EMF-REST: Generation of RESTful APIs from Models

Ed-Douibi Hamza, Javier Luis Cánovas Izquierdo, Abel Gómez, Massimo Tisi, Jordi Cabot

EMF-REST: Generation of RESTful APIs from Models

Hamza Ed-douibi
UOC
Av. Tibidabo, 39-43
Barcelona, Spain
hed-douibi@uoc.edu

Javier Luis Cánovas Izquierdo
UOC
Av. Tibidabo, 39-43
Barcelona, Spain
jcanovasi@uoc.edu

Abel Gómez
Universidad de Zaragoza
María de Luna, 1
Zaragoza, Spain
abel.gomez@unizar.es

Massimo Tisi
AtlanMod team (Inria, Mines Nantes, LINA)
4, rue Alfred Kastler
Nantes, France
massimo.tisi@inria.fr

Jordi Cabot
ICREA - UOC. Internet Interdisciplinary Institute
Av. Carl Friedrich Gauss, 5
Castelldefels, Spain
jcabot@icrea.cat

ABSTRACT

In the last years, there has been an increasing interest for Model-Driven Engineering (MDE) solutions in the Web. Web-based modeling solutions can leverage on better support for distributed management (i.e., the Cloud) and collaboration. However, current modeling environments and frameworks are usually restricted to desktop-based scenarios and therefore their capabilities to move to the Web are still very limited. In this paper we present an approach to generate Web APIs out of models, thus paving the way for managing models and collaborating on them online. The approach, called EMF-REST, takes Eclipse Modeling Framework (EMF) data models as input and generates Web APIs following the REST principles and relying on well-known libraries and standards, thus facilitating its comprehension and maintainability. Also, EMF-REST integrates model and Web-specific features to provide model validation and security capabilities, respectively, to the generated API.

CCS Concepts

•**Software and its engineering** → *Development frameworks and environments; Software development techniques;*

Keywords

Model-Driven Engineering; Model-Driven Web Engineering; Domain-Specific Languages; EMF; REST

1. INTRODUCTION

Model-Driven Engineering (MDE) methods and techniques have been matured along recent years. In the last years, the emergence of distributed architectures, specially Cloud-based ones, and mobile devices have promoted the development of model-based applications on the Web and over portable devices. Current modeling environments (e.g., Xtext, Epsilon or EMFText) and frameworks (e.g., the plethora of modeling facilities in Eclipse such as EMF or GMF) have successfully contributed to the broad use of MDE techniques. However, model-based applications on the Web require relying on Web technologies rather than on current heavyweight desktop environments and frameworks.

Web services offer a suitable solution to expose model-driven environments for remote access. Among the different approaches for designing distributed services (e.g., SOAP, WSDL or WS-* specifications), there is a rising trend to use lightweight solutions based on plain HTTP, referred to as REpresentational State Transfer (REST) [9] services. REST proposes the development of stateless distributed services and relies on simple URIs and HTTP verbs to make the Web services broadly available for a number of front-end devices. With the increasing interest in developing model-based applications on the Web, some approaches have appeared [12, 16, 20] to cope with the model-driven generation of Web services, however, their support to generate RESTful Web APIs is very limited.

In this paper we present EMF-REST, an approach that leverages on MDE techniques to generate RESTful Web APIs out of EMF models, thus promoting model management in distributed environments. The generated RESTful Web API relies on well-known libraries and standards with the aim of facilitating its understanding and maintainability. Unlike other existing MDE-based approaches targeting the generation of Web services, ours provides a direct mapping to access data models by means of Web services following the REST principles. Additionally, EMF-REST takes advantage of model and Web-specific features such as model validation and security, respectively.

By using EMF-REST, developers have the ground to leverage on Platform-as-a-Service (PaaS) providers, in which model management would take advantage of their scalabil-

ity capabilities. Also, adopting a Web-based solution would promote the collaboration between modelers, thus facilitating the collaborative development of new software models.

The remainder of this paper is structured as follows. Section 2 presents some background of REST and MDE. Section 3 describes how we devised the mapping between EMF and REST principles, while Section 4 describes the additional EMF-REST features. Section 5 presents the technical architecture of the generated REST API. Section 6 describes the steps we followed to generate the API. Section 7 discusses some related work. Finally, Section 8 concludes the paper and presents the future work.

2. BACKGROUND

2.1 The MDE paradigm

The MDE paradigm emphasizes the use of models to raise the level of abstraction and to automate the development of software. Abstraction is a primary technique to cope with complexity, whereas automation is the most effective method for boosting productivity and quality [19].

Modeling languages express models at different abstraction levels, and are defined by applying metamodeling techniques [5]. In a nutshell, models are defined according to the semantics of a model for specifying models, the so called *metamodel*. A model that respects the semantics defined by a metamodel is said to *conform to*/*to be an instance of* such a metamodel.

The Eclipse Modeling Framework (EMF) [2] has become the main reference for modeling in Eclipse [8]. Among its features, EMF allows creating metamodels – by using the Ecore language, a subset of the UML class diagrams – and their instances. Along this paper, we refer to metamodels as *Ecore models*, and their instances as *model instances*. Ecore can be considered as an implementation of Meta-Object Facility (MOF) [14], a modeling and metamodeling standard developed by the Object Management Group (OMG). Additionally, EMF provides a generative solution which constructs Java APIs out of those models to facilitate their management, thus promoting the development of domain-specific applications.

On the other hand, model transformations generate software artifacts from models, either directly by model-to-text transformations (e.g., using languages such as EGL[1] or JET[2]) or indirectly by intermediate model-to-model transformations (e.g., using languages such as ATL[3] or ETL[4]). By means of modeling languages and model transformations, it is possible to increase both the level of abstraction and provide automation in MDE, respectively.

In EMF-REST we use EMF and we apply a set of model-to-text transformations using JET and EGL templates to generate a Web application exposing the management of Ecore models by the mean of REST APIs

2.2 REST principles

In 2000, Roy Fielding identified specific design principles that led to the architectural style known as *REpresentational State Transfer* (REST) [9]. By relying on the HTTP protocol, this architectural style consists of several constraints to address separation of concerns, visibility, reliability, scalability and performance. REST principles are defined as:

Addressable resources — Each resource must be addressable via a Uniform Resource Identifier (URI).

Representation-oriented — A resource referred by one URI may have different representation formats (e.g., JSON, XML, etc.).

Statelessness — Servers cannot hold the state of a client session Instead, data representation formats provide information on how to manage the state of the application for each client (e.g., using embedded URIs).

Uniform and Constrained Interface — A small set of well-defined methods are used to manipulate resources (i.e., HTTP verbs).

The last two principles are maybe the most distinguishing features of REST from other Web services specifications. According to these principles, each request is treated as an independent transaction and must only rely on the set of operations of the HTTP protocol. HTTP methods are used in REST as follows:

GET is used to retrieve a representation of a resource. It is a read-only, *idempotent* and *safe* operation.

PUT is used to update a reference to a resource on the server and it is *idempotent* as well.

POST is used to create a resource on the server based on the data included in the body request. It is the only *nonidempotent* and *unsafe* operation of HTTP.

DELETE is used to remove a resource on the server. It is *idempotent* as well.

HEAD is similar to GET but returning only a response code and the header associated with the request.

OPTIONS is used to request information about the communication options of the addressed resource (e.g., security capabilities such as CORS).

Being a collection of principles rather than a set of standards, several resources on best practices and recommendations were written to help developers to write RESTful Web services. In order to generate a high-quality RESTful Web API, we apply in EMF-REST the best practices described in [13].

3. MAPPING EMF AND REST PRINCIPLES

The first step to build EMF-REST is to align the principles behind the MDE/EMF and REST worlds. We rely on EMF to represent the models from which the RESTful Web APIs are generated. As models and their instances are managed by the corresponding APIs provided by the framework (i.e., Ecore and EObject APIs, respectively), we need to define a mapping between such APIs and the REST principles presented before. In this section we explain how we map EMF with each principle.

To illustrate the approach, we will use a running example consisting on creating a distributed application aimed

[1] `http://www.eclipse.org/epsilon/doc/egl/`
[2] `https://eclipse.org/modeling/m2t/`
[3] `https://eclipse.org/atl/`
[4] `http://www.eclipse.org/epsilon/doc/etl/`

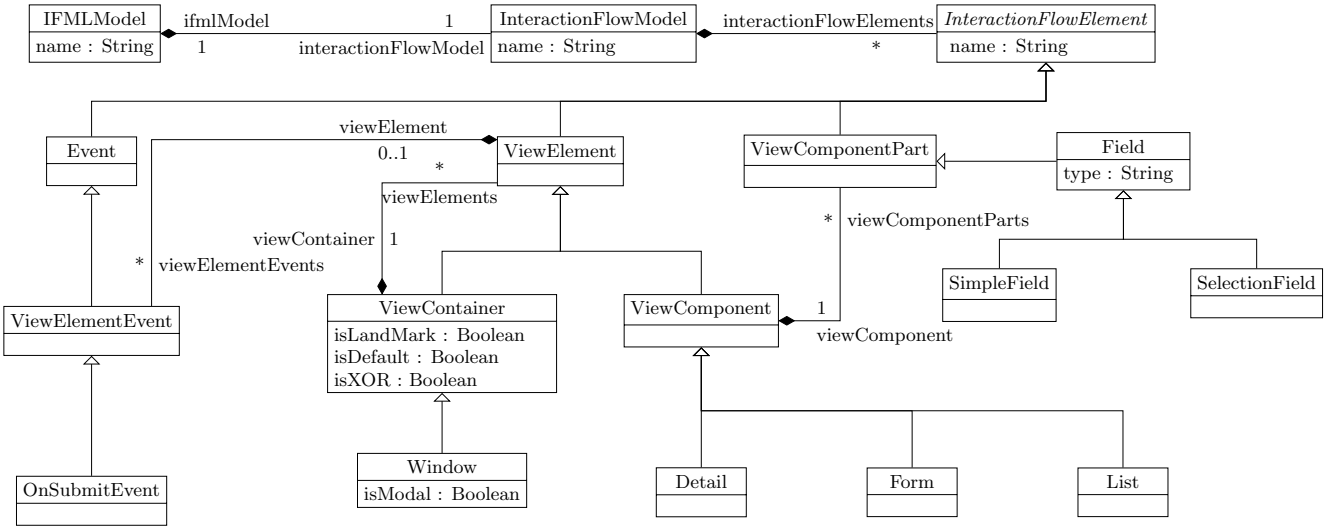


Figure 1: Simple Ecore model of an IFML subset.

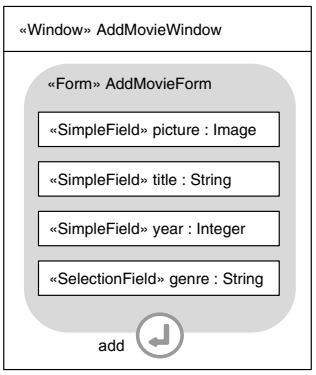


Figure 2: IFML model of AddMovie

at managing IFML (Interaction Flow Modeling Language) models. IFML is an OMG standard language designed for expressing the content, user interaction and control behavior of the front-end applications developed for systems such as Computers and Mobile phones [6]. Figure 1 shows an excerpt of the IFML metamodel. As can be seen, *IFMLModel* is the top-level container of all the model elements and represents an IFML model. It contains an *InteractionFlowModel* which is the user view of an application. The concepts extending *ViewContainer*, *ViewComponets*, *ViewComponentPart*, and *ViewElementEvent* represent the visual elements of an IFML model.

Figure 2 shows an IFML model which consists of a form allowing to add a movie. The model is composed of (i) a *Window* container named *AddMovieWindow*, (ii) a *Form* component named *AddMovieForm*, (iii) a list of fields of types *SimpleField* and *SelectionField* representing the elements of the form, and finally (iv) a *ViewElementEvent* of type *OnsubmitEvent* allowing to submit the form. In what follows we will see how we would allow creating the *AddMovieForm* form by calling a REST API generated from the IFML model following the REST principles.

3.1 Addressable Resources

Models in EMF are addressed via a URI, which is a string with a well-defined structure as shown in the expression (1). This expression contains three parts specifying: (1) a *scheme*, (2) a *scheme-specific part* and (3) an optional *fragment*. The scheme is the first part separated by the *":"* character and identifies the protocol used to access the model (e.g., *platform*, *file* or *jar*). In Eclipse we use *platform* for URIs that identify resources in Eclipse-specific locations, such as the workspace. The scheme-specific part is in the middle and its format depends on the scheme. It usually includes an *authority* that specifies a host, the *device* and the *segments*, where the latter two constitute a local path to a resource location. The optional fragment is separated from the rest of the URI by the # character and identifies a subset of the contents of the resource specified by URI, as we will illustrate below. The expression (2) shows an example of a platform-specific URI which refers to the *AddMovie* model, represented as a file *AddMovie.xmi* contained in a project called *project* in Eclipse workspace. It is important to note that in EMF model instances include a reference to the Ecore model they conform to.

```
[scheme:][scheme-specific-part][#fragment]                    (1)
platform:/resource/project/AddMovie.xmi                       (2)
```

We map the previous URI to a Web URL as follows. The base URL pattern of a model instance is defined by the expression (3). In the pattern, the part *https://*[applicationLink]/rest is the URL of the Web application, *modelId* is the identifier of the model (e.i., the Ecore model) and *ModelInstanceId* is the identifier of the model instance being accessed (the XMI file). The URL (4) represents an example to retrieve the IFML model used in the example. As can be seen, while the URI can address a file representing a model instance (where a reference to the Ecore model is included), the URL requires indicating the identifier of both the Ecore model and the model instance.

```
https://[applicationLink]/rest/[ModelId]/[ModelInstanceId]    (3)
https://example.com/rest/IFMLModel/AddMovie                   (4)
```

This URL acts as the entrypoint for a particular model instance and points to its root element, which is normally the case in EMF. When the model instance has more that one root, we point at the first.

Once pointing to the root of a model instance, addressing a particular element of the model in the EMF is done by using the part *fragment* in (1). The navigation is done using the reference names in the Ecore model. For instance, the concept *IFMLModel* has the reference *interactionFlowModel* to access the *InteractionFlowModel*. Using the EMF API, the URI is shown in (5), while using the Web API, the URL is shown in (6).

```
platform:/resource/project/AddMovie.xmi#//@interactionFlowModel    (5)
```

```
https://example.com/rest/IFMLModel/AddMovie/interactionFlowModel    (6)
```

Depending on the cardinality of the reference this will return a specific element – if it is single-valued (like in the case of *interactionFlowModel*) – or a collection of elements – if it is multi-valued. Accessing a specific element contained in a collection can be done using (i) the identifier of the element or (ii) its index in the list. Also, when navigating through the references contained in elements being subclasses of a hierarchy, the appropriate filtering is done on the fly. For instance, the URI (7) retrieves the element representing *title* in EMF, while in EMF-REST it is done using the call (8). Note how the latter navigates through the reference *viewElements*, which is only included in *ViewContainer* element. To identify an element, we rely on the *identifier* flag provided by Ecore, which allows setting the attribute acting as identifier for a given class[5].

```
platform:/resource/project/AddMovie.xmi#title    (7)
```

```
https://example.com/rest/IFMLModel/AddMovie/
interactionFlowElements/AddMovieWindow/viewElements/
AddMovieForm/viewComponentsParts/title    (8)
```

On the other hand, the call (9) will retrieve the first element of the collection of *viewComponentsParts* in the EMF API. In our approach, it is done by adding the parameter *index* in the URL as illustrated in the call (10).

```
platform:/resource/project/AddMovie.xmi#//.../@viewComponentsParts.0    (9)
```

```
https://example.com/rest/IFMLModel/AddMovie/
interactionFlowElements/AddMovieWindow/viewElements/
AddMovieForm/viewComponentsParts?index=0    (10)
```

3.2 Representation-Oriented

By default, EMF persists models using the XMI representation format. Our approach relies also on XMI to save the models internally, however, models are offered to clients using both JSON-based and XML-based storages in order to comply with the representation-oriented principle of the REST architecture.

For the JSON, we adher to the following structure. Model concepts are represented as JSON objects containing key/-value pairs for the model attributes/references. Keys are the name of the attribute/reference of the concept and values are their textual representation in one of the datatypes supported in JSON (i.e., string, boolean, numeric, or array). For attributes, their values are mapped according to the corresponding JSON supported datatype or String when there is not a direct correspondence (e.g., float-typed attributes). When the attribute is multi-valued, its values are represented using the array datatype. For references, the value is the URI of the addressed resource within the server (if the reference is multi-valued, the value will be represented as an array of URIs). Listing 1 shows an example of the content format in JSON. Note that references containing a set of elements from model hierarchies are serialized as a list of JSON objects corresponding to their dynamic type (see *viewComponentParts* reference including *SimpleField* and *SelectionField* JSON objects).

In XML, model concepts are represented as XML elements including an XML element for each model attribute/reference. Attribute values are included as string values in the XML element representing such attribute, references are represented according to their cardinality. If the reference is single-valued, the resulting XML element will include only

[5]When the *identifier* flag is not used, the fallback behavior looks for an attribute called *id*, *name* or having the *unique* flag activated.

Listing 1: Partial JSON representation of the example model

```
1  {
2    "form":{
3      "name":"addMovieForm",
4      "viewComponentParts":{
5        "simpleField":[{
6          "uri":"https://example.com/rest/IFMLModel/
                AddMovie/interactionFlowElements/
                AddMovieWindow/viewElements/
                AddMovieForm/viewComponentsParts/
                picture"},{
7          "uri":"https://example.com/rest/IFMLModel/
                AddMovie/interactionFlowElements/
                AddMovieWindow/viewElements/
                AddMovieForm/viewComponentsParts/title
                "},...],
8        ...
9      },
10     "viewElementEvents":{
11       "onsubmitevent":{"uri":"https://example.com/
               rest/IFMLModel/AddMovie/
               interactionFlowElements/AddMovieWindow/
               viewElements/AddMovieForm/
               viewelementevents/add"}
12     }
13   }
14 }
```

Listing 2: Partial XML representation of the example model

```
1  <form>
2    <name>AddMovieForm</name>
3    <viewComponentParts>
4      <simpleField>
5        <uri>https://example.com/rest/IFMLModel/
               AddMovie/interactionFlowElements/
               AddMovieWindow/viewElements/AddMovieForm
               /viewComponentsParts/picture</uri>
6      </simpleField>
7      <simpleField>
8        <uri>https://example.com/rest/IFMLModel/
               AddMovie/interactionFlowElements/
               AddMovieWindow/viewElements/AddMovieForm
               /viewComponentsParts/title</uri>
9      </simpleField>
10     ...
11   </viewComponentParts>
12   <viewElementEvents>
13     <onSubmitEvent>
14       <uri>https://example.com/rest/IFMLModel/
               AddMovie/interactionFlowElements/
               AddMovieWindow/viewElements/AddMovieForm
               /viewElementEvents/add</uri>
15     </viewElementEvents>
16   </viewElementEvents>
17 </form>
```

the URI of the addressed resource in the server. On the other hand, if the reference is multi-valued, the resulting XML element will include a set of XML elements including the URIs addressing the resources. Listing 2 shows an example of the content format in XML format.

3.3 Uniform and Constrained Interface & Statelessness

EMF supports loading, unloading and saving model instances after their manipulation. In our approach, these operations are managed by the application server. Models are loaded (and unloaded) dynamically as resources when running the application managing the Web API, and they are saved after each operation is done, thus conforming to the REST statelessness behavior.

To manipulate model instances, EMF enables the basic CRUD (i.e., create, read, update and delete) operations over model instances by means of either the EMF generated API or the EObject API. We map the same CRUD operations into the corresponding HTTP methods (*POST*, *GET*, *PUT*, and *DELETE*). For instance, Listing 3 shows the code to modify the name of the form called *AddMovieForm* using EMF generated API for the AddMovie model. The same operation can be done on our Web API by sending the PUT HTTP method containing the JSON representation of the new *Form* model element, as shown in Listing 4.

Table 1 shows how each CRUD operation is addressed along with several URL examples. The first column of the table describes the operations. As can be seen, the first two rows represent operations over collections, enabling adding new elements (see first row) and reading their content (see second row). The rest of the rows describe operations over either individual elements of a collection (see cases 1 and 2 of these operations) or elements contained in a single-valued reference (see case 3). The second column shows the correspondent HTTP method for each operation while the third column presents the corresponding URL for each case. Finally, the last column includes a small model to better illustrate the cases considered in the table.

4. ADDITIONAL EMF-REST FEATURES

We provide also support for validation and security aspects in the generated RESTful Web API.

Listing 3: Update the attribute of a concept using EMF generated API.

```
1 ...
2 addMovieFormObj.setName("toto"); //
      addMovieFormObj is of type Form
3 ...
```

Listing 4: HTTP call and JSON representation to update the name of the addressed form.

```
1 PUT https://example.com/rest/IFMLModel/AddMovie/
      interactionFlowElements/AddMovieWindow/
      viewElements/AddMovieForm
2 {"form":{
3         name:"toto"
4         }
5 }
```

4.1 Validation

Support for validating the API data calls is pretty limited in current web technologies. The most relevant one for our scenario would be the Bean Validation specification to enforce the validation of *Java Beans*. However, this specification can only ensure that fields follow certain constraints (e.g., a field is not `null`) and cannot satisfy complex validation scenarios for model integrity (e.g., a *form* must have at least one *field*). On the other hand, MDE provides specific support for validating models, for instance, the Object Constraint Language [21], a language complementing UML [17] that allows software developers to write complex constraints over object models. Thus, we employ OCL to define constraints as annotations in the model elements.

OCL annotations can be attached to concepts in the model as invariants. An example on the IFML example model is shown in Figure 3. As can be seen, concepts include a set of invariants inside the annotation *OCL* plus the annotation *Ecore/constrains* which specifies the invariants to execute. Invariants are checked each time a resource is modified (i.e., each time the Web API is called from a Web-based client using the `POST`, `PUT` or `DELETE` methods). This validation scheme is imposed to comply with the stateless property of REST architetures, however, it may involve some design constraints when creating the model. In those cases where models cannot be validated each time they are modified (e.g., creating model elements requires several steps to fulfill cardinality constraints), we allow this validation process to

Table 1: Supported operations in the generated API.

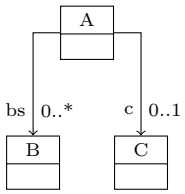
Operation	HTTP Method	URL	Model
CREATE and add element to the collection	POST	`.../a/bs`	
READ all the elements from the collection	GET		
READ the element (1) identified by <id>, (2) in the <i> position of the collection, or (3) the element c	GET	(1) `.../a/bs/<id>` (2) `.../a/bs?index=<i>` (3) `.../a/c`	
UPDATE the element (1) identified by <id>, (2) in the <i> position of the collection, or (3) the element c	PUT		
DELETE the element (1) identified by <id>, (2) in the <i> position of the collection, or (3) the element c	DELETE		

Figure 3: Annotations on an excerpt of the example model.

be temporary be deactivated. The results of the validation process are mapped into the corresponding HTTP response messages (i.e., using status codes)

4.2 Security

While there is little support for security definition and enforcement from the MDE side, we have plenty of support from web technologies. In particular, our approach allows designers to provide some security annotations on the model that are then translated into security restrictions as described below. As part of the generation, we also create a separated admin view where additional security information (like users and passwords) can be maintained.

In order to secure a Web application, we have to: (i) ensure that only authenticated users can access resources, (ii) ensure the confidentiality and integrity of data exchanged by the client/server, and (iii) prevent unauthorized clients from abusing data. In order to address the previous requirements, we rely on a set of security protocols and services provided by *Java EE* which enable encryption, authentication and authorization in Web APIs, as we will explain in the following.

Encryption: The Web defines HTTPS protocol to add the encryption capacities of SSL/TLS to standard HTTP communication. We enforce the use to HTTPS to communicate with its services.

Authentication: We rely on basic authentication to provide the authentication mechanism since it is simple, widely supported, and secure by using HTTPS. The basic authentication involves sending a Base64-encoded username and password within the HTTPS request header to the server.

Authorization: While the authentication is enabled by the protocol/server, the authorization is generally provided by the application, which knows the permissions for each resource operation. We use a simple role-based mechanism to support authorization in the generated Web API. Roles are associated to users (i.e., authentication) and operations in the Web API (i.e., authorization). In our approach roles are assigned to resources by adding annotations to the model. Figure 3 illustrates the use of these annotations (e.g., see annotation *Ecore/roles* in the *Form* concept).

5. EMF-REST API ARCHITECTURE

To implement the features described in the previous sections, we devised the application architecture presented in Figure 4. This architecture can then be seamlessly accessed with a variety of clients.

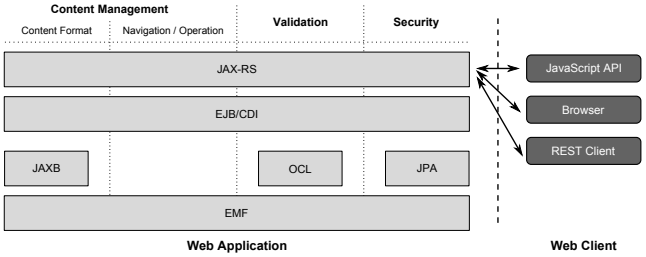


Figure 4: Architecture of the generated application.

The Web application is split into three main components according to the functionality they provide: (1) content management, (2) validation and (3) security. The application relies on EMF as modeling framework and uses the following additional frameworks/specifications for each component, respectively: (1) *Java Architecture for XML Binding (JAXB)* to enable the content format support, (2) Eclipse OCL framework to provide validation before updating the model, (3) *Java Persistence API (JPA)* to provide security support by storing the system users and their permissions in an embedded database. The Web application also leverages on *Enterprise Java Bean (EJB)*, *Context dependency Injection (CDI)* and *Java API for Representational State Transfer (JAX-RS)* specifications. EJBs enable rapid and simplified development of distributed, transactional, secure and portable applications. They are in charge of loading the EMF resources from the persistent storage and providing the necessary methods to manage the resources (e.g., obtaining objects from the resource, removing objects) in a secure and transactional way. These EJBs are then injected into JAX-RS services using CDI technology. Thus, JAX-RS is used to expose EMF resources as Web services. In the remaining of the section we describe how all these technologies are used in each component.

5.1 Content Management

This component addresses the mapping between EMF and REST principles. It is in turn split into two subcomponents: (1) content format, which addresses the mapping of the second REST principle (i.e., Representation-Oriented); and (2) navigation/operation, which addresses the rest of the REST principles.

Regarding the content format, we enrich the EMF generated API with JAXB[6] annotations, which enable the support for mapping Java classes to XML/JSON (i.e., marshalling/unmarshalling Java object into/from XML/JSON documents). The Listing 5 shows an example of the use of JAXB annotations to produce the corresponding representation in JSON (as shown in Listing 1) and XML (as shown in Listing 2). As can be seen, each concept class is mapped to an `XmlRootElement` element, while either `XmlElement` or `XmlElementWrapper` elements are used to map the attributes or references of the class, respectively. Other annotations are used to deal with the references and inheritance. For instance, `XmlJavaTypeAdapter` and `XmlAnyElement` are used to associate a reference of an element with the corresponding representation.

Navigation and operations are enabled by using JAX-RS, which provides a set of Java APIs for building Web services

[6] `https://jaxb.java.net/`

Listing 5: Part of the ViewComponent concept.

```
1  @XmlRootElement (name="viewcomponent")
2  @XmlSeeAlso ({ViewComponentProxy.class,//...
3  })
4  public  class ViewComponentImpl extends
       ViewElementImpl
5    implements ViewComponent {
6    //...
7    @XmlElementWrapper(name =  "viewComponentParts")
8    @XmlAnyElement(lax=true)
9    @XmlJavaTypeAdapter(value=
         ViewComponentPartAdapter.class)
10   public EList<ViewComponentPart>
         getViewComponentParts() {
11     if (viewComponentParts == null) {
12       viewComponentParts = new
             EObjectContainmentWithInverseEList<
             ViewComponentPart>(ViewComponentPart.
             class, this, IfmlPackage.
             VIEW_COMPONENT__VIEW_COMPONENT_PARTS,
             IfmlPackage.
             VIEW_COMPONENT_PART__VIEW_COMPONENT);
13     }
14     return viewComponentParts;
15   }
16   //...
17 }
```

conforming to the REST style. Thus, this specification defines how to expose POJOs as Web resources, using HTTP as network protocol. For each concept (e.g., *IFMLModel*) a resource will be created (e.g., `IFMLModelResource`) annotated with `@Path` (e.g., `@Path("IFMLModel")`). The `@Path` annotation has the value that represents the relative root URI of the addressed resource. For instance, if the base URI of the server is `http://example.com/rest/`, the resource will be available under the location `http://example.com/rest/IFMLModel`. To produce a particular response when a request with GET, PUT, POST and DELETE is intercepted by a resource, resource methods are annotated with `@GET`, `@PUT`, `@POST` and `@DELETE` what are invoked for each corresponding HTTP verb.

5.2 Validation

Our approach leverages on Eclipse OCL[7] to validate the data by means of annotations including the constrains to check the model elements. The generated API relies on the provided APIs for parsing and evaluating OCL constraints and queries on Ecore models. When constraints are not satisfied, the validation process will fire an exception that will be mapped by JAX-RS into an HTTP response including the corresponding message indicating the violated constraint.

5.3 Security

We rely on the combination of Java EE and JAX-RS for the authentication and authorization mechanisms by using the concept of role, while encryiption is provided by using HTTPS. To enable authentication, the deployment descriptor of the WAR file (i.e., `WEB-INF/web.xml`) has been modified to include the security constraints (i.e., `<security-constraint>`) defining the access privileges. Assigning permissions for HTTP operations based on the roles provided in the model is done by using the `@RolesAllowed` annotation. For example, as shown before, Figure 3 shows that the role allowed for the *Form* concept is `admin`.This will

[7] `http://www.eclipse.org/modeling/mdt/?project=ocl`

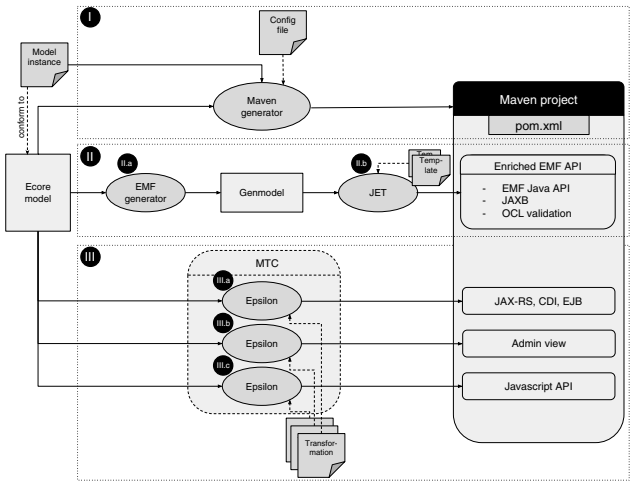


Figure 5: EMF-REST generation process.

restrict access to the resource to the users having the role `ADMIN`. To express this in the generated API, the annotation `@RolesAllowed({"ADMIN"})` is placed on top of `FormResource`. If no role is assigned to a concept, a `@PermitAll` annotation is placed on the resource class meaning that all security roles are permitted to access this resource. Note that security roles assigned to a resource are not inherited by its sub-resources.

To manage the list of users and their roles, we generate an admin view that allows the manager of the API to add, edit and remove users. All created users have a default role (i.e., *user*) allowing them to access unannotated concepts. The manager can assign more roles to a user in order to grant him/her access to a specific resource.

6. CODE GENERATION AND TOOL SUPPORT

In order to generate the REST APIs we created a Java tool available as an open-source Eclipse plugin [1]. Figure 5 shows the steps followed by the tool to generate the application starting from an initial Ecore model.

Step I of the process generates a Maven-based [8] project that serves as a skeleton of the application. Maven allows a project to be built by using the *Project Object Model* (POM) file, thus providing a uniform build system. The POM is initialized with the required library dependencies described in the previous section.

In Step II, the EMF code generation facility has been extended to include the required support for JAXB and validation. In particular, the JET templates used by EMF to generate Java code have been extended to produce the code corresponding to the JAXB annotations and the required methods to execute the OCL validation process.

Step III performs a set of model-to-text transformations using EGL to generate the remaining elements, including: (1) the JAX-RS, CDI and EJB implementation classes, (2) the admin view developed and (3) a simple JavaScript API to facilitate Web developers to build clients for the generated Web API. For each part of the application (e.g., JAX-RS resources, etc.), an EGL transformation template has been

[8] `http://maven.apache.org/`

implemented to generate the appropriate behavior according the input Ecore class. Since this step requires several transformations, the MTC tool [4] has been used to orchestrate the flow of the EGL templates.

7. RELATED WORK

Several efforts have been made to bring together MDE and Web Engineering. This field is usually referred to as Model Driven Web Engineering (MDWE) and proposes the use of models and model transformations for the specification and semiautomatic generation of Web applications [7, 10, 11, 15, 18, 22]. Mainly, data models, navigation models and presentations models are used for this purpose.

Some of these works provide support for the generation of Web services as well, but support for generation of RESTful APIs is very limited [12, 16, 20]. Moreover, these approaches require the designer to specifically model the API itself using some kind of tool-specific DSL from which then the API is (partially) generated. To the best of our knowledge, only Texo [3] provides an extension to EMF in order to support the creation of RESTful APIs but providing a proprietary solution which requires extending the user's metamodel with meta-data to drive the API generation. Instead, our approach is able to generate a complete RESTful API from any data model.

8. CONCLUSION

In this paper we have presented EMF-REST, an approach to generate RESTful Web APIs out of EMF models. We believe our approach fills an important gap between the modeling and Web technologies, thus enabling MDE practitioners to bring their models into the Web. EMF-REST has been released as an Eclipse-plugin and is publicly available [1].

As further work, we plan to extend the current support for security (e.g., a more fine-grained security mechanism) and validation (e.g., supporting transactions). Also, we would like to work on a small configuration DSL to help designers parameterize the style of the generated API (e.g., configuring the URIs to the resources). Existing approaches, like WADL[9] and RSDL[10], which also propose DSLs to describe Web APIs could be useful here. We are also interested in exploring the benefits of using EMF-REST in combination with client-side modeling environments, for instance, in Eclipse, thus enabling developers to deal with large EMF models in a transparent way (i.e., models in Eclipse that are remotely stored using an EMF-REST backend).

9. REFERENCES

[1] EMF-REST website. `http://emf-rest.com/` (last accessed on Dec. 2015).

[2] EMF. `http://www.eclipse.org/modeling/emf/` (last accessed on Dec. 2015).

[3] Texo. `http://wiki.eclipse.org/Texo` (last accessed on Dec. 2015).

[4] C. Alvarez and R. Casallas. MTC Flow: A Tool to Design, Develop and Deploy Model Transformation Chains. In *ACME workshop*, pages 1–9, 2013.

[5] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

[6] M. Brambilla, P. Fraternali, and et al. The interaction flow modeling language (ifml), version 1.0. Technical report, Object Management Group (OMG), http://www.ifml.org, 2014.

[7] S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): a Modeling Language for Designing Web Sites. *J. Comp. Netw.*, 33:137–157, 2000.

[8] Eclipse website. `http://eclipse.org` (last accessed on Dec. 2015).

[9] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000.

[10] N. Koch and S. Kozuruba. Requirements models as first class entities in model-driven web engineering. In *ICWE Workshops*, pages 158–169, 2012.

[11] B. Marco, J. Cabot, and M. Grossniklaus. Tools for Modeling and Generating Safe Interface Interactions in Web Applications. In *ICWE conf.*, pages 482–485, 2010.

[12] E. Maximilien, H. Wilkinson, N. Desai, and S. Tai. A Domain-Specific Language for Web APIs and Services Mashups. In *ICSOC conf.*, pages 13–26. 2007.

[13] B. Mulloy. *Web API Design - Crafting Interfaces that Developers Love*. Apigee, 2012.

[14] OMG MOF Specification. `http://www.omg.org/mof` (last accessed on Dec. 2015).

[15] X. Qafmolla and V. C. Nguyen. Automation of Web Services Development Using Model Driven Techniques. In *ICCAE conf*, volume 3, pages 190–194, 2010.

[16] J. M. Rivero, S. Heil, J. Grigera, M. Gaedke, and G. Rossi. MockAPI: An Agile Approach Supporting API-first Web Application Development. In *ICWE conf.*, pages 7–21, 2013.

[17] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[18] A. Schauerhuber, M. Wimmer, and E. Kapsammer. Bridging Existing Web Modeling Languages to Model-driven Engineering: A Metamodel for WebML. In *ICWE conf.*, 2006.

[19] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Softw.*, 20(5):19–25, 2003.

[20] N. A. C. Tavares and S. Vale. A model driven approach for the development of semantic restful web services. In *IIWAS conf.*, page 290, 2013.

[21] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

[22] WebRatio. `http://www.webratio.com` (last accessed on Dec. 2015).

[9] `http://www.w3.org/Submission/wadl/`

[10] `http://goo.gl/7wpf9y`