements generalized design pattern database API layer.
3. **Run Time Objects Layer**: The layer implements generalized design pattern run time objects layer.
4. **RESTful Memento Layer**: The layer consists of a single service interface. The role of the interface is to provide a service to generate a copy of the overall BIM based on a service call, and store this query in a model server database (i.e. where the current model resides). A BIM data warehouse can be built upon the stored versions of the model in a later stage. An example call to the service would involve a real-time back-up request or a batch request, or a scheduled request which can be accomplished when system resources at the data layer

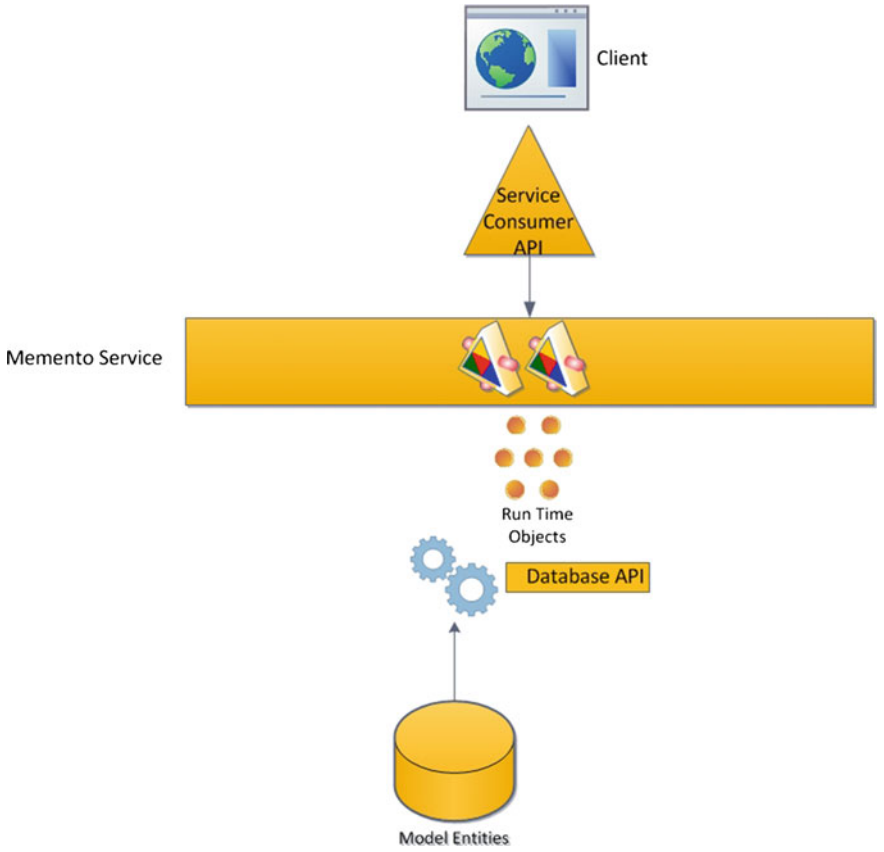

Fig. 6.5 RESTful memento

become free or at the scheduled time intervals. The service would also have the capability to restore the model from one of the previous versions. An interaction with the REST API of this service can include REQUESTS such as

HTTP GET http://www.service.com/memento/backupnow
HTTP GET http://www.service.com/memento/backup/weekly/saturday/22/30
HTTP GET http://www.service.com/memento/backup/monthly
HTTP GET http://www.service.com/memento/restore/version/date/10/20/2015
HTTP GET http://www.service.com/memento/restore/version/last
HTTP GET http://www.service.com/memento/restore/version/lastweek

5. **Service Consumer API**: This layer implements generalized design pattern service consumer API layer.
6. **The Client**: This layer implements the client description in generalized design pattern.

6.8 RESTful Model Multi-view Controller Pattern

The Problem The process in the BIM-based construction management requires environments for supporting synchronous communication and collaboration between the stakeholders. For example, an architect, an engineer and a customer would like to observe updates to the project schedule (on finished tasks which are) maintained by the BIM in real-time (while a site engineer is controlling the processes in a construction site). This process would require real-time updates to all stakeholders' devices/user interfaces regarding the real-time representation of the 4D visualization of the BIM and the project schedule.

The Solution The traditional model-view-controller pattern focuses on decoupling the user interface (view) from the business logic objects (model). A controller layer is located between the user interface and the business logic to update the states of the business logic objects based on the user interaction. In the pattern the views are observers of the model; once the model changes the views get notified. This traditional approach enables automatic update of the views once the model changes. In multi-view architectures the pattern is extremely useful as all changes in the model object's state are automatically reflected in the views. This RESTful model multi-view controller pattern provides an architecture for adopting the MVC approach for facilitating BIM-based collaboration systems, in order to enable real-time representation of 4D information in multiple clients (and user interfaces). Although the pattern has similarities with the traditional MVC there are also some differences. The main difference is at the controller component of this pattern which updates the views, in contrast to the traditional MVC where the views are updated by the model. Thus the business logic to update the views is shared between the model and the controller component. In the presented pattern RESTful MMVC, the number of views is more than one, the controller is implemented as a RESTful service, and the model consists of two layers (one being the run time objects and the other being the persisted information model that resides in the database). The controller role is overloaded in this pattern. The architecture consists of six layers.

1. **Data Layer**: This layer implements generalized design pattern data layer, which consists of a BIM that can be queried through the run time objects.
2. **Database API Layer**: The layer implements generalized design pattern database API layer.
3. **Run Time Objects Layer**: The layer implements generalized design pattern run time objects layer; in addition, components in this layer make calls to the controller service layer to update the view.
4. **Controller Service Layer**: This layer consists of a single service interface. The role of the interface is to provide a controller layer. At the start of the sequence, each client (view) subscribes to the controller service by providing its GUID or IP address in order to get up-to-date information about the model (i.e. both transient objects and persistent model). Once subscribed, each view is provided with the permission to manipulate the model. Upon user interaction with the

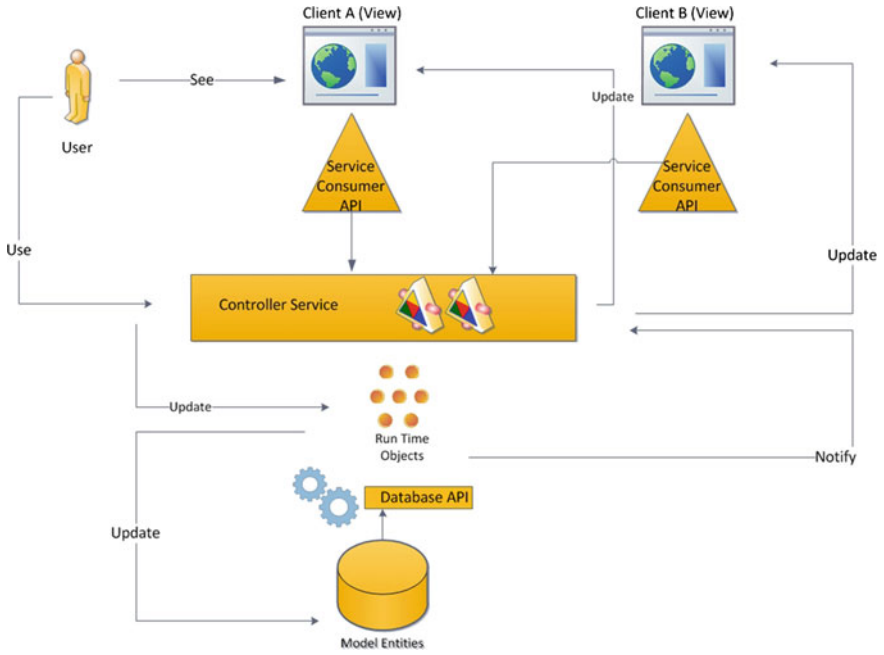

Fig. 6.6 RESTful MMVC

view, the controller service is notified by the service consumer API, which would also provide information regarding what has changed in the view. The controller service once being notified by the call of the service consumer API, notifies the run time objects which then manipulates the transient model objects and the persistent model in the database. Following the persistence of the change (or update) in the data layer, the run time objects interact with the controller service REST API to update the views by sending the changes in the model, for example, as a .json message. The REST API would then interact with the service consumer API to update the views. In this pattern the container of the run time objects would reside in a different platform/or even hardware from the service container (Fig. 6.6). An interaction with the REST API of this service can include REQUESTS such as

Client → Controller HTTP GET http://www.srv.com/controller/subscribe/id/2/ip/22.11.11.22:7777
Service Consumer API → Controller HTTP PUT http://www.srv.com/controller/update/model {data: json message}
Run time Objects → Controller HTTP PUT http://www.srv.com/controller/update/view/2 {data: json message}
Run time Objects → Controller HTTP PUT http://www.srv.com/controller/update/allviews {data: json message}

5. **Service Consumer API**: This layer implements generalized design pattern service consumer API layer.
6. **The Client**: This layer implements the client description in generalized design pattern. In fact there are multiple clients in this layer.

6.9 RESTful Call-Back Responder Pattern

The Problem A standard BIM-based collaboration environment consists of multiple users that are connected concurrently to a model server, either in a tightly coupled manner or in a loosely coupled one (such as over a REST API). In addition to these constraints a REST API in the architecture should need to respond to multiple queries in a sequence without causing latency for the client. In some situations these queries would include time-consuming calculations on the server side (such as structural analysis) which would generate responses that are very late for the client and this would cause inefficiencies in the architecture.

The Solution The term "blocking code" refers to a problem in computer programming where a piece of code would need to wait for a response from an operation in order to move on to the next step. If the operation—that is waited for—takes very long to complete, the overall response time increases. This situation is termed as the blocking code. The blocking code can appear anywhere in a service-oriented architecture (SOA), but if it happens on the server side, all clients will be affected by the poor time performance of the service. The call-back functions are the mechanism designed to tackle the problem of blocking code, and are implemented in some server-side programming frameworks such as JQuery, Node.JS/Express.JS. Operations that take a long time to complete in BIM-based collaborative environments can be facilitated by the use of REST APIs developed to support call-back functions. Frameworks such as Node.Js are ideal for the development of these services. The architecture consists of six layers.

1. **Data Layer**: This layer implements generalized design pattern data layer, which consists of a BIM or a BIM file which can be queried through the RESTful real-time view generator.
2. **Database API Layer**: The layer implements generalized design pattern database API layer.
3. **Run Time Objects Layer**: The layer implements generalized design pattern run time objects layer.
4. **Call-Back Responder Layer**: The layer consists of a single service interface developed with a non-blocking coding framework. The role of the interface is to provide a service that is developed using development environments that support the use of call-back functions. For instance, as illustrated in Fig. 6.7 there can be a REQUEST A to the service which takes 7 s to complete; during this

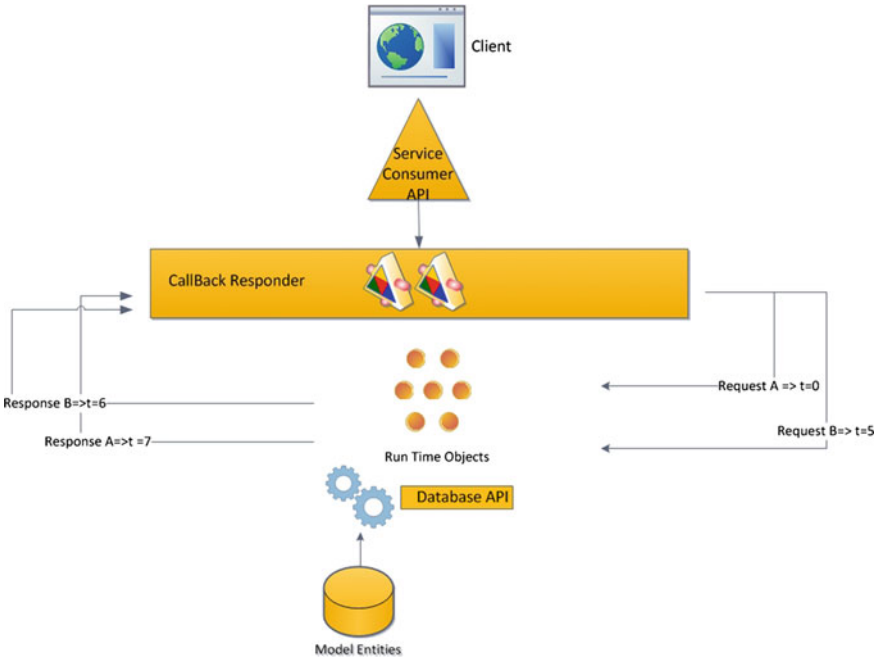

Fig. 6.7 RESTful call-back responder

process there can be another request, i.e. REQUEST B to the service 5 s after the first request. In such a situation as the call-back responder layer does not have to wait for REQUEST A to be completed, it can send REQUEST B to the run time objects layer to be processed. This second process can take 1 s to be completed and response to REQUEST B can be provided in the sixth second, while response to REQUEST A is provided at the seventh second, i.e. after the completion of the second request. REQUEST B in this situation is responded to without any latency. An interaction with the REST API of this service can include REQUESTS such as

HTTP GET http://www.service.com/callbackresponder/makeanalysis
HTTP GET http://www.service.com/callbackresponder/beams/secondfloor

5. **Service Consumer API**: This layer implements generalized design pattern service consumer API layer.
6. **The Client**: This layer implements the client description in generalized design pattern.

6.10 RESTful Authenticator Pattern

The Problem In multi-user environments for collaboration, such as systems used for BIM-based management of the building design and construction, there will be many stakeholders that require access to the BIM Web service. In fact, access by unauthorized parties would create chaotic situations as the models can be updated with unwanted information, or key information regarding construction process can be stolen or deleted.

The Solution This pattern presents an authentication architecture that is loosely coupled with the service. The architecture for the RESTful service that would interact with the model (i.e. BIM) is the architecture presented in the generalized design pattern, in fact this pattern has additional layers. An additional data layer and an additional service layer are present to serve for authentication purposes. The architecture in principle focuses on disabling the service discovery of the RESTful service that would interact with the model. In fact, the URI of the RESTful service will be provided at a later stage by the authenticator service as a result of successful authentication of the client side (Fig. 6.8). Further information on SOA authentication patterns can be found in Erl (2009). The architecture consists of six layers.

1. **Data Layer**: This layer implements generalized design pattern data layer that consists of a BIM or a BIM file which can be queried through the RESTful

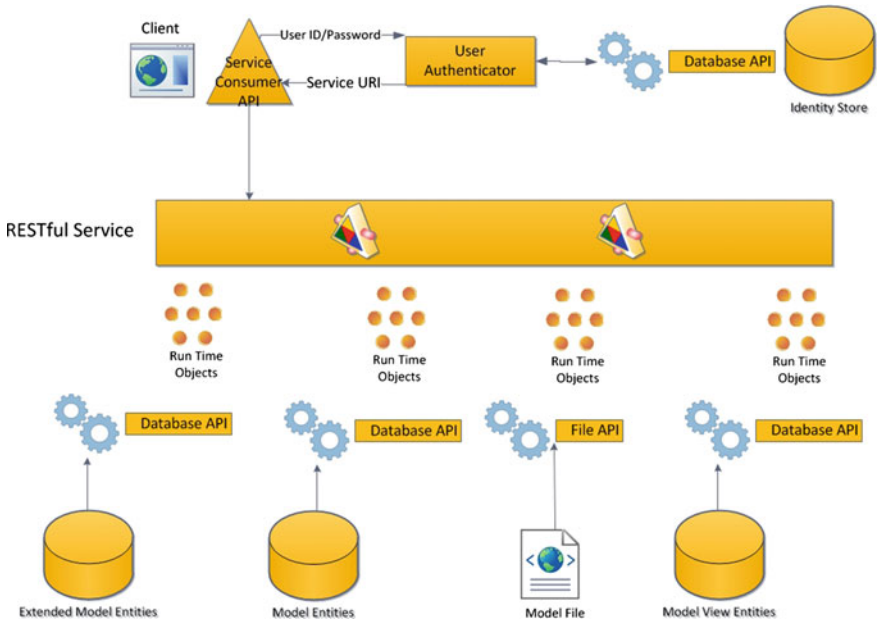

Fig. 6.8 RESTful authenticator

real-time view generator. This layer also implements another data layer where a relational or object relational database/cloud database (i.e. the identity store) holds the user information which will serve for client authentication.

2. **Database/File API Layer**: The layer implements generalized design pattern database API layer. The layer consists of another database API to enable interaction between user and authenticator service.
3. **Run Time Objects Layer**: The layer implements generalized design pattern run time objects layer. The layer does not exist on the authenticator side.
4. **User Authenticator Layer**: The layer consists of two service interfaces (i.e. user authenticator, and RESTful service). The user authenticator service will operate as an authentication gateway. A client will call the service with an authentication request by providing a username or a password. The service will then query the identity store and if the information matches the records in the identity store, it will respond to the URL of the RESTful service that the client requested. The user authenticator service can be developed in such a way that it will provide a set of URIs as a result of successful authentication. An interaction with the REST API of this service will include a REQUEST such as

   HTTP POST http://www.service.com/authenticator {data: json message}

5. **Service Consumer API**: This is a generalized API but also implements a function for passing the URI of the RESTful service (sent by the user authenticator) to the client.
6. **The Client**: Once the client finalizes the authentication and acquires the URI of the REST API of the target service it implements the client description in generalized design pattern.

6.11 RESTful Data Management Pattern

The Problem Interactions with building information models do not only involve data acquisition or data update functions at the fine granular object level. Sometimes the data layer might be complex and fuzzy in distributed systems. For instance, BIMs in use might not reside in a single database server, and the project may not contain a single standard BIM such as an IFC, but multiple BIMs. There might be different models defined with different schemas (i.e. Green Building XML, CIS2, IFC …) which reside in different platforms. In these situations, management and housekeeping at the back-end (i.e. the data layer) become vitally important. If the back-end is not managed successfully the problems will lead to chaos where information exchange and sharing would become a nightmare with so many models defined with different schemas, conflicting views, files that are not persisted in model servers and different versions of different models independently floating around in the data layer.

The Solution The RESTful data management pattern introduces a set of Web services to facilitate data management tasks in a distributed system. The Web services introduced in the pattern can be thought of as a Swiss Army knife for managing the information transformation tasks and facilitating data persistence in model servers. Two of the Web services concentrate on transforming information between two information models, while the other concentrates on persisting the BIM file contents (which has arrived into the data layer as a result of data exchange) in model server databases. The main difference in the architecture presented here from other patterns presented in this section is that the client layer includes data components (i.e. which can be regarded as the target data layer). The architecture consists of eight layers.

1. **(Source) Data Layer**: This layer is shown at the bottom of the diagram and implements generalized design pattern data layer, which consists of a BIM or a BIM file which can be queried through the Web services defined in this pattern.
2. **(Source) Database/File API Layer**: The layer is shown at the bottom of the diagram and implements generalized design pattern database/file API layer
3. **(Source) Run Time Objects Layer**: The layer is shown at the bottom of the diagram and implements generalized design pattern run time objects layer (Fig. 6.9)
4. **Model Management Service Layer**: The layer consists of three different service interfaces. The first one is the model transformer service which can be used

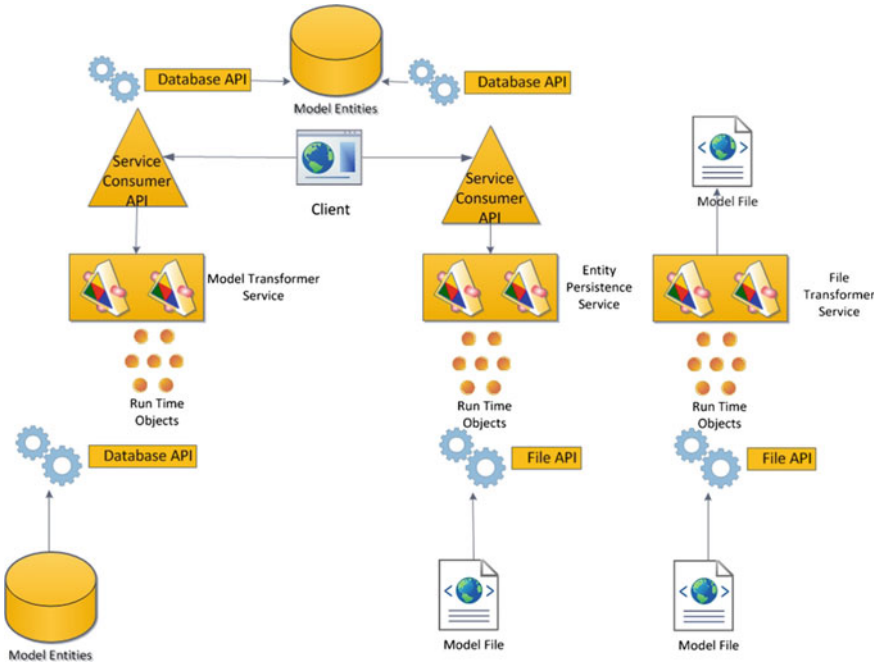

Fig. 6.9 RESTful data management

to transform a source information model with a different schema to the target information model (which is the main data source of another business process). The second service is the entity persistence service which can be used to transfer model entities residing in a file to the model server database in the target data layer. The third service is the file transformer service, which can be used to transform an exchange file of a BIM formatted with source schema to exchange file of a BIM formatted according to the rules of a target schema. For example, a transformation from IFC BIM to CIS/2 BIM can be accomplished using the service layer defined in this pattern. A request from the client to the service starts the transformation process, and the transformed model entities would then be persisted either (i) in target data layer databases using the service consumer API and the database API of the target data layer or (ii) in a target BIM exchange file. An interaction with the REST API of this service can include REQUESTS such as

HTTP GET http://www.service.com/modeltransformer/sourcemodel_id/001/targetschema/ifc2x4
HTTP GET http://www.service.com/entitiypersistence/sourcemodel_id/002
HTTP GET http://www.service.com/filetransformer/sourcemodel_id/001/targetschema/ifc2x4

5. **Service Consumer API**: This layer implements generalized design pattern service consumer API layer.
6. **The Client**: Client is responsible for providing the user interface to manage source/target files and databases, and managing the transformation operations by issuing call to the REST API of the model management service layer.
7. **(Target) Database API**: The layer implements generalized design pattern database API layer for the target database.
8. **(Target) Data Layer**: This layer implements generalized design pattern data layer where a BIM is persisted in an object database and in an exchange file.

6.12 RESTful View Synchronizer Pattern

The Problem In multi-user multi model distributed system architectures, which can be beneficial for facilitating the communication, interoperability and information, the clients (i.e. the user interfaces) would have a need to combine information acquired from multiple Web services. In fact as these services are also endpoints which are loosely coupled with the client, the data transfer speed from each different service can vary depending on the location of the servers' bandwidth in the communication process. This variance will result in latency for visualization of information when some services wait for other services to respond and send information. This would create a blocking situation for the visualization environment, which prevent timely visualization of information sent by the Web services.

The Solution The pattern view synchronizer explained here is adopted from the UI mediator pattern (Erl 2009). The architecture of this pattern utilizes a view synchronizer service which will act as a component that will synchronize information coming from multiple sources. The synchronizer service presented in this pattern has two functions. It synchronizes the information sent from multiple endpoints (REST APIs) and also acts as a façade layer and provides a single gateway to multiple REST endpoints. The architecture consists of seven layers (Fig. 6.10).

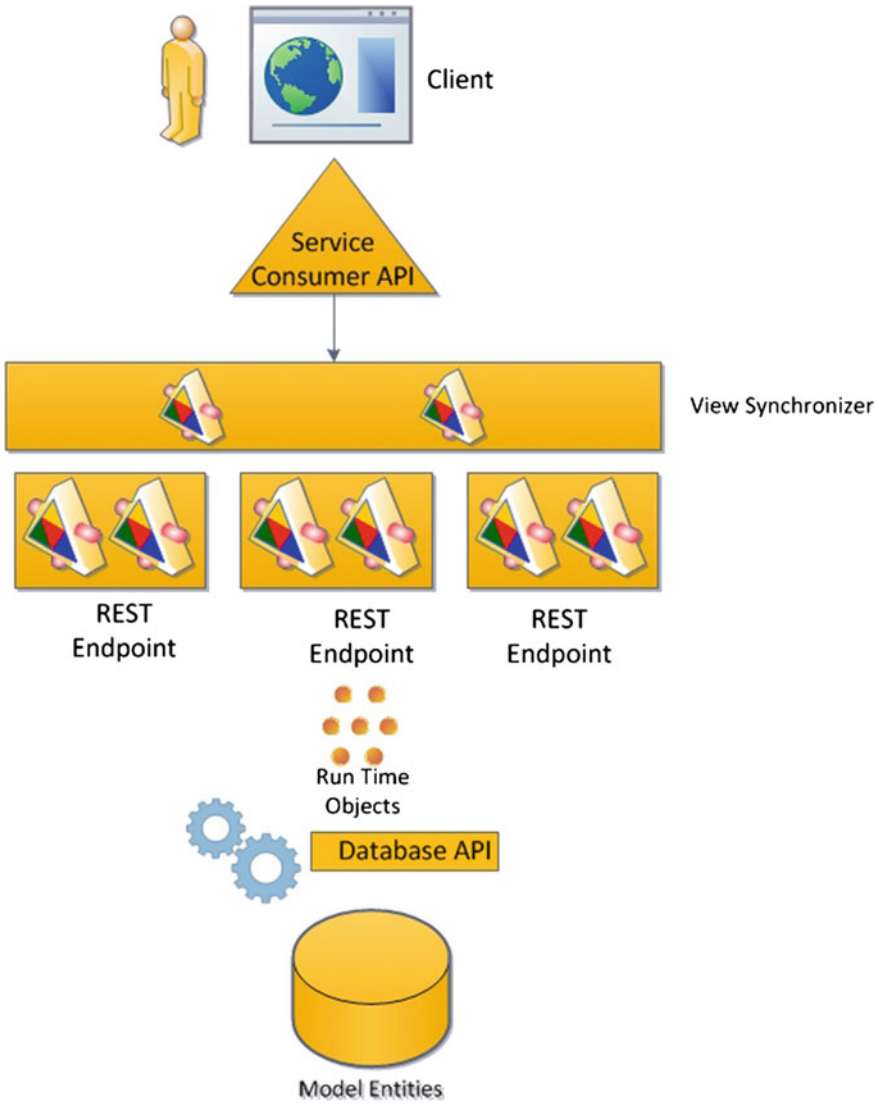

Fig. 6.10 RESTful view synchronizer

1. **Data Layer**: This layer implements generalized design pattern data layer, which consists of a BIM residing in a model server database.
2. **Database API Layer**: The layer implements generalized design pattern database API layer.
3. **Run Time Objects Layer**: The layer implements generalized design pattern run time objects layer.
4. **REST Endpoint Layer**: The layer consists of a multiple service interfaces which comply with the definitions in generalized design pattern.
5. **View Synchronizer Service Layer**: The layer consists of a single service interface. The role of the interface is to provide a service that provides the synchronized information derived from multiple REST Endpoints (i.e. REST APIs). As the business logic for the synchronization would be towards enabling automatic synchronization, the interaction with this layer would be similar to the REST façade pattern. An interaction with the REST API of this service can include REQUESTS such as

   HTTP GET http://www.service.com/synchronized/outside_elements
   HTTP GET http://www.service.com/synchronized/all_utilities

6. **Service Consumer API**: This layer implements generalized design pattern service consumer API layer.
7. **The Client**: This layer implements the client description in generalized design pattern.

6.13 RESTful Event Manager Pattern

The Problem As explained in Chap. 2, building information modelling in the near future would require real-time information provision regarding the states of the elements. This information will be represented in the BIM in real-time. For instance, a state of the door (such as being open or shut) or a state of the elevator (being busy or idle) would reside in BIM databases to support smart city and smart building applications. The next chapter will provide patterns for information provision from sensors along with BIM for supporting these applications. The information consumers in this case are not interested in the information unless a state change occurs. In other words, if the client is notified when a door's state has changed from open to closed, the client is not interested in updates on the state of the door every minute, but would like to get notified when the door's stage changes back to open again. In fact, the Web services are mainly capable of providing the information as a response to user request but not when a change occurs on the server-side model.

The Solution In a situation where a view (client) interacts with the model to change its state it would be feasible to implement the RESTful MMVC pattern; in fact, in this situation the role of the client is only visualization, not interaction with the service. Erl (2009) proposes the event-driven messaging pattern for messaging services for a similar situation. The idea of the event manager proposed by Erl (2009) can be extended to define an event manager service, which is working with publish/subscribe approach. Once the client subscribes for specific events, the event manager can send notifications to the client once the event occurs (such as the state change of door from open to close).

1. **Data Layer**: This layer implements generalized design pattern data layer, which consists of a BIM residing in a model server database
2. **Database API Layer**: The layer implements generalized design pattern database API layer.
3. **Run Time Objects Layer**: The layer implements generalized design pattern run time objects layer; in addition, components in this layer make calls to the controller service layer to update the client.
4. **Event Manager Layer**: The layer consists of a single service interface. At the start of the sequence each client (view) subscribes to the event manager service by providing its GUID or IP address in order to get state changes of the model. Once a state change is represented in the model (such as the start of air conditioning units in the building), the run time objects interact with the event manager to update the views by sending the changes in the model, for example, as a .json message. The REST API would then interact with the service consumer API to update the client. In this pattern the container of the run time objects would reside in a different platform/or even hardware than the service container (Fig. 6.11). An interaction with the REST API of this service can include REQUESTS such as

   Client → Controller HTTP GET http://www.srv.com/emanager/subscribe/id/2/ip/22.11.11.22:7777
   Run time Objects → Event manager → Service Consumer API
   HTTP PUT http://www.srv.com/emanager/update/client {data: json message}

5. **Service Consumer API**: This layer implements generalized design pattern service consumer API layer.
6. **The Client**: This layer implements the client description in generalized design pattern.

This chapter provided software architectures for provision of BIM information through Web service. The following chapter will elaborate on software architectures for provision of sensor information that is acquired from indoor sensors.

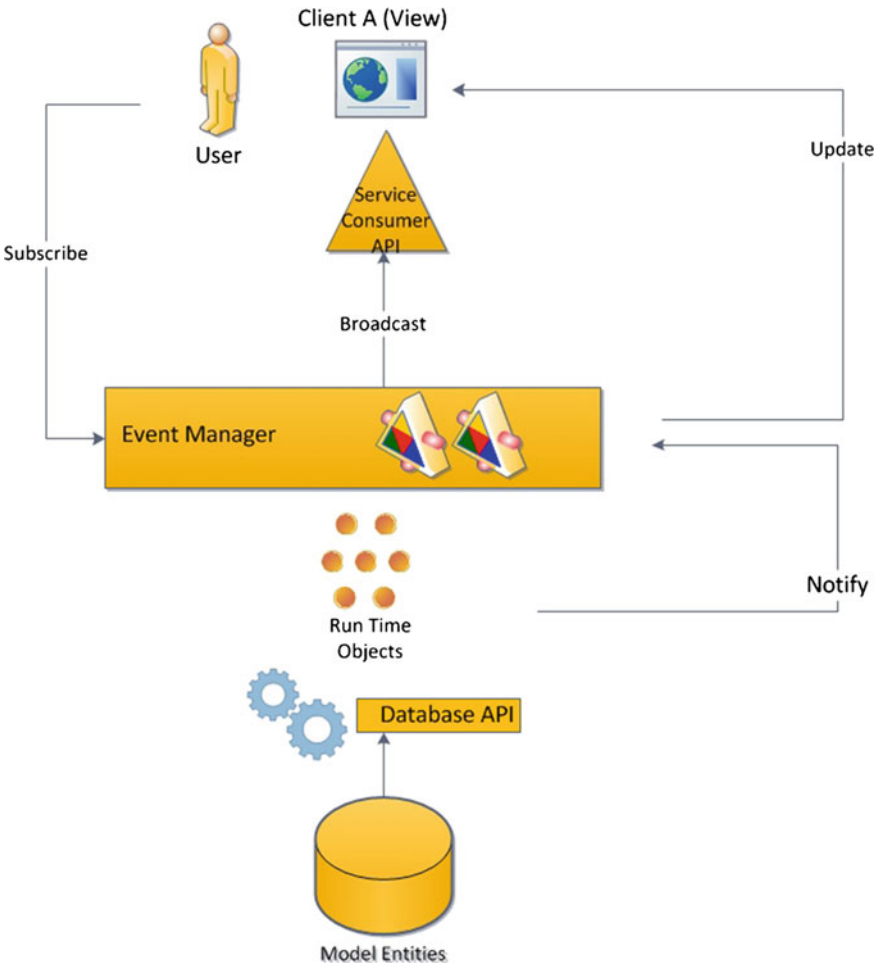

Fig. 6.11 RESTful event manager

References

Erl, T.: SOA Design Patterns. Prentice Hall, New Jersey (2009)

Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, Department of information and computer science, University of California, Irvine (2000)

He, H.: What is service-oriented architecture. Online at http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html. Accessed 21 July 2004 (2003)

Isikdag, U., Underwood, J.: Two BIM based web-service patterns: BIM SOAP façade and RESTful BIM, construction in the 21st century conference, Istanbul, May 2009 (2009)

Pautasso, C., Zimmermann, O., Leymann, F.: Restful web services vs. “big” “web services: making the right architectural decision” WWW ‘08: proceeding of the 17th international conference on World Wide Web, pp. 805–814 (2008)
Pulier, E., Taylor, H.: Understanding Enterprise SOA. Manning Publications, Greenwich (2006)
RESTful API Tutorial: http://www.restapitutorial.com/lessons /restfulresourcenaming.html (2015)
Techtarget: Definition of REST available at : http://searchsoa.techtarget.com/definition/REST (2015)