A Concept for Generating Simplified RESTful Interfaces

Markus Gulden
Zielpuls GmbH
Leopoldstraße 244
Munich, Germany
markus.gulden@zielpuls.com

Stefan Kugele
Institut für Informatik
Technische Universität München
Garching bei München, Germany
kugele@in.tum.de

ABSTRACT

Today, innovative companies are forced to evolve their software systems faster and faster, either for providing customer services and products or for supporting internal processes. At the same time, already existing, maybe even legacy systems are crucial for different reasons and by that cannot be abolished easily. While integrating legacy software into new systems in general is considered by well-known approaches like SOA (service-oriented architecture), at the best of our knowledge, it lacks of ways to make legacy systems available for remote clients like smart phones or embedded devices.

In this paper, we propose an approach to leverage heterogeneous (legacy) applications by adding RESTful web-based interfaces in a model-driven way. We introduce an additional application layer, which encapsulates services of one or several existing applications, and provides a unified, web-based, and seamless interface. This interface is modelled in our own DSL (domain-specific language), the belonging code generator produces productive Java code. Finally, we report on an case study proving our concept by means of an e-bike sharing service.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Languages*; D.2.12 [**Software Engineering**]: Interoperability—*Data mapping, interface definition languages*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*; H.5.4 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia—*Architectures*

General Terms

Design, Languages

Keywords

Data transfer object, domain-specific language, representational state transfer, service-orientated architecture

1. INTRODUCTION

The year 2007 represents an essential moment: with the release of the first iPhone, Apple made it to start a trend, which has influenced our private and professional life until today, and there is no end in sight. Before, mobile phones were only used for calling and texting. At best business men had smart phones with Internet connectivity, but only for writing emails and for synchronising appointments and contacts. The iPhone changed this: not least by forcing consumers to book a data plan with their conventional service plan, mobile Internet got mass appeal. Today, smart phones are unimaginable without Internet connection. There are several hundred thousands of apps available for multiple platforms, and most of them need to access the Internet for retrieving time tables, booking a car or a train ticket, accessing the users' cloud storage, downloading a new song, posting news on a social network, or looking for a restaurant nearby.

But what only was a gimmick for private users and hobbyists at the beginning, has now become crucial for modern enterprises, on one side for offering customer products and service, on the other side for supporting internal processes and business. Today, companies like banks or delivery services have to supplement their service by an app to stay attractive. Mobility services like car rental, train operators, or travel agencies have to be reachable on the way. But there are also totally new opportunities to provide context-dependent information via apps, for example special offers or advertisement from a shop, the customer is close by. All in all, consumer mobile app market has a volume of several billion US dollars per year. But also companies can get more efficient by equipping their employees with mobile apps, for example to support inventory tasks or the tracking of shipments.

Since apps often shall make already existing services available on smart phones, companies do not want to develop new systems, but want to make the existing ones reachable via the Internet. Certainly, most enterprise application landscapes consist out of more than one system, which are implemented with more than one, and sometimes even with legacy technologies.

Thereby, several issues are conceivable, like existing systems are not Internet-capable, interfaces are technologically and syntactically heterogeneous, or their data model simply might be more complex than needed by smart phone clients. Since changing the existing system normally is not an option, there must be another way to make existing interfaces accessible for external clients.

To address this problem, this paper proposes a model-driven approach to adapt existing systems and generate Java-based, web service-capable interfaces. It combines the well-

WWW 2013 Companion, May 13–17, 2013, Rio de Janeiro, Brazil.
ACM 978-1-4503-2038-2/13/05.

known *Representational State Transfer* [5] with the *Data Transfer Object*-pattern [7, 20] to provide a seamless and unified interface, using a simplified data model, which fits on the respective client and its use case.

The rest of the paper is structured as follows. Section 1.1 summarises related work, followed by the contribution of this work in Section 1.2. The general approach is introduced in Section 2 and its implementation in Section 3, both using some exemplary issues from the case study, which is entirely explained in Section 4. Finally, Section 5 concludes this paper by discussing pros and cons as well as introducing further work.

1.1 Related Work

Representational State Transfer (REST) means a set of architectural principles, which enable the design of scalable, maintainable, and evolvable web services. It was introduced in 2000 by Roy Fielding [5, 6]. REST is one of the two major architectural styles for designing and implementing web services. Pautasso et al. [17] compare the resource-oriented one to the RPC-based one and its most common implementation—the SOAP standard—and look into them regarding different conceptual and technological criteria. In a similar way, Pautasso and Wilde [15] compare RESTful HTTP, RPC over HTTP and the WS-* technology regarding the degree of loose respectively tight coupling.

In the meantime, due to its simplicity, lightweight, and interoperability, REST has gained more and more attention for the creation of interfaces for various use cases. It is used for web APIs, but also for enterprise application integration and service-oriented architectures [4, 16, 23]. The concept for creating simplified, web-based interfaces presented in this paper is mainly based on the REST principles.

Due to the relevance in implementing interfaces and APIs mentioned before, there are some promising approaches dealing with different ways of designing, modelling, and creating as well as mapping existing applications into RESTful interfaces. Schreier [18] introduces a meta model for designing REST applications, which supports developers in model-driven development. Szymanski et al. [21] present a modelling process, which enables the mapping of a service-oriented legacy application into a RESTful interface, and handles the challenge of creating an equivalent resource model. Yet, a manual proceeding is proposed to identify resource candidates, refine the received model, consolidate activities, and validate the result. Liu et al. [13] propose a process for reengineering legacy and service-oriented systems as well. They consider entities from the data model, relationships between entities and business processes, and combine that into the resource model of the RESTful interface. Strauch et al. [19] present an iterative process model, which supports developers to redesign a service-oriented interface into a RESTful one. All these approaches provide—technical or conceptual—tools to transform service-oriented systems and interfaces into really RESTful ones. Laitkorpi et al. [11, 12] introduce a tool-supported process to abstract application interfaces to REST-like services. Different from our approach, this one enables developers to create a new service from the scratch and generate a one-to-one RESTful interface for that. Engelke and Fitzgerald [3] report on a case study aiming to encapsulate an existing reliable and proven legacy application by a RESTful interface specifically developed for this purpose. In contrast to our work, there is no general approach, and the RESTful interface only forwards payload in a transparent way, against which our approach does some data transformation for providing a simplified data model to respective clients. All these approaches have in common, that only one application—already existing or developed from scratch—is encapsulated by a newly designed RESTful interface.

There are several approaches to measure the "RESTfulness" of APIs, Fowler for example proposes the *Richardson Maturity Model* [8]. In our case, we assume that developers are familiar with RESTful design and rely on them designing a nice RESTful interface. Therefore, we want to provide a tool which facilitates the implementation of this design.

The *Data Transfer Object* pattern is one of the Core J2EE Patterns introduced by Sun Microsystems [20] as well as described by Fowler [7]. It propagates the encapsulating of several objects into one Data Transfer Object (DTO) to reduce data overhead of remote calls. This pattern is extended by the *Custom Data Transfer Object* pattern [14], which demands to encapsulate only attributes required by the specific client of the remote call. This is the second work, our approach is based on. We use this pattern to achieve the simplification of an existing data model and thereby the simplification of the involved interfaces.

1.2 Contribution of this Paper

This paper contributes to the area of enterprise application integration. A concept is presented, which enables developers to adapt—personally developed or bought—applications or even a composition of several applications, which are based on heterogeneous technologies. Moreover, they are made available as web services with a uniform and simplified RESTful interface (wherein "'simplified"' does not mean a limited maturity of RESTfulness as mentioned by Fowler [8], but concerning a reduction of the interface extent and the complexity of its data model). The generator-based implementation enables developers to create and evolve this interface in a rapid and model-driven way. Beyond this, the modular tool implementation enables the integration of additional features like version control or conflict detection for interfaces. In particular, the following contributions are presented:

(i) An additional layer is introduced, which implements the interfaces of existing services and enables the provision of a unified and seamless RESTful interface.
(ii) Developers are enabled to simplify this interface by determining extent, structure, and data model. Since different interface consumers have different requirements for the respective interface, the developer is able to create a separate so-called *use case-specific interface* for each of these interface consumers.
(iii) To facilitate the design of this new interface as well as the coupling to services of the existing system, a domain-specific language is developed. This enables a quick and easy integration of new services.
(iv) This approach is realised as an Eclipse-based tool, consisting of an editor supplemented by a code generator. It produces productive JAX-RS code [10], which is runnable in a standalone environment, but above all scales for use with enterprise applications. Due to its modular design, the tool is extendable with several features.

2. APPROACH

The presented approach is a framework for the model-driven development of simplified and use case-specific RESTful interfaces, building on already existing business logic. The overall architecture is shown in Figure 1: starting from an existing system (corresponding to *Application layer*) with a global data model (cf. *data model* at *Application layer*), several services and databases, the developer is able to create use case-specific interfaces (*UCSI*s). Each of them has its own, simplified data model in the form of *DTO*s.

To illustrate the approach, the following scenario is assumed: a developer shall create a new mobility service, for instance for e-bike sharing. Most of the functional requirements can be fulfilled by the composition of existing, perhaps bought subsystems (or COTS), which offer functionalities for customer administration, bike management and reservation, billing, etc. and thereby constitute the backend of the system. Since this system is adjusted for a specific domain (in this case for the e-bike sharing domain), hereinafter it will also be referred to as *domain-specific application*. In Figure 1, this domain-specific application is represented by the *Application layer*.

Besides others, one use case is that the system shall enable the customer to find free e-bikes as well as make and manage reservations on the way. Therefore, the developer shall create a client, e. g. a smart phone app, which meets these requirements. In principle, the subsystems offer interfaces for that client, but for the following issues (I1 - I3), the creation of a use case-specific interface is necessary.

(I1) *Heterogeneous and legacy interfaces:* Existing interfaces are not only syntactically and technically heterogeneous, but also might not support web-based access. Therefore, it is hard to make them attainable for clients via the Internet.

(I2) *Complex interface structure:* Subsystems might offer a larger number of and more complex methods than needed for this use case.

(I3) *Complex data model:* The underlying data model might be more complex than needed by the client, which has adversely effects at runtime on quality attributes like performance, and responsiveness as well as additional implementation effort on client side.

Each of these issues can be considered separately. Thereby, we are able to derive the following assumptions (A1 - A3). Afterwards, one requirement is formulated for each assumption, which represents the base for the general approach.

(A1) The introduction of an additional web service layer provides a unified and seamless interface and enables various clients to implement against it and access it.

(A2) An appropriate selection of needed services and methods along with a hierarchical grouping instead of offering a loose set intuitively makes the interface better manageable and capable of being integrated, as we will show later.

(A3) A simplified and use case-specific data model designed for a specific client reduces implementation effort on client side in general as well as transferred data at runtime.

2.1 An Additional Web Service Adaption Layer

In a first step, an additional layer is introduced: it implements the interfaces of its required target services and provides a web service-based interface for any clients (see *Presentation layer* in Figure 1). Operations that are performed on data transferred between both layers, will be specified in the next sections.

At the moment, there are two common ways to implement web services: the SOAP standard and RESTful HTTP. The latter is not a standard, but a set of design principles. Both sides have their pros and cons: as Pautasso et al. [15, 17] show, SOAP is more suitable for complex and static services with higher quality of service requirements and with a longer lifespan like for instance business processes, whereas REST enables really loose coupling and is more appropriate for ad hoc integration via the Internet. For REST, concrete implementations can draw from a couple of well-known standards (HTTP, XML, URI), for which libraries are available for most platforms. Compared to SOAP, thereby the footprint regarding traffic and system resources is also reduced, which favours the deployment on remote devices like smart phones or embedded devices. Since RESTful services shall be hypertext-driven, no contract has to be declared. This loose coupling enables the further developing of interfaces without the need to adjust client implementations. Beyond this, caching, clustering and load balancing is very well supported by REST. Since we want to create lightweight and easily evolvable interfaces for remote clients with limited resources, but without supporting complex constructs like business processes or transactions, we choose REST for our purpose.

Derived from issue (I1), we define the first requirement:

(R1) An adapter shall be added in the form of an additional layer and it shall offer its interfaces as RESTful web services.

The further implementation of the REST principles will be explained in the following two sections.

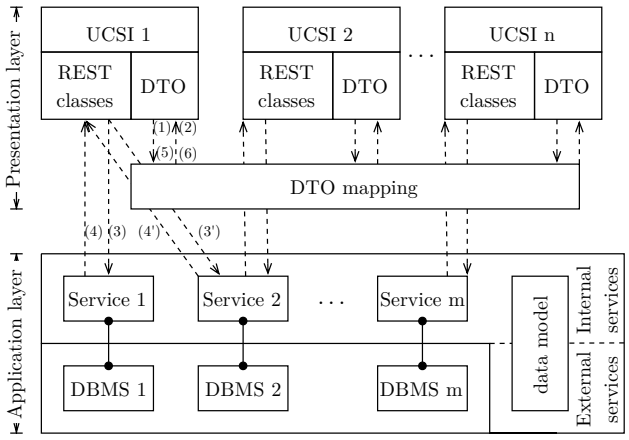

Figure 1: Overall architecture: existing, domain-specific application with additional use case-specific interfaces.

2.2 Use Case-specific Interface

The above introduced additional layer is now specified in more detail by designing use case-specific and simplified interfaces. As mentioned before, the composed subsystems usually offer a higher number of services and methods than

needed for a specific use case. Moreover, they are grouped in the existing subsystems, but there is no seamless interface for all subsystems (cf. issue (I2)). Regarding this, one more requirement is formulated:

(R2) The developer shall be able to explicitly select, which services and methods are mapped by the adapter, and at which place in the adapter structure they are accessible.

One of the key features of REST is the resource-oriented handling of any information. Fielding [6] calls a resource a "conceptual mapping to a set of entities", which can be a document, a handler for a non-virtual, i. e., physical object (like an e-bike), or a temporal service (e. g. the reservation plan of an e-bike). Beyond this, the possibility to reference a concept instead of a singular representation permits the changing of a representation without changing all references to that. Moreover, possibly complex and heterogeneous interfaces can be hidden from the client. Thereby, a uniform and seamless interface is created. An example is shown in Figure 2. On the left side is the "conventional", service-oriented interface, and building on that, on the right side the coupled, resource-oriented one. The service-oriented in-

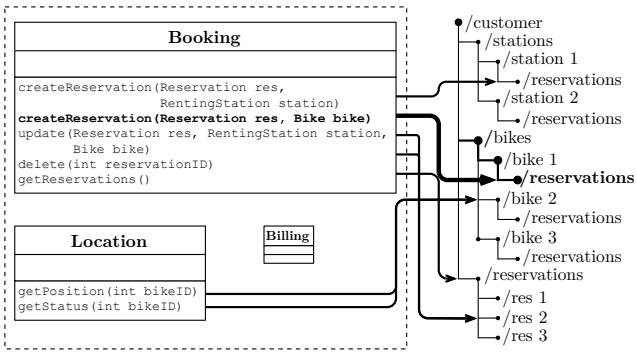

Figure 2: "Conventional" service-oriented interface and RESTful interface building on that.

terface offers an interface contract with several methods, including (i) one for the creation of reservations at a specific station, (ii) one for updating, and (iii) one for deleting existing reservations. In this case, the client has to implement each of the three methods.

However, REST requires a so-called "uniform interface", which means a limited number of operations with a well-defined semantics. In this case, the client only has to use one of the HTTP commands on the respective resource: (i) a `POST` request on the resource `/stations/{id}/reservations`, or (ii) `PUT` and (iii) `DELETE`, respectively, on `/reservations/{id}`, yielding a reduction of implementation effort.

In case another method is added to the backend, e. g. `createReservation(Reservation res, Bike bike)` to enable the reservation of a specific bike, the service-oriented interface contract thereby will be broken, and the client necessarily will have to implement the new contract. However, in the case of RESTful interfaces, the backend will offer a new resource like `/bikes/{id}/reservations` (exemplary highlighted in Figure 2). The client can ignore the new functionality, or use it through the well-known `POST` command.

2.3 Use Case-specific Data Model

So far, the existing interface was changed from a non-web service to a web service one, and it was made RESTful. The advantages are obvious, but not new: without the fact that we later want to develop a DSL coming along with a code generator, it is only an ordinary RESTful interface.

But as determined in issue (I3), the domain-specific data model might be more complex than needed by the client. For that, building on the RESTful interface and employing the *Custom Data Transfer Object* pattern, now the added value is created:

(R3) The developer shall be able to design a use case-specific data model: based on the domain-specific data model, s/he shall be able to choose entities and attributes, which will be encapsulated into a single DTO respective into the resource representation.

By choosing only needed domain-specific attributes, s/he is able to design a use case-specific data model, which is a subset of the domain-specific data model and specifically fits on the certain client.

Commonly, backend systems are developed for a specific domain, but they might be used by users with different views. For example, an e-bike reservation and booking system should allow customers to make reservations via a smart phone app, but also provide resource management and billing functionalities for the service operator as well as a location functionality. Every view has its own sub-requirements regarding supported data. Therefore, the domain-specific data model represents a superset of all sub-requirements. Advantages are maintainability and extensibility, but at the cost of a higher complexity of the transferred data: this causes not only higher data traffic at runtime, but also more implementation effort on client side.

As mentioned before, the REST principles propagate a resource-oriented handling of all information. We take advantage of this by making all entities from the domain-specific data model available via resources. The client shall be able to retrieve and also to manipulate their representations. Again, we use the example introduced in Figure 2. The corresponding data model is shown in Figure 3: it consists of the basic reservation entity, the assigned e-bike, and the station where the reservation begins. This data model also supports other use cases like bike management, but is overdone for a customer client that only shall display a list of pending reservations. In the last section, we mapped the `getReservations()` procedure into the `/reservations`-resource, on which a `GET`-request has to be applied. In a next step, we will define the representation, which the client will receive. It is based on the existing data model, but is customised for the specific use case. This is achieved by employing the *Data Transfer Object* (DTO) pattern, which is known from the J2EE world and was originally introduced for remote calls in EJB context. By combining multiple objects into one DTO, a reduction regarding data overhead is reached. As in this case not only the quantity, but also the data model complexity is crucial, the *Data Transfer Object* is extended to the *Custom Data Transfer Object* pattern: only attributes are added to the DTO, which are required by the client.

As we see in the example shown in Figure 3, through a selection of needed attributes, we can reduce the number of required DTOs to only two (starting from four entities).

Certainly, this means an alignment on one specific use case, but for this use case, the complexity of the interface is decreased significantly.

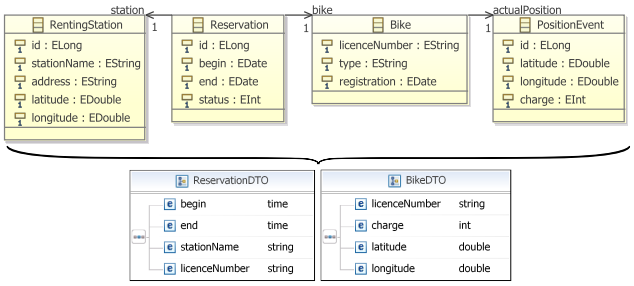

Figure 3: Reservation: domain-specific data model and client-specific DTOs.

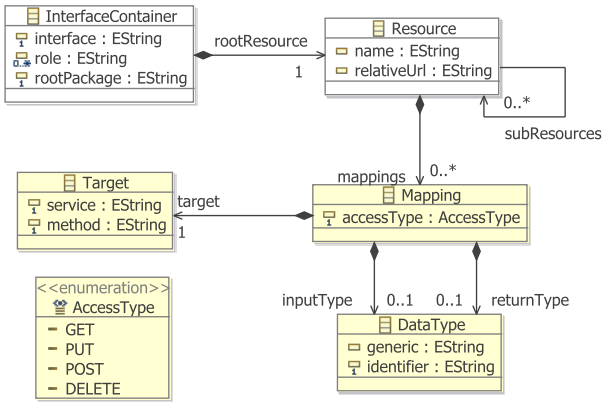

Figure 4: Meta model of the DSL.

2.4 A DSL for Interface Specification

After having introduced our approach, in principle we could implement our interface in a general-purpose language, for example Java. But since we do not want to concentrate on technical details, a DSL-based approach combined with a code generator is more suitable, which enables modelling on a higher-level abstraction with less implementation effort. As mentioned before, the REST principles enable a separation of resources and their representations, hence two separated steps are carried out:

1. Designing the representations (respective DTOs), which consists of creating a set of entities. This is done by creating an XML schema (cf. Section 3.2).
2. The more complex part is designing the interface structure. Since not only the resource hierarchy has to be defined, but also the coupling with the respective target methods has to be determined, a domain-specific language was created. The meta model is shown in Figure 4.

In a first step, an *InterfaceContainer* is defined, which contains some general information of the interface like its name, a Java package where output classes are generated to, and a list of user roles, which are allowed to access the interface (to reach a more fine-grained role concept, each resource and even each mapping can have its own roles in the actual implementation, but this is omitted for simplicity). The actual interface structure is built by one *Resource* referenced as *rootResource*, and a set of subordinate *Resources*.

In the second step, one or more so-called *Mappings* are defined for a *Resource*. For each *Mapping*, an *AccessType* is defined as well as the target, which shall be coupled to that. Therefore, a *Target* with attributes *service* and *method* is defined.

Finally, the needed DTOs are defined. Therefore, every *Method* gets an *inputType* and a *returnType*, which are *DataTypes*. A *DataType* defines the corresponding class by its *dataType* attribute. Furthermore, the optional attribute *generic* enables to prepend a generic type, for example to specify a list.

The DSL is demonstrated exemplary by the interface specification for the customer client use case in Figure 6.

3. IMPLEMENTATION

After having introduced the basic concepts, the technologies employed at runtime as well as details of tool implementation are presented in the following. This section is divided into two parts: in the first part, we show the interfaces' runtime implementation; in the second part, the realisation of the DSL and its belonging code generator as well as the way to design the DTOs is introduced.

3.1 Runtime

Since the interfaces shall be operated in environments of different sizes, but particularly in combination with large scale enterprise information systems, we build on the *Java Platform, Enterprise Edition*. Java EE allows a faster and less expensive development of highly available, secure, reliable, and scalable applications. Developers benefit from being able to concentrate on the functional development of the application, while infrastructure issues are taken by the runtime environment.

In our case, mainly the provision of RESTful web services is crucial, but also the coupling with the target services via different technologies.

For providing RESTful web services, Java EE includes the *Java API for RESTful Web Services* (JAX-RS) [10]. JAX-RS enables the developer to create RESTful web services by annotating Java classes with some specific annotations. The application can be operated in different environments like a standard Java SE runtime or a servlet container.

JAX-RS integrates *Java Architecture for XML Binding* (JAXB) [9] as a mechanism for runtime un-/marshalling Java objects, for example from/into XML or JSON. Unfortunately, there is only limited support for hypermedia as the engine of state in JAX-RS (even the specification leads admit that [22]). Therefore, we integrate the proprietary atom implementation provided by RESTEasy [1], which is our JAX-RS implementation.

The integration of the target services makes two aspects necessary: (i) implementing the protocol of the respective target service and (ii) mapping domain-specific objects into DTOs and vice versa. Since this solution shall be extensible for as many service protocols as possible, Java EE is a good choice. It offers a set of different adapters to integrate various components. By default, components can be integrated via transactions (JTA), Remote Method Invocation (RMI-

IIOP), CORBA (Java IDL), or messages (JMS). However, the Java EE Connector Architecture (JCA) is especially interesting. It enables the development of Java EE conform adapters for any service, even legacy ones.

To realise the DTO mapping, an object factory is used. This factory uses runtime-reflection, just as proposed by Sun Microsystems in the *Transfer Object Pattern*, to instantiate the target objects, and to copy the attributes. Identifiers of source and target objects are determined by the developer in our DSL, and are generated as Strings into the code of the JAX-RS classes.

The common way of mapping DTOs and calling a target service is shown in Figure 1: after receiving a DTO from a client, the REST class sends it to the DTO mapping mechanism (1) and gets a domain model-specific object returned (2). Taking this object, it calls its target service (3) and gets a response back (4). This domain-specific object is translated back into a DTO ((5) and (6)). Note, each UCSI might use several services, i. e., we observe a 1:m relation. The REST classes of UCSI 1, for example, utilise *Service 1* and *Service 2* indicated with (3') and (4'), respectively.

3.2 Tool Support

For the implementation of the DSL, the Xtext framework [2] is used. Xtext is a mature Eclipse project, which enables the developer to create an own DSL and also provides an Eclipse-based development environment for that DSL. The syntax is modelled equivalently to the DSL meta model shown in Figure 4, a code example is shown in Figure 6. The belonging code generator is written in the integrated Xtend language [2].

The generator creates the RAX-RS-annotated classes out of the DSL code; based on the target services defined via the *service* and *method* attributes, it retrieves the signatures of the called methods from a data source (for example a WSDL file or a database); this information is used to generate the method calls as well as calling the DTO mapping, which performs the object translation. Retrieving method signatures, generating method calls as well as the JAX-RS code are outsourced into their own Eclipse plugins. Thereby, using another data source or calling target services with a different technology (even several technologies simultaneously) can be simply done by exchanging the respective plugins.

Figure 7 depicts a code snippet, which corresponds to lines 12 to 20 of the DSL example from Figure 6. The `@Path` annotation at line 1 makes the *BikeService* and its methods accessible via the URL `/customer/bikes`, and the `@RolesAllowed` annotation declares user roles allowed to access this resource. Through the `@GET` annotation at line 7, respective requests are mapped on the `getReservations()` method, the `@Produces` annotation at line 8 determines that the returned objects shall be marshalled to XML, containing atom links. Furthermore, `@AddLinks` at line 5 denotes that atom links shall be injected into the returned entities, and `@LinkResource` at line 6 defines that all `BikeDTO` entities shall have an atom link on this URI. Lines 11 to 14 implement the target service call through JNDI, since currently services are implemented as EJBs. The target service is called in line 15, from line 16 to line 21, the transformation of the domain-specific list of *Bike*s into a *BikeDTO* list is performed. Everything else like authentication, un-/marshalling of objects, handling of HTTP connections, and so on is done by the servlet container.

As mentioned before, the DTOs respective resource representations are designed as XML schema; Eclipse provides the XML Schema Definition SDK, which includes a graphical editor and also a schema-to-class-compiler to create ready-to-use JAXB-annotated classes from the schema.

4. CASE STUDY

This case study deals with a corporate e-bike sharing system. Since nowadays IT infrastructures get more and more important as a backbone for mobility services of this kind, we integrate this e-bike sharing as case study for the presented approach.

A company's branches are spread over a metropolitan area and its employees often have to move between these branches. Since going by public transports is very time consuming and parking situation is often bad in this area, the company wants to introduce an e-bike sharing system to provide a clean, healthy, and fast transport alternative for its employees. They shall be able to locate a free bike via a smart phone. If an appointment is fixed a long time before, a reservation for a time slot shall be able, too. Since there is only a closed group of users, no personalised bike access is required like for B2C sharing systems. Instead, bikes have locks with a combination known to all employees.

The case study considers the following components and users and is depicted in Figure 5.

(i) Employees use smart phones to locate and reserve available e-bikes.
(ii) The backend infrastructure orchestrates between the various devices and provides functionalities to them.
(iii) Client devices (e. g. smart phones) demand functionality provided by the backend.
(iv) Tracking devices, which are integrated into every e-bike, enable to retrieve their location, and to detect the charge level.

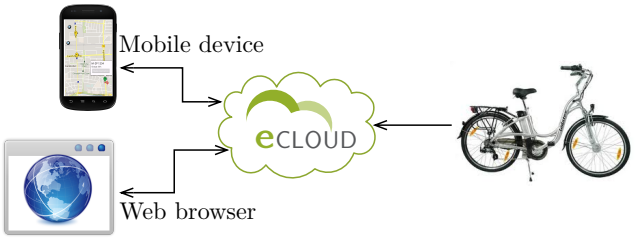

Figure 5: Overall topology of the case study.

The backend is the backbone of the whole system, which offers the central business logic for management and reservation of e-bikes, and for integrating tracking devices to locate them. It offers use case-specific interfaces for tracking devices and customer smart phones.

The application is implemented with a 3-tier architecture: the domain-specific part consists of business layer and persistence (see *Application layer* in Figure 1), building on that, several use case-specific interfaces are integrated, which represent the *Presentation layer*. The domain-specific application is built on the Java EE-specification, therefore, we use a JBoss Application Server 7.1.1, the interfaces are operated at the integrated Tomcat 6-fork.

4.1 Domain-specific Application

In technical terms, the domain-specific application is completely based on Java EE. Business logic as well as data access objects are deployed as Enterprise Java Beans.

The business logic offers services for reservation management and for location of bikes. The reservation service enables clients to create, modify, and delete reservations for a certain time slot. The location service receives position events from the tracking devices, and enables user clients to request the position of a certain bike.

4.2 Use Case-specific Interfaces

Two use case-specific interfaces are provided: (i) one for tracking devices, which offers only one simple resource accepting a `POST` request with a PositionEventDTO, and (ii) one regarding customer functionalities, e. g. for a smart phone app. This structure is shown in Figure 6 as DSL snippet. It offers functionalities to locate free bikes, make reservations at a specific station or for a specific bike, and to retrieve, update, and delete existing reservations. As mentioned before, the interfaces use the JAX-RS-specification. JBoss AS integrates the RESTEasy framework [1] as JAX-RS implementation. Additionally, two DTOs are defined (see Figure 3), one to represent e-bikes (BikeDTO) and one for reservations (ReservationDTO).

5. CONCLUSION AND FUTURE WORK

In this paper we presented a way to adapt existing, maybe legacy services and to offer them via a RESTful web-based interface. This is achieved by employing an interface description DSL supplemented by a code generator. The produced productive Java code scales for use with enterprise applications due to the employed technology. A case study showed the feasibility of the presented approach by dint of an e-bike sharing facility. Advantages just as described at the beginning of this paper were verified during the case study:

(i) Starting with only a syntactic description of the existing services, a developer is able to create a complete web-based interface only with a few lines of code.

(ii) Up to a certain point, the presented tooling enables a developer to concentrate on the essentials rather than having to care about technical issues like platform specific peculiarities.

(iii) The generator-based approach enables a fast interface evolution: changing a parameter like a resource URI produces deployable code at the same moment.

But the evaluation also brought some weaknesses to the light:

(i) The presented approach does not support developers to comply with REST design constraints (as Szymanski et al. [21] or Strauch et al. [19] do, for example), so it is up to the developer to take care of REST principles and not just to model ordinary HTTP-based APIs.

(ii) The mechanism for mapping between use case- and domain-specific data models is based on runtime reflection, attributes to copy are compared by their names. Therefore, the developer has to make sure that attribute pairs have the same name and at least compatible data types.

Practical knowledge gained from the case study convinced us that we are on a good way and can make a real contribution to the area of enterprise application integration. To improve our approach and the belonging tools, we plan some further work: (i) *Implementation of further adapters*: with the current implementation, only EJBs can be integrated as target services. Therefore, more adapter technologies shall be implemented. For that, JCA is on top of our agenda. (ii) *Design support*: as mentioned before, this approach does not support in meeting the REST design principles. Therefore,

```
1  InterfaceContainer {
     interface customerInterface
3    role customer
     rootPackage de.zielpuls.ebike.interfaces.customer
5    rootResource RootResource
   }
7  Resource RootResource{
     relativeUrl /customer
9    subResources (LocationResource, StationBookingResource,
              BikeBookingResource, ReservationResource)
11 }
   Resource LocationResource{
13   relativeUrl /bikes
     Mapping{
15     accessType GET
       service de.zielpuls.ebike.logic.LocationLogic
17     method getBikes
       inputType
19     returnType generic List dataType BikeDTO
   }}
21 Resource StationBookingResource{
     relativeUrl /stations/{id}/reservations
23   Mapping{
     accessType POST
25     service de.zielpuls.ebike.logic.BookingLogic
       method createReservationForStation
27     inputType dataType ReservationDTO
       returnType
29 }}
   Resource BikeBookingResource{
31   relativeUrl /bikes/{id}/reservations
     Mapping{
33     accessType POST
       service de.zielpuls.ebike.logic.BookingLogic
35     method createReservationForBike
       inputType dataType ReservationDTO
37     returnType
   }}
39 Resource ReservationResource{
     relativeUrl /reservations
41   subResources (SingleReservationResource)
     Mapping{
43     accessType GET
       service de.zielpuls.ebike.logic.BookingLogic
45     method getReservations
       inputType
47     returnType generic List dataType ReservationDTO
   }}
49 Resource SingleReservationResource{
     relativeUrl /{id}
51   Mapping{
       accessType  PUT
53     service de.zielpuls.ebike.logic.BookingLogic
       method updateReservation
55     inputType dataType ReservationDTO
       returnType
57   },
     Mapping{
59     accessType  DELETE
       service de.zielpuls.ebike.logic.BookingLogic
61     method deleteReservation
       inputType
63     returnType
   }}
```

Figure 6: Syntax example: client interface for e-bike locating and reservation handling.

```
@Path("/customer/bikes")
@RolesAllowed("customer")
public class LocationService {

  @AddLinks
  @LinkResource(value = BikeDTO.class)
  @GET
  @Produces("application/atom+xml")
  public Response getBikes() {
    try {
      Context jndiContext = new InitialContext();
      LocationLogicLocal locationService = (LocationLogicLocal) jndiContext.lookup(
           "java:app/de.zielpuls.ebike.logic/LocationLogic!de.zielpuls.ebike.logic.LocationLogic");
      jndiContext.close();
      List<Bike> domainObjectList = loactionService.getBikes();
      DtoTransformationLogic transformation = new DtoTransformationLogic();
      List<BikeDTO> returnDtoList = new LinkedList<BikeDTO>();
      for (Bike b : domainObjectList) {
        returnDtoList.add((BikeDTO)transformation.createTransferObject(b,"de.zielpuls.ebike.dto.customer.BikeDTO",
                                                                       "de.zielpuls.ebike.datamodel.Bike"));
      }
    return Response.status(200).entity(returnDtoList).build();
    } catch(Exception e) {
      //
} } }
```

Figure 7: Code example: generated client interface for location of e-bikes.

we will think about combining our approach with one of the support processes presented in Section 1.1.

6. REFERENCES

[1] RESTEasy framework. http://www.jboss.org/resteasy/.

[2] Xtext/Xtend project. http://www.eclipse.org/{Xtext,xtend}.

[3] C. Engelke and C. Fitzgerald. Replacing legacy web services with RESTful services. In *WS-REST*, pages 27–30, 2010.

[4] T. Erl, B. Carlyle, C. Pautasso, and R. Balasubramanian. *SOA with REST - Principles, Patterns and Constraints for Building Enterprise Solutions with REST*. The Prentice Hall service technology series. Pearson Education, 2013.

[5] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. Phd thesis, University of California, Irvine, 2000.

[6] R. T. Fielding and R. N. Taylor. Principled design of the modern Web architecture. In *ICSE*, 2000.

[7] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[8] M. Fowler. Richardson Maturity Model. http://martinfowler.com/articles/richardsonMaturityModel.html, 2010. [Online; accessed February 16th 2013].

[9] Java Community Process. JSR 222: Java Architecture for XML Binding (JAXB) 2.0, December 2009. Java Specification Request.

[10] Java Community Process. JSR 311: JAX-RS: The Java API for RESTful Web Services, November 2009. Java Specification Request.

[11] M. Laitkorpi, J. Koskinen, and T. Systa. A UML-based Approach for Abstracting Application Interfaces to REST-like Services. In *WCRE*, 2006.

[12] M. Laitkorpi, P. Selonen, and T. Systa. Towards a Model-Driven Process for Designing ReSTful Web Services. In *ICWS*, 2009.

[13] Y. Liu, Q. Wang, M. Zhuang, and Y. Zhu. Reengineering Legacy Systems with RESTful Web Service. In *COMPSAC*, 2008.

[14] F. Marinescu. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 2002.

[15] C. Pautasso and E. Wilde. Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design. In *WWW*, 2009.

[16] C. Pautasso and E. Wilde. RESTful web services: principles, patterns, emerging technologies. In *WWW*, 2010.

[17] C. Pautasso, O. Zimmermann, and F. Leymann. RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. In *WWW*, 2008.

[18] S. Schreier. Modeling RESTful applications. In *WS-REST*, 2011.

[19] J. Strauch and S. Schreier. RESTify: from RPCs to RESTful HTTP design. In *WS-REST*, 2012.

[20] Sun Microsystems, Inc. Core J2EE Patterns - Transfer Object. http://www.oracle.com/technetwork/java/transferobject-139757.html, 2002. [Online; accessed at September 18th 2012].

[21] C. Szymanski and S. Schreier. Case study: Extracting a resource model from an object-oriented legacy application. In *WS-REST*, 2012.

[22] S. Tilkov. JSR 311 Final: Java API for RESTful Web Services. http://www.infoq.com/news/2008/09/jsr311-approved. [Online; accessed at February 18th 2013].

[23] S. Tilkov. REST und HTTP. *Einsatz der Architektur des Web für Integrationsszenarien, dpunkt. verlag*, 2009.