Model-Driven Generation of a REST API from a Legacy Web Application*

Roberto Rodríguez-Echeverría, Fernando Macías, Víctor M. Pavón, José M. Conejero, and Fernando Sánchez-Figueroa

University of Extremadura (Spain),
Quercus Software Engineering Group
{rre,fernandomacias,victorpavon,chemacm,fernando}@unex.es
http://quercusseg.unex.es

Abstract. Web 2.0 phenomenon, REST APIs and growing mobile service consumption, among other factors, are leading the development of web applications to a new paradigm, named cross-device web application. Those web sites let organizations of all sizes provide a pervasive and contextual access to their information and services, to customers, employees and partners via potentially any kind of device. Most organizations often possess legacy systems which should face an ongoing evolution process to enhance its accessibility and interoperability. Yesterday they had to evolve to provide the user with a Web layer, and now they should evolve again to adapt to the new ways of data and services consumption on the Web. In such scenario, a REST API plays a key role, defining the interaction layer between the legacy system and all its heterogeneous front ends. This work presents a model-driven approach to derive a REST API from a legacy web application within the frame defined by a modernization process. This approach departs from a conceptual model of the legacy application generated by reverse engineering techniques. In this work we detail the API generation process and provide a sample implementation instrumenting one of the studied web development frameworks to evaluate the suitability of the approach.

Keywords: Software Modernization, Software Reengineering, Rich Internet Applications, REST.

1 Introduction

Since the publication of the REST architectural style [4], the design and development of RESTful web services has become the *de facto* standard to define the interaction between Web 2.0 frontends and their backends. Actually, REST APIs may be currently conceived as one of the key factors in the success of a web (cloud) application, so a great effort is dedicated to its development and

* Work funded by Spanish Contract MIGRARIA - TIN2011-27340 at Ministerio de Ciencia e Innovación and Gobierno de Extremadura (GR-10129) and European Regional Development Fund (ERDF).

Q.Z. Sheng and J. Kjeldskov (Eds.): ICWE 2013 Workshops, LNCS 8295, pp. 133–147, 2013.

evolution. REST APIs define, on the one hand, a way of lean integration among a service provider and other applications (consumers), and on the other hand, a mean to get a cross-device web application, spreading the original scope of the web application.

Legacy systems have always considered web service technology as a proper mean to gain interoperability and to decrease their evolution related costs. Since RESTful services became mainstream fueled, first, by the Web 2.0 adoption and, then, by ongoing mobile service consumption raise, legacy applications have searched for alternatives to evolve their interaction layers (web or RPC services) to this new standard and become cross-device web applications.[1]describe how web service ecosystems have risen to become the predominant model in software solutions. Traditionally centralized domains, such as business solutions, electronic shops and electronic auctions are now publishing pieces of their functionality to create web service ecosystems.

The main goal of this work is to present an approach for the automatic generation of a REST API that provides an alternative interface to a legacy Web system. In this work we present, on the one hand, the architecture of a REST support layer defined as an extension of the Struts v1.3 Web development framework and, on the other hand, the model-driven process to adapt such layer to obtain a REST API conformed to the legacy subsystem to be modernized.

The rest of the paper is structured as follows. Section 2 briefly presents an overview of the MIGRARIA project and its MVC metamodel. In Section 3 an illustrative example is depicted. Section 4 introduces our approach to generate a REST API from a legacy web application. The related work is discussed in Section 5. Finally, main conclusions and future work are outlined in Section 6.

2 MIGRARIA Project

The process defined herein is framed within a whole modernization strategy: MIGRARIA project [9]. This project defines a systematic and semi-automatic process to modernize legacy non-model-based data-driven Web applications into RIAs. The process starts with the generation of a conceptual representation of the legacy system by using reverse engineering techniques. According to that project, this conceptual representation is based on the MIGRARIA MVC metamodel that allows the specification of legacy web applications developed by means of MVC-based web frameworks. Once the conceptual representation is obtained, the modernization engineer must decide the legacy system functionalities that should be part of the target system, i.e. the new RIA client. In other words, she must specify the set of components or subsystem to modernize. This decision is carried out in our approach by the selection of the different views included in the subsystem within the generated conceptual model of the legacy application. Note that the modernization process defined by MIGRARIA does not aim at sustituying the current system by a new one. Indeed, its main purpose is to complement the system by the generation of new ways of access and interaction such as a RIA client [8] and its corresponding server connection layer, a REST API in this work.

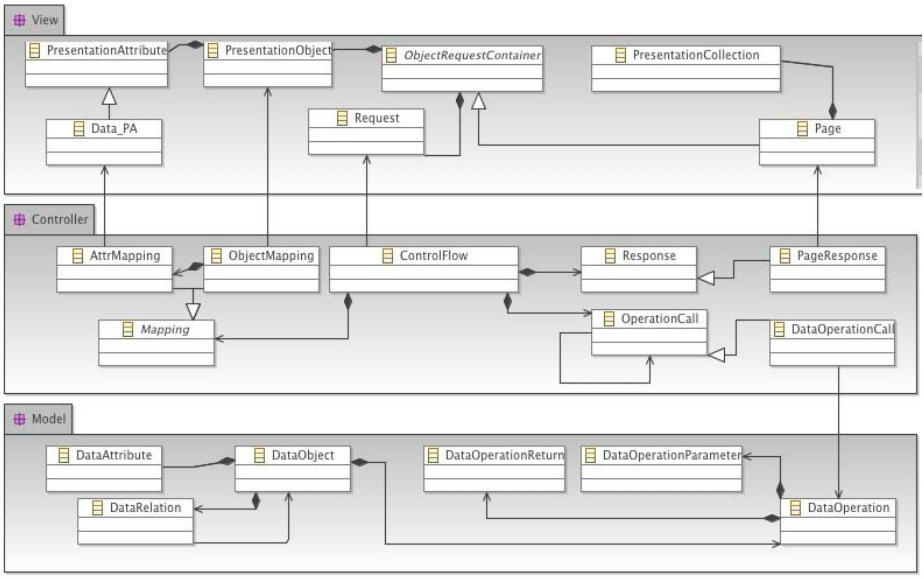

Fig. 1. MIGRARIA MVC metamodel overview

2.1 MIGRARIA MVC Metamodel

Within the MIGRARIA project, a specific language has been defined to generate a conceptual representation of a legacy MVC-based web application: the MIGRARIA MVC metamodel. This metamodel has been designed based on the Model-View-Controller pattern that has become the conceptual foundation of a great number of web development frameworks. In that sense, this metamodel specifies the main concepts of the development of a web application arranged in the three main components of the MVC pattern. Figure 1 presents an excerpt of the MIGRARIA MVC metamodel focusing on the main elements of each component. Such elements and their relations are used throughout this work to develop an illustrative example of the approach. The Model package provide elements to represent data objects, their attributes, their relationships and the operations defined over them. The View package provide elements to represent pages, as main containers, and presentation objects and requests, as main containtments. Presentation objects, basically, have a set of attributes, can indicate data input or data output and can be presented individually or inside a collection. Meanwhile, requests are characterized by their parameters and path and define connection points with controller elements by means of the request-handler association. The Controller package provides elements to represent request handlers (ControlFlow), their mappings defined between presentation and data objects, their response defining a relationship with the target page element, and the sequence of operation calls performed to execute the requested action or to fetch

the requested data. This metamodel is a comprehensive revision and extension of our work presented last year [8].

Although our intention is to follow the guidelines proposed by Architecture Driven Modernization (ADM) [12], we have declined to use Knowledge Discovery Metamodel (KDM) because of its complexity to specify MVC-derived semantics. We have also declined to use one of the ripe existing MDWE approaches [10] cause most of them are designed within forward engineering approaches. The abstract concepts they defined are difficult to match with the information resulting of a reverse engineering process. In our opinion, the semantic gap may be wide enough to motivate the definition of a new language to bridge the technology platform of a web application and MDWE approaches.

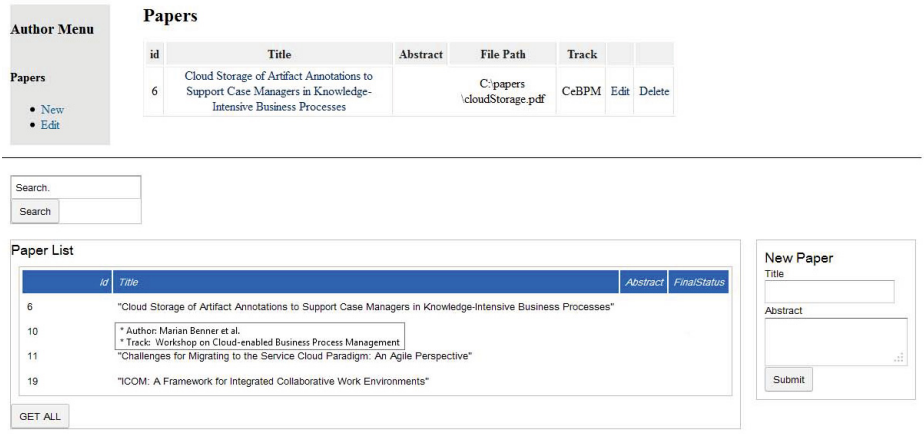

Fig. 2. Legacy Web Application and RIA snapshots

3 Illustrative Example

Figure 2 illustrates a modernization scenario extracted from the main case study of the MIGRARIA project, named the Conference Review System[1]. The data model proposed by [2] has been respected during system development. In this case, the paper submission subsystem has been selected to be modernized. In Figure 2, the top half presents a legacy view listing the papers submitted by a concrete author, and the bottom half shows a possible RIA client as the result of the modernization process. This new client is a composite formed by the legacy views: paperList (in the image), paperCreate, paperEdit and paperDetails. So the generated RIA client follows the single-page paradigm providing all the CRUD operations from a single view [7]. Basically this page is composed of an interactive list of papers, popping up additional information on mouse

[1] `http://www.eweb.unex.es/eweb/migraria`

over event, a search box, and a web form to submit a new paper or to edit an existing one. All the requests performed by that client are AJAX-based and REST-compliant, and used JSON as data exchange format. So the server side needs to provide a convenient connection layer for such client.

Having introduced that modernization scenario, in the next sections we will tackle the generation of a REST API to connect that new RIA client with the legacy application of our case study.

4 The Approach

The main goal of this work is to present an approach for the automatic generation of a REST API that provides an alternative access to a legacy Web system. This process is framed within the modernization framework defined by the MIGRARIA project [9].

Based on the selected subsystem to modernize, the process must generate, on the one hand, a RIA client and, on the other hand, a REST API that allows the communication between the new RIA client and the legacy Web application. In this paper, we focus on the approach defined to generate the REST API in the context of the modernization scenario defined. The REST layer must then fulfill the next requirements:

1. Adapting its behavior based on a configuration file.
2. Adding support for HTTP PUT and DELETE methods, if needed.
3. Adding parse and conversion support for common data formats (JSON, XML).
4. Delegating to a legacy controller the action of handling a REST request (REST-controller mapping).
   (a) Generating the context for the proper execution of the legacy controller (JSON-objects conversion).
   (b) Generating the response according to a particular format (previously decided).

In the specification of the REST API we may identify two different parts clearly separated: (1) a part of the system depending on the Web framework used to develop the legacy application; (2) a part specific for the legacy system being modernized. The former includes all the extensions and adaptations performed in the framework to add it the REST capabilities aforementioned. The latter mainly concerns to the specification of the behavior of the API for a particular Web application, namely: (i) the mappings among REST requests and legacy controllers; (ii) the input (request) and output (response) data conversion among the formats expected by, on the one hand, the RIA client and, on the other hand, the controller. Obviously, while the first part remains unchanged for different legacy Web applications, the second one depends on the concrete legacy system being modernized. The process followed to specify the Web application dependent part is illustrated by Figure 3. As it can be observed, the work presented herein takes the MVC model of the legacy application as input and generates its REST API. This process is defined by the next activity sequence:

1. Identification of the resources related to the subsystem to be modernized.
2. REST URI[2] generation for the identified resources.
3. Identification of the available operations over the resources based on the selected views, their requests and the controllers that manage these requests.
4. Generation of request mappings for the operations available for each resource and for a particular data format.
5. Delegation of the REST request handling to the controllers available in the legacy application and providing the correspondingly data conversions.

A detailed description of both parts of this REST API generation process is provided in the next sections.

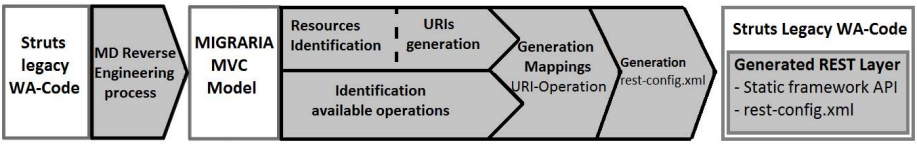

Fig. 3. Process outline

4.1 Framework-dependant REST API Specification

In order to generate a REST API from the legacy application, the framework used to develop the legacy system must be extended conveniently to enable the handling of REST requests. From a modernization point of view, we may rely on two different approaches to obtain this REST API:

1. To generate the REST API from its conceptual description by using model-to-code transformations.
2. To manually implement the REST API and, later, automatically generate its configuration based on the information extracted from the legacy application and the functionality to be included into the target RIA client.
   The main disadvantage of the first alternative concerns to the need of adapting (or redefining) the transformations once and again for each Web framework considered in the approach even, in many cases, for its different versions. Moreover, based on the structural properties of this REST API which clearly contains a static part, it does not seem to make sense to completely generate it from scratch by means of transformations. Thus, the second alternative seems more suitable although it implies implementing and maintaining different versions of the REST API according to the different frameworks and versions. In other words, this second approach requires different implementations for the framework-dependent part for different frameworks and versions whilst the variable part is automatically generated by means of model-to-text transformations. This variable part represents, in our case, the REST API configuration and, thus, it will be automatically generated by model transformations.

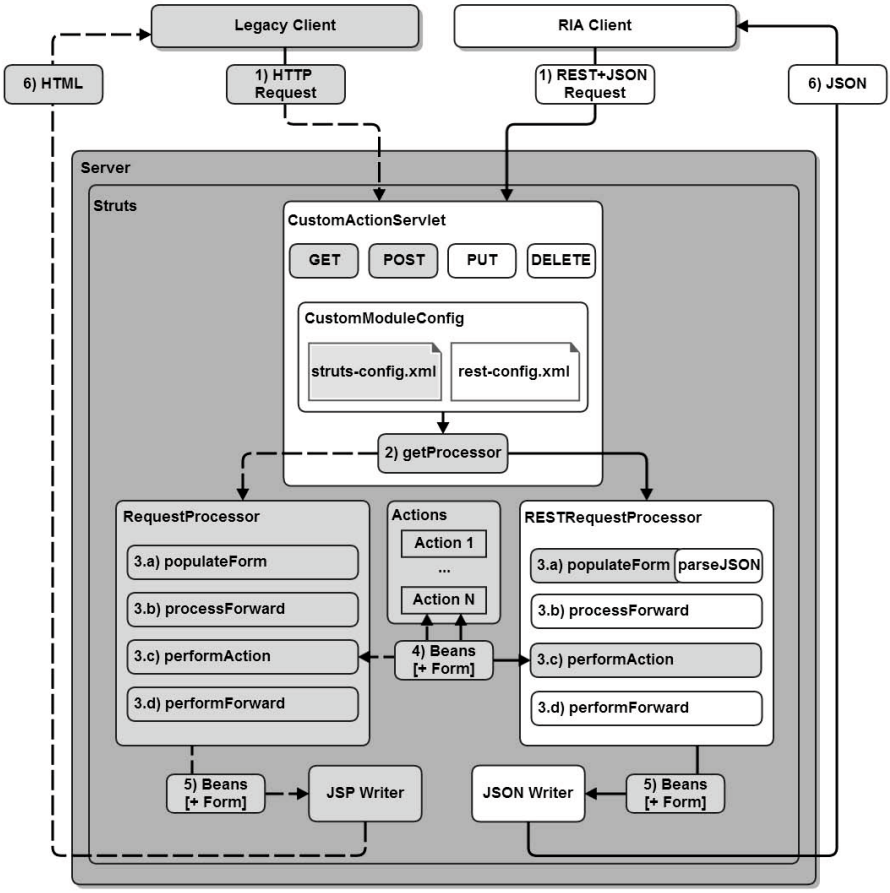

Fig. 4. REST layer for Struts v1.3.X

In order to illustrate how a particular Web framework is instrumented to provide REST API support according to our approach, an implementation for the Struts v1.3 Web framework has been developed. This implementation is graphically described in the process shown in Figure 4. This figure is organized according to the lifecycle of a request in the Struts framework extended with the REST API support. Note that each step in the lifecycle has been enumerated in the figure. The figure shows, on the one hand, the request-response lifecycle for a request generated by the legacy Web application (dashed line) and, on the other hand, the request-response cycle for a REST request generated by the target RIA client (solid line). This comparative illustration allows appreciating the modifications performed to the Web framework in order to incorporate the

[2] Uniform Resource Identifier.

REST API support. Notice that the white boxes indicate modified or generated elements. It may be easily observed that the extension has basically required the modification of the classes: *ActionServlet* (that implements the Front Controller [3] in Struts) and *RequestProcessor* (which handles the request lifecycle within the framework). on the one hand, the *CustomActionServlet* class has the next responsibilities: (1) adding support for PUT and DELETE HTTP requests; (2) loading the specific configuration of the REST API for a concrete Web application (rest-config.xml); and (3) deciding which *RequestProcessor* must handle the request. On the other hand, *RESTRequestProcesssor* is responsible of: (1) generating the needed context for the legacy controller invocation (step 3-a); (2) generating the response object according to the specified data format (step 3-b); (3) delegating to the legacy controller (*Action*), according to the REST-controller mapping specified in the configuration, the handling of the request (step 3-c); and (4) delegating the response generation in the corresponding component according to the specified data format (step 3-d). In addition to the commented extensions, the *JSONWriter* class has been added, as an alternative new response composer. In this case, it generates a JSON-valid response from the data generated by the invoked controller (*Action*). Obviously, the generation of responses in other data formats would require the implementation of different Writer classes.

4.2 Application-dependant REST API Specification

Next, we describe the main steps of our approach to extract the information to set up the framework-dependant REST API implementation. Such information is obtained from the MIGRARIA MVC model of the legacy Web application.

Table 1. REST resources and relations

Views	Potential Resources	Resources	I/O	Collection	Relations
paperCreate	paperForm	Paper	I	NO	1 Track : N Paper
	track	Track	O	YES	1 Paper : N Coauthor
paperEdit	paperForm	Paper	I	NO	1 Paper : N Review
	track	Track	O	YES	1 Author : N Paper
	coauthorForm	Coauthor	I	NO	1 Author : N Review
	coauthor	Coauthor	O	YES	N Author : M Track
paperDetails	paper	Paper	O	NO	
	coauthor	Coauthor	O	YES	
	review	Review	O	YES	
paperList	paper	Paper	O	YES	
	author	Author	O	NO	

[3] `http://struts.apache.org/development/1.x/apidocs/org/apache/struts/action/ActionServlet.html`

Identifying Resources. The specification of a REST API implies to redesign the interaction between client and server from a resource-centric point of view. In that sense, the adequate identification of the involved resources and their relations becomes a fundamental step in the REST API generation process. In order to carry out this activity, the modernization engineer must analyze and query the MIGRARIA MVC model to find out the resources related with those views that compose the subsystem to be modernized. Each view container (page) is composed by one or more presentation objects that define a concrete data view over the corresponding data objects defined at the model component, e.g. presenting some of their attributes. These presentation objects may represent input or output data and they may be either presented individually or organized into collections. In order to identify the model objects, an analysis of the controller component is required to identify the actual mappings defined among presentation and model objects within a concrete view.

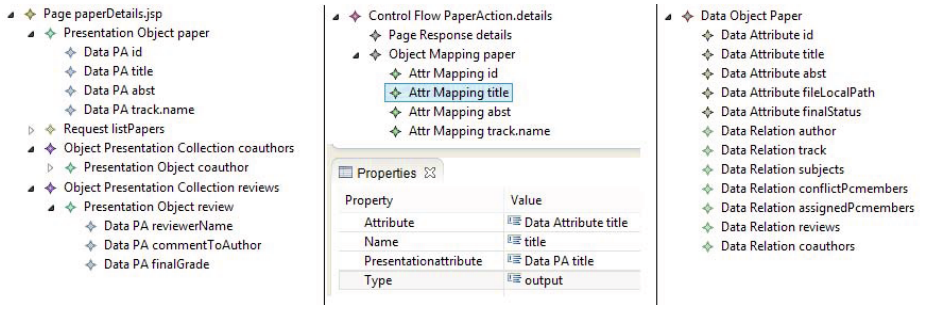

Fig. 5. Resource identification on a MIGRARIA-MVC Model

Figure 5 presents the presentation object *paper*, included in the view *paperDetails*, and its mapping with the data object *Paper*, defined in the controller *paperAction.details*. As the final result of the analysis of the MIGRARIA MVC model, the engineer must obtain: i) the data objects involved in the subsystem to be modernized; ii) their relationships; and, moreover, iii) the different ways of being presented by views and consumed by controllers. All this information is gathered in the potential resources column of the table 1 which shows the results for the paper submission subsystem in our running example. This table contains also the relations established among the collected resources.

Generating REST URIs. Taking as input the information regarding the resources and their relations obtained in the previous step, all the feasible REST URI combinations for each obtained resource are generated by means of the next process. The URIs column of table 2 shows a brief sample of the collection generated for our running example which illustrates the main construction blocks of our process.

The steps of the process are described as follows:

1. Generate base URI pair for each resource:
   (a) /resources
   (b) /resources/{id}
2. Generate base URI pair for each one-to-many relationship:
   (a) /resources_one/{id}/resources_many
   (b) /resources_one/{id}/resources_many/{id}
3. For each many-to-many relationship, locate the master resource and execute the previous step taking the master resource as the main one.
4. Complete the URIs adding to the query string any control parameter that appear in the original requests.

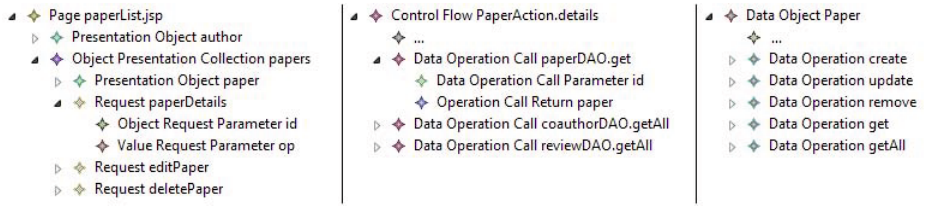

Fig. 6. Operation identification on a MIGRARIA-MVC Model

Identifying Available Operations. Considering that our modernization process takes as input a previously implemented legacy application, obviously the set of operations for each resource that the REST API must provide is limited by the operations available in the original system. Therefore, we need to identify the operations previously defined over these resources to be modernized in order to properly filter the set of possible REST requests to be provided. Again, this information is obtained by the modernization engineer by analyzing and querying the MIGRARIA MVC model. This analysis allows, in this case, obtaining all the requests related with the set of views that compose the subsystem to be modernized. Although most of such requests are included in these views, there may be other ones not included in the views but being responsible for generating a concrete view, e.g. a request generated by a menu link. Note that the MIGRARIA MVC model provides enough information to identify the controller handling a particular request and moreover the controller responsible for the generation of a particular view. The Controller component of a MIGRARIA MVC model permits the engineer to specify the operation call sequence executed by a controller to manage the data objects of the legacy application. Based on the analysis of this sequence, the operations being performed over a resource within a controller may be identified. A comprehensive analysis of the controllers that handle the set of requests related with the modernization provides, as result, the list of available operations for each resource.

To illustrate the operation identification process, we consider here a specific request contained in one of the views to be modernized of the running example, concretely the *paperDetails* request of the *paperList* view. Based on this request, the controller (ControlFlow) that handles it is identified by means of the existing relationships between the *View.Request* and *Controller.ControlFlow* elements. In this case, the *paperDetails* request is handled by the *paperAction.details ControlFlow* which contains the operation call sequence described in figure 6. In this case, for example, the operation *get* is invoked over the data object *Paper*, passing as parameter the *id* of the paper (request parameter) to fetch and receiving a *Paper* object as result. As shown, in this model excerpt, all the operation calls are related to concrete data operations defined in a concrete data object. In a MIGRARIA MVC model, all the data operations are classified according to its defined mission among the following categories: create, update, remove, get, getAll and getFilter. Such classification provides the modernization engineer with a common vocabulary to identify unequivocally the list of data operations performed by a controller. Such identification is necessary to match properly those operations with the correct HTTP method to use in a REST REQUEST. The table 2 shows such mapping.

Generating REST Requests Mappings. Following a similar approach to the generation of REST URIs, our process generates all the possible combinations of URIs and HTTP methods, producing four different request mappings for each REST URI. Next, that collection is filtered by the set of available operations, so only the REST requests with a proper matching with a legacy controller is maintained, while the rest is removed. Table 2 summarizes the results obtained for the REST URIs selected in our illustrative example.

Delegating REST Requests Processing to Legacy Struts Actions. It comprises the following steps:

1. The definition of the mappings between REST requests and existing controllers. This mapping is derived from the previously obtained information regarding the relation request-controller. Therefore, for instance and according to table 2 , the management of an HTTP GET request for the path */paper/{id}* will be delegated to the *paperAction.details* controller.
2. The conversion of the request input data (parameters or message body) to the corresponding data format expected by the controller. Since legacy controllers are reused in the process, the context that they expect must be built in order to ensure a right execution. Thus, it could be needed, for instance, that a particular object that represents the information sent by an HTML form (expected by the controller) must be populated with the data obtained from the REST request (e.g. in JSON format). The generation of this context needed by the controller requires to know, on the one hand, the parameters of the source request and, on the other hand, the presentation objects related with the source request. All this information may be also

Table 2. REST configuration process result summary

URIs	Operations	Actions	REST Requests
/paper	Create	paperAction.new	POST
	Read	paperListAction	GET
	Update		
	Delete		
/paper/{id}	Create		
	Read	paperAction.details	GET
	Update	paperAction.save	PUT
	Delete	paperAction.delete	DELETE
/paper/{id}/coauthor	Create	coauthorAction.new	POST
	Read	paperAction.details	GET
	Update		
	Delete		
/paper/{id}/coauthor/{id}	Create		
	Read		
	Update	coauthorAction.save	PUT
	Delete	coauthorAction.delete	DELETE

easily gathered from the MIGRARIA MVC model of the legacy application by analyzing the structure of the view that contains the source request. In our running example, the *new* request contained in the *paperCreated* view is related with the input presentation object named *paperForm*. Therefore, the object expected by the controller must be generated to handle that input presentation object. Figure 7 presents, on its left side, the structure of the JSON message for the creation of a new paper, and on its right side, the expected object by the Action *paperCreate* which will be created and initialized conveniently from the JSON message received.

3. The generation of the response in a particular data format (e.g. in JSON format) instead of generating an HTML page as response, REST API clients usually expect responses in a particular data format (JSON, XML) in order to properly manage them at client tier. Therefore, the response generated by the legacy application, as an HTML page, is not useful in this case and it must be replaced by a response generated in the previously accorded data format. In MVC-based Web development frameworks, controllers used to delegate to a different framework component the composition of the final response, i.e. a presentation template management system. Thus, controller main responsibilities are to generate the needed output objects to populate the template selected to compose the response. Based on this approach, we may know the presentation objects that a particular controller will generate (and their structure) just analyzing the MIGRARIA MVC model. As an example, the *paperActions.details ControlFlow* generates the *paper* output presentation object to compose the response by means of the template view *paperDetails*. The structure of the *paper* presentation object, defined within

```
{
  "paperForm" : {
    "id" : ""
    "title" : "My New Paper"
    "abst" : "Abstract..."
    "fileLocalPath" : "mypaper.pdf"
    "author" : "5"
    "track" : "3"
  }
}
```

```
public class PaperForm extends ActionForm {

    private Long id;
    private String title;
    private String abst;
    private String fileLocalPath;
    private long author;
    private long track;
```

Fig. 7. JSON input to form bean

```
<h2>Paper <bean:write name="paper" property="id" /> details</h2>
<table id="paperDetailsTable">
    <tr>
        <th>Title</th>
        <th>Abstract</th>
        <th>Track</th>
    </tr>
    <tr>
        <td><bean:write name="paper" property="title" /></td>
        <td><bean:write name="paper" property="abst" /></td>
        <td><bean:write name="paper" property="track.name" /></td>
    </tr>
</table>
```

```
{
  ...
  "paper" : {
    "id" : "12"
    "title" : "Example Paper"
    "abst" : "Abstract..."
    "track.name" : "Example Track"
  }
  ...
}
```

Fig. 8. Output bean to JSON

the *paperDetails* template view, is shown in Figure 8 (left side). Based on this information the final response is generated in the agreed data format, JSON in this example.

Generating REST API Configuration. As previously stated, our approach consists on extending the base Web development framework to incorporate REST capabilities and automatically configuring it to provide an API to the underlying legacy system. The above sections describe in detail both. In this work, the (semi)automatic configuration is generated by means of a model to model transformation described in ATL [5]. That transformation implements the process depicted above and generates an XML file as output. The schema of such file has been defined as an extension of the one used by the configuration file of Struts v1.3, so its deserialization can be implemented as an extension of the actual Struts configuration module. For the shake of brevity, we do not explain the concrete extensions performed.

5 Related Work

Although other model-based approaches have been proposed to define a REST API from a legacy application [6,3,11], to our knowledge none of them is contextualized within a complete model-driven modernization process and nor provides

extensions of legacy technology, i.e. Struts v1.X, as the approach presented in this work.

Different approaches and case studies exist about generating a REST API from a legacy application. [6] describes a common process for re-engineering legacy systems to RESTful services which is focused on data-driven systems. The process description is performed at a conceptual level. Parts of their process have been used to derive the process presented in this work. Our approach has extended and adapted that process to a concrete scenario of the modernization of a web application. We also provide a realization of this process in a specific web development technology. Both approaches share a common focus on URIdefinition. [3] describes the case study of an existing legacy application for Internet bidding generalized and replaced by a RESTful API. Instead of reusing existing implementations the auhtors designed a new protocol and implemented it in different languages. The main common points is the focus on data-centric CRUD operations. Finally,[11] develops an API oriented to resources and CRUD operations, by the transformation of an object-oriented model of a legacy application into a resource-oriented one, hereafter generating the URIs is a simple step (M2M transformation on the contrary to our M2T translation).

6 Conclusions and Future Work

This work tackles a part of the modernization process defined within the MIGRARIA project which faces the evolution of the presentation of a legacy Web system towards a Web 2.0 new RIA client. This process requires not only the generation of the new RIA client but also the creation of a new connection layer for enabling the data interchange between the client and the original system functionalities. In this paper we have presented how this connection layer is generated. In particular, the generation of a REST service layer is described that provides the client with an API to handle the user requests. By means of a running example (excerpt of a fully-functional case study) we have detailed the activities that comprise the process that are mainly divided into two main steps: a) framework-dependant REST API specification and b) application-dependant REST API specification. While the former is performed just once and reused for applications developed in the same MVC Web framework, the latter depends on the particular application being modernized so that, in this case, model-driven techniques come to scene to systematically automate the process providing benefits regarding reusability and effort reduction. Note that the work presented here takes as input the results obtained in the previous steps of MIGRARIA that generates a conceptual representation of the legacy system in terms of models (conformed to a new defined metamodel).

As main lines for future work on MIGRARIA we consider the following: 1) improve the tool support by developing a visualization tool integrated into the MIGRARIA tool chain and that eases the developer's intervention in the process; 2) enrich the process with domain semantic information to improve the resource and relations identification and the REST request generation.

References

1. Barros, A.P., Dumas, M.: The rise of web service ecosystems. IT Professional 8(5), 31–37 (2006)
2. Ceri, S., Fraternali, P., Matera, M., Maurino, A.: Designing multi-role, collaborative web sites with webml: a conference management system case study. In: 1st Workshop on Web-oriented Software Technology (2001)
3. Engelke, C., Fitzgerald, C.: Replacing legacy web services with RESTful services. In: Proceedings of the First International Workshop on RESTful Design - WS-REST 2010, vol. 5, p. 27 (2010)
4. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
5. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
6. Liu, Y., Wang, Q., Zhuang, M., Zhu, Y.: Reengineering Legacy Systems with RESTful Web Service. In: 2008 32nd Annual IEEE International Computer Software and Applications Conference (2100219007), pp. 785–790 (2008)
7. Mesbah, A., Van Deursen, A.: An architectural style for ajax. In: The Working IEEE/IFIP Conference on Software Architecture, WICSA 2007, p. 9. IEEE (2007)
8. Rodríguez-Echeverría, R., Conejero, J.M., Clemente, P.J., Pavón, V.M., Sánchez-Figueroa, F.: Model Driven Extraction of the Navigational Concern of Legacy Web Applications. In: Grossniklaus, M., Wimmer, M. (eds.) ICWE 2012 Workshops. LNCS, vol. 7703, pp. 56–70. Springer, Heidelberg (2012)
9. Rodríguez-Echeverría, R., Conejero, J.M., Clemente, P.J., Preciado, J.C., Sánchez-Figueroa, F.: Modernization of legacy web applications into rich internet applications. In: Harth, A., Koch, N. (eds.) ICWE 2011. LNCS, vol. 7059, pp. 236–250. Springer, Heidelberg (2012)
10. Rossi, G., Pastor, O., Schwabe, D., Olsina, L.: Web Engineering: Modelling and Implementing Web Applications (Human-Computer Interaction Series) (October 2007)
11. Szymanski, C., Schreier, S.: Case Study: Extracting a Resource Model from an Object-Oriented Legacy Application (2012)
12. Ulrich, W.: Modernization Standards Roadmap, pp. 46–64 (2010)