FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Web Service Composition in Drupal

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Klaus Purer
Matrikelnummer 0426223

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Prof. Dr. A Min Tjoa
Mitwirkung: Univ.-Ass. Dr. Amin Anjomshoaa

Wien, 11.5.2011

(Unterschrift Verfasser/in)

(Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11.5.2011

Klaus Purer

Abstract

Building web applications has become a complex task and often requires interaction with other web applications, such as web services. Drupal is a free and open source content management system and framework that provides a rich platform for rapid web development. The modular and extensible nature of Drupal allows developers to customize and embrace the core functionality and to create new features. This thesis is about investigating and implementing a web service client module for Drupal that is able to consume classical WS* web services as well as RESTful web services. We will present a web service abstraction model which supports different web service types in order to facilitate integration of web service data into workflows in Drupal. Those workflows are built with the help of a rule engine module ("Rules") that offers the creation of event-condition-action rules. We will discuss a solution that provides a web service operation as Rules action and that achieves web service composition by invoking multiple web services in a Rules workflow. This is important for web applications that need to communicate with several external web services and require the orchestration of the data flows between them. Additionally a user interface has been built where web services can be described and used on Drupal administration pages, which means that no programming effort is needed to access web services. Other features such as automatic parsing of WSDL files or sharing of web service descriptions between different Drupal sites are also realized. The implementation has been evaluated and tested on the basis of an automatic translation use-case that is comprised of a workflow with multiple web service invocations.

Zusammenfassung

Das Erstellen von Webapplikationen ist mittlerweile eine komplexe Aufgabe und erfordert oftmals die Integration mit anderen Webapplikationen, im speziellen mit Webservices. Drupal ist ein freies Open Source Content Management System und Framework, das eine umfassende Plattform für schnelle Web-Entwicklung bereitstellt. Die modulare und erweiterbare Charakteristik von Drupal erlaubt EntwicklerInnen die Kernfunktionalität anzupassen und auszunutzen, um neue Funktionalitäten zu erstellen. Diese Diplomarbeit beschäftigt sich mit der Erforschung und Implementierung eines Webservice Client Moduls für Drupal, welches in der Lage ist, sowohl klassische WS* Webservices als auch RESTful Webservices zu konsumieren. Wir werden ein Abstraktionsmodell für Webservices präsentieren, das verschiedene Webservice-Typen unterstützt und welches die Integration von Webservice-Daten in Drupal Workflows ermöglicht. Diese Workflows werden mit Hilfe eines regelbasierten Moduls ("Rules") konstruiert, mit dem Event-Condition-Action Regeln erstellt werden können. Wir werden eine Lösung diskutieren, die eine Webservice-Operation als Rules Action zur Verfügung stellt und die damit die Komposition von Webservices erreicht, indem mehrere Webservices in einem Rules Workflow aufgerufen werden. Das ist wichtig für Webapplikationen, die mit vielen externen Webservices kommunizieren müssen und den Datenfluss zwischen diesen orchestrieren müssen. Zusätzlich wurde eine Benutzeroberfläche implementiert, womit Webservices auf Drupal Adminstrationsseiten beschrieben und benutzt werden können. Dadurch werden keine Programmierkenntnisse benötigt, wenn Webservices angesprochen werden sollen. Die Realisierung beinhaltet auch andere Funktionalitäten wie das automatische Auslesen von WSDL-Dateien oder die Weitergabe von Webservice Beschreibungen an andere Drupal-Installationen. Die Implementierung wurde mit einem Anwendungsfall zur automatischen Übersetzung evaluiert und getestet, der aus einem Workflow mit mehreren Webservice Aufrufen besteht.

Acknowledgements

I would like to dedicate this thesis to the Drupal community who inspired me in many ways and showed me the benefits of sharing code, ideas and support.

I wish to acknowledge Wolfgang “fago” Ziegler for his comprehensive feedback when developing the project of this thesis. Kudos go out to Klaus Furtmueller that came up with the initial idea for the thesis.

I thank Dr. Amin Anjomshoaa for the supervision of this thesis and Prof. A Min Tjoa for the opportunity of writing the thesis at the Institute of Software Technology & Interactive Systems.

To the Free Software / Open Source communities, I extend my gratitude for making all of my work worthwhile – it’s just so much more fun if there is someone out there who can put the results into productive use.

Contents

Chapter 1

Introduction

If you can, help others; if you cannot do that, at least do not harm them.
– Dalai Lama

1.1 Motivation and background

Drupal

Drupal[1] is a popular Open Source Content Management System (CMS) that allows simple creation and management of web sites and web applications. It was introduced in 2001 with the idea of storing web content in a database instead of putting it into HTML files. Historically the web was a collection of documents linked together statically [Jaz07]. But now administrators and web masters were able to add and edit content directly on the site – instead of uploading files with a FTP account to the hosting server, they authenticated on site and performed changes in an administration interface.

Nowadays Drupal has evolved: it is not only a CMS anymore, but has matured into a web framework as well that provides many APIs for developers to easily integrate their customizations and features. There are over 6,000 contributed modules[2] on drupal.org that extend or modify the Drupal core system. All of them are distributed under the terms of the GNU General Public License (like Drupal itself) and are part of the reason why Drupal is so successful. The dynamics of Free and Open Source Software and the module ecosystem strongly influence innovation and broad reach among the Drupal community.

[1] Drupal: http://drupal.org
[2] Drupal modules: http://drupal.org/project/modules

Since the web grew and the term Web 2.0[3] came up, Drupal was redefined as a provider for social network platforms. Content and users were already the primary focus in Drupal, so it was a reasonable step to let arbitrary users manage their content which previously was done by site administrators only. But building sites and opening them up for users is not enough – integration with other social services like Facebook, Twitter or other web services is most often a requirement. As web sites get bigger and more complex, they also need to address more and more workflows between users, administrators or other data providers and consumers (services, external sources, business processes etc.).

Web services

Web services allow humans or automated agents to interact with a system via the web. They are described by a well known interface, are self-contained and expose a certain functionality of the system to the outside world. They offer operations to send and retrieve data and it is possible to compose them in a workflow. The term web services was often associated with the WS* stack, a set of standards for description, lookup and communication regarding web services mostly based on exchanging SOAP messages [DS05]. This formally very strict approach did not satisfy simple needs for some use cases and lead to the rise of RESTful services in recent years [FT00]. They offer an interface that is simple but not formally described and rely on the architecture of HTTP and are therefore more resource oriented than operation oriented.

Both types of web services are now in wide use and are accepted as one major concept of the web. Modern web sites are forced to provide services themselves to allow third parties easy and fast consumption of the sites' data. On the other hand complex sites often need to connect to other sites to import data or aggregate content. In most cases there is a considerable amount of development and programming effort needed to integrate the machine readable web service interfaces and to map internal data structures to the corresponding service parameters or results.

Drupal offers the possibility to provide various kinds of services, like built-in support for RSS feeds or more advanced components like the Services module[4]. The latter allows the configuration of SOAP, REST and other service types to expose Drupal internals via known interfaces. There are also approaches for the other way around (consuming services in Drupal), but they all are tied to specific services that need to be integrated into the system and the data workflows.

[3] Web 2.0 is a fuzzy buzz word that mostly describes interactive and collaborative behavior of web users that create and update web content. Tim O'Reilly has the most widely accepted description of the term [O'R05].

[4] Services module: `http://drupal.org/project/services`

Workflows and Rules

Web applications fulfill more and more different tasks at a time and often need to organize workflows, business processes and automatic data management. An example would be the use case of buying an item in a web shop, where several follow-up tasks need to performed. The customer needs to be billed, the products need to be scheduled for delivery, the remaining amount of products needs to be updated, external software services must be notified or invoked etc. Those tasks need to be implemented, coordinated and updated on a regular basis. They comprise one or more workflows which need to be re-configured or fine-tuned periodically.

CMS like Drupal aim to make many configurations available to site builders and administrators, so that no extra programming effort is needed when customizing the system. This applies to workflows as well and there is the Rules module[5] for Drupal that especially targets that. It allows site administrators to define event-condition-action rules, that represent workflows on a high abstraction level. The actions are executed after an event was triggered and if the conditions are satisfied. For example after a user updates some content (this is the event) she must not match the original author (this is the condition) then the original author is notified per e-mail that his content was changed (this is the action). More complex rules are possible and events provide a data context (e.g. affected content, user, etc.) that can be used and extended by the actions.

The Rules module is extensible and allows developers to easily implement new events, conditions or actions. They can be combined with the existing components and offer new possibilities when creating workflows. This flexible approach solves the problem of recurring needs and keeps the definition of a rule on a high level that is easy to understand and maintain.

Free and open source software

Drupal is licensed under the terms of the GNU General Purpose License (GPL) and is therefore free and open source software. All Drupal extending modules must be released under the same terms which creates a huge ecosystem of freely available software. This is one reason of Drupal's success because people can inspect the source code and contribute improvements and bug fixes back. When building web applications it is not necessary to reinvent the wheel all the time; people can instead work collaboratively on new features and modules.

It is also important for new concepts and ideas to be developed in an open manner in order to be accepted by the community. Only free and open source modules will get wide adoption and development momentum. Therefore the implementation of this project will also be released as free and open source software to comply with the Drupal

[5]Rules module: http://drupal.org/project/rules

licensing requirements on one hand, and to encourgage other developers to co-operate on the other hand.

1.2 Problem statement and goal

As we saw there is an increasing need to integrate web applications with web services and to manage complex tasks in highly abstracted workflows. Currently there is no uniform solution for Drupal to connect arbitrary web services without extra programming effort. For Drupal users it is not possible to specify web service metadata and then make use of them in a workflow system like Rules. There are several Drupal modules available that integrate with one selected service, but they do not offer generic support for other services nor are they designed to be used in rules or workflows.

Furthermore there is no framework in Drupal to allow the composition of web services. Often it is a use case for workflows to use multiple external services to exchange data or to trigger follow-up actions. A major problem in this regard is the transformation of data that has to fit different formats for different services. There is no conversion tool that maps inputs and outputs of services between services and Drupal and there is no integration for Rules yet.

The goal of this project is to explore existing concepts and implementations and to embrace them to the needs of workflows with web services in Drupal. The focus will be on a web service abstraction module and on the Rules module integration to accomplish this task.

1.3 Outline

This thesis is structured in the following chapters:

Chapter 2 gives some overview of theoretical concepts of web services and their architecture. Then also web service composition is covered where existing paradigms are examined that provide a foundation for this work. Content Management Systems, i.e. Drupal, are discussed and important modules in the Drupal ecosystem are introduced.

Chapter 3 contains objectives and goals that are addressed by the implementation of this project. It describes the requirements that have to be met in order to fulfill the project goal.

Chapter 4 goes into the details of the practical part of this thesis. Design and realization are discussed and technical solutions are presented. A new web service client module is introduced and its relationship to the Rules module is explained.

Chapter 5 describes the use case of an automated translation workflow that applies the implementation to demonstrate the functionality of the developed solution.

Chapter 6 will give an overview of related work and other systems that deal with similar problems.

Finally chapter 7 concludes the document and outlines the findings and lessons learned during this work. Future aspects and open issues are discussed.

CHAPTER 2

Foundations

Wanda: But you think you're an intellectual, don't you, ape?
Otto: [superior smile] Apes don't read philosophy.
Wanda: Yes they do, Otto, they just don't understand it!
– from the movie "A Fish Called Wanda"

In this chapter I will lay out some technology foundations that are necessary for my work. It will cover existing concepts and approaches that form a basis for the developments I am going to present in chapter 4.

2.1 Common protocols and standards

There are a lot of standards around the web and services, so I will cover some common of them here, which will be later mentioned and referenced.

XML The eXtensible Markup language is a data format or more generically a way to define data formats. It has consistent and clean text tagging, it separates content from format and allows hierarchical data structures. It also has facilities for user-definable data structures [UG98], which is a central feature needed by web services.

HTTP The Hypertext Transfer Protocol is the standard application layer protocol to exchange hypermedia and other resources on the web. It is designed for client-server style request-response communication patterns and it is stateless, which means that every request-response interaction is independent from any other. HTTP is a light weight protocol and is widely used and implemented on many systems. The current version of the protocol 1.1 which is defined by RFC2616 [FGM+99].

2.2 Web Services

Service Oriented Architecture

Before going into details with web services one should have a decent understanding of the underlying paradigm called Services Oriented Architecture (SOA), which is an abstract concept in software engineering. The key components are services that are independent from each other and interact on a well defined communication channel with each other. There are several properties that *services* fulfill [PTDL07, SHM08]:

- **Platform independent interface.** Services can be accessed in a standards-based manner.
- **Self-contained.** Services are modular and provide their functionality independently of other services.
- **Loosely coupled.** A service is a "black box", e.g. service consumers do not need to know about underlying technical internals of the service.

SOA is not tied to any specific technology but rather implies some driving forces according to Michael Stal [Sta06]:

- **Distribution.** Software components of the system run in different locations on a network. They need to communicate via a protocol.
- **Heterogeneity.** Different software entities may be implemented in different technologies. Integration must be possible without knowing detailed contexts.
- **Dynamics.** How the system is comprised may change at runtime and cannot be assumed statically.
- **Transparency.** As a result of heterogeneity and dynamics service providers and consumers are oblivious to implementation details of a service.
- **Process-orientation.** Services allow for composition in more coarse-grained workflows.

As we see SOA is a perfect fit for complex systems that need to integrate various independent subsystems. In order to make use of the services they must be discoverable by service requesters and publishable by service providers. This is often accomplished by a service registry, where services can be looked up and registered [Pap08]. Figure 2.1 visualizes the interaction of these roles.

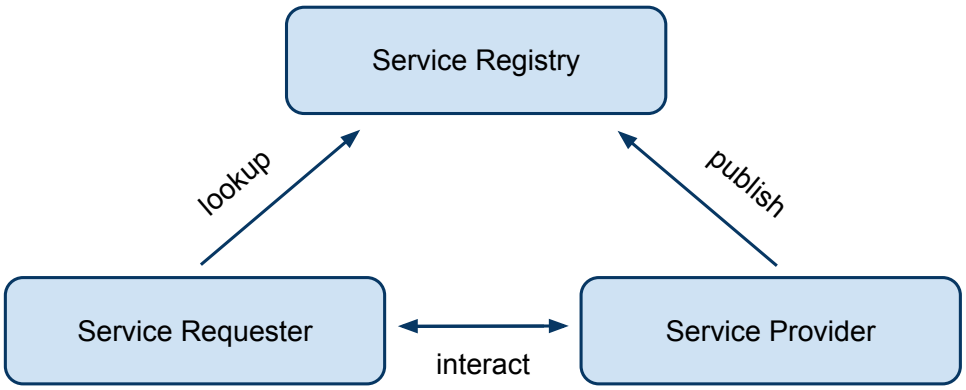


Figure 2.1: SOA roles and their relationship.

WS* Web Services

One possible realization of SOA is the classical WS* protocol stack. It is called WS* because most existing standards in the protocol family have abbreviations that start with "WS". The World Wide Web Consortium (W3C) has a definition of web services in their glossary [W3C04]:

> *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*

Web services of this kind are often also called *Big Web Services*, *SOAP oriented Web Services* or *WSDL based services* [Bru09]. As the names already suggest, there are three core standards that are significant: SOAP, WSDL and UDDI. All of them make heavy use of XML as a basic expression format. Figure 2.2 shows how these standards play out in the roles of SOA.

SOAP

The Simple Object Access Protocol is a standard to issue remote procedure calls and send/receive messages over the Internet. Commonly it uses HTTP as underlying transport protocol, but can be used with others as well. Messages are encoded with XML

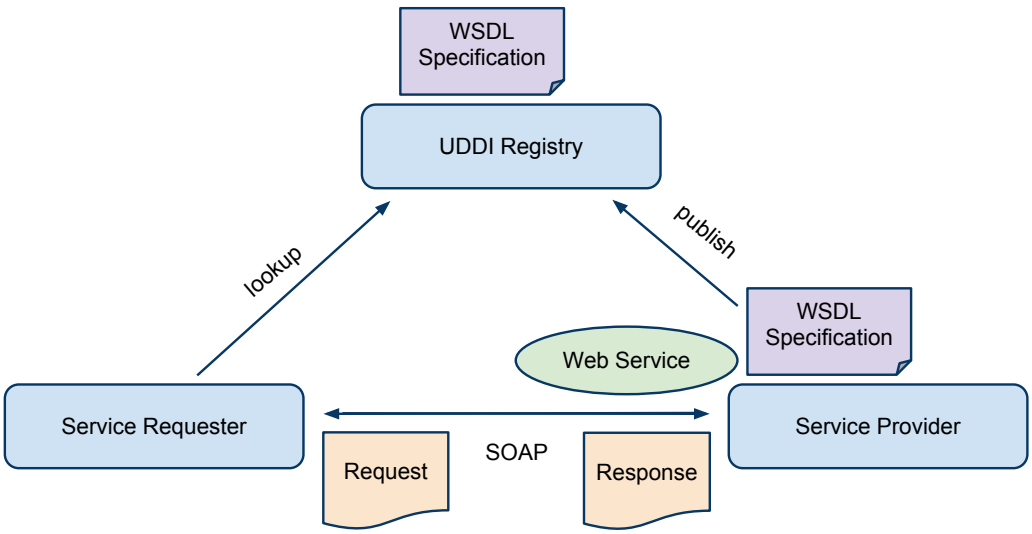


Figure 2.2: Web Service standards and their relationship in SOA.

and consist of an envelope for namespace definitions, an optional header for additional information (e.g. security, addressing etc.) and a body containing the message data itself, i.e. service operations and their arguments. There are two types of messages: service requesters send SOAP Requests and service providers send back SOAP Responses [TP02].

WSDL

The Web Service Description Language is an XML vocabulary to specify metadata for web services like where and how clients can invoke the service and what operations and arguments are available. WSDL is extensible and is designed as a machine-readable format, so that service consumer agents can pick up the necessary information about the service automatically. Currently WSDL 1.1 is the dominant version that is widely accepted, however WSDL 2.0 has been released as W3C recommendation in 2007, but has not been adopted by the industry that often yet [Bru09]. An alternative to WSDL is the Web Application Description Language (WADL), also an XML based description standard but intended specifically for RESTful web services (see section 2.2) [Bru09].

UDDI

UDDI is an abbreviation for Universal Decription, Discovery and Integration and implements the service registry in the SOA model. It allows service providers to publish

their service descriptions (i.e. WSDL) and service consumers consumers to lookup and locate web services they need. UDDI specifies the API to interact with such a registry via SOAP messages [Bru09, TP02].

The UDDI vision of central global registry where all available web services are available has not been realized so far and can be considered as a failure [DS05]. Instead, there are business specific or internal registries in use, or other channels to exchange web service metadata information are implemented.

Resource Oriented Architecture and REST

Resource Oriented Architecture (ROA) is a refinement of SOA with some additional architectural constraints [Ove07]. It is the basis for the second common type of web services – RESTful web services, see section 2.2 – besides WS* web services. The central entity in ROA is the resource, an abstract information item that has a name, a representation and references to other resources. The name plays the role of an identifier to address a resource. Representations of resources are data elements that are transfered between the actors in ROA.

REpresentational State Transfer (REST) is an architectural style of communication between web components and was first introduced by Roy Fielding in his dissertation in 2000 [Fie00]. It reflects the design principles of the World Wide Web (WWW), the largest and most complex distributed system nowadays. REST and ROA principles overlap in many aspects and they will be explained together here. The main characteristics of both can be described as follows [Bru09]:

Addressability. Resources are assigned with unique names that make them globally addressable in the system. Unified Resource Identifier (URI) is the standard to achieve this concept in the Internet, as described in RFC2360 [BLFM98]. It is important for clients that the naming scheme is meaningful and expressive.

Uniform interface. All resources can only be exchanged with four fixed operations: Create, Read (or Retrieve), Update and Delete. This ensures a very simple but sufficient pattern for communicating all relevant states of resources. It is no coincidence that HTTP itself provides similar methods to manipulate resources, which can be mapped to CRUD accordingly (see table 2.1).

Statelessness. Interaction between client and server is always opened and closed by one request-response sequence. This means that each request must contain all necessary information at once so that the operation can succeed [Fie00]. On one hand this allows flexibility and scalability, on the other hand information like authorization details must be sent in every request and can result in a worse network performance. However, it fits perfectly to the statelessness of HTTP.

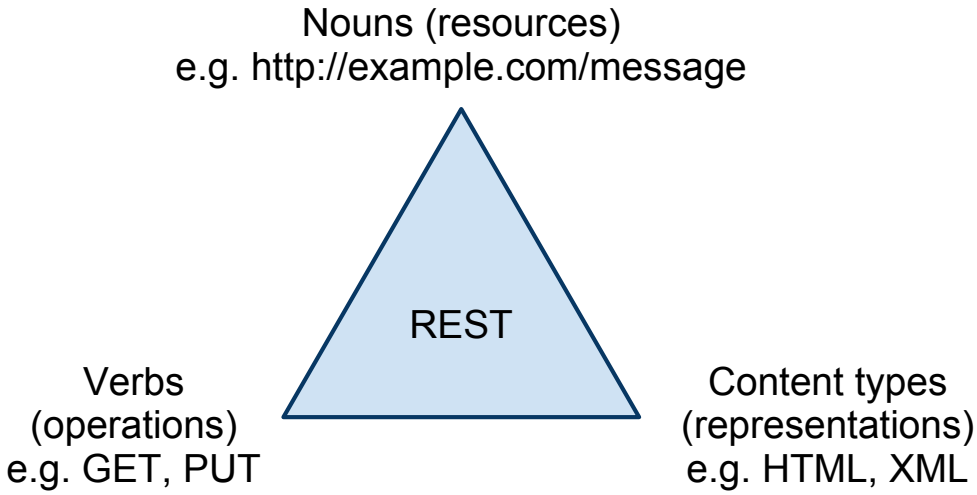


Figure 2.3: REST triangle with examples for resources, operations and content types.

Layered System. A hierarchical set of layers helps to manage system complexity and independence of components. “Each component cannot "see" beyond the immediate layer with which they are interacting” [Fie00], which allows the insertion of caches or load balancers as proxy network components [Bru09].

Table 2.1: Mapping CRUD operations to HTTP methods [BB08].

CRUD operation	HTTP method	description
Create	POST	Create a new resource or replace it if it already exists.
Read	GET	Retrieve a resource.
Update	PUT	Update an existing resource or create it if it does not exist.
Delete	DELETE	Delete the addressed resource.

Another perspective on REST is the REST triangle, which describes the semantics of the REST naming scheme. *Nouns* represent resources, *verbs* are used for operations on resources and *content types* define the representation of the resource (see figure 2.3) [Wil10].

RESTful Web Services

Since there are so many standards for WS* style web services and the protocol stack is overwhelming for service implementers, a new movement for RESTful web services that follow the REST principles came into existence. The goal is to work with simple and scalable services that make heavy use of existing web standards and leverage the full potential of the underlying protocol features. All of the SOA design principles (see section 2.2) apply to RESTful web services as well, but the focus is more on exchanging resources with services instead of the remote procedure call (RPC) style in WS* services.

Many Web 2.0 platforms offer RESTful web services to provide their functionality to third parties, a popular example is the Facebook Graph API[1]. The vast majority of operations on those RESTful services are GET operations for retrieving resources; POST, PUT and DELETE are not used that often.

JSON

Besides the technology standards HTTP, URI and XML there is one further common content type format for RESTful web services: JSON (JavaScript Object Notation). It is described in RFC4627 [Cro06] and is a light weight data format that is used to carry a resource's representation. Although JSON originates from JavaScript it is considered language independent and is supported by many platforms [Bru09]. An advantage of JSON over XML is that it can be used directly in client-side JavaScript interpreters, e.g. no parser is needed, which results in a performance gain [NPRI09].

REST-RPC hybrids

The concept of REST is not implemented fully on many RESTful services today. Due to the fact that RPC semantics are well known and are used in WS* services, many "REST" services followed and employ them as well. Those services that violate one or more constraints of REST are called *Hybrid Web Services* [RR07] or *REST-RPC Hybrids*. Some common misconceptions regarding REST are [Bru09][PZL08]:

- **RPC semantics in the payload.** Services use the HTTP payload as an envelope for carrying an operation request rather then using the correct HTTP request type in the header.
- **Ignoring HTTP method semantics.** Services do not use the correct HTTP request type for CRUD, e.g. HTTP GET with an extra query parameter is implemented for all four operations.

[1] Facebook Graph API: `http://developers.facebook.com/docs/api/`

- **Ignoring HTTP header facilities.** Services put information about authorization or response encoding into query parameters instead of using the destined HTTP headers.
- **One endpoint catches all.** Services misuse URI by putting the resource name in a query element, so that several resources live at the same base URI.

Service description

In order to make RESTful services metadata machine-readable, a description format like WSDL is needed. However, some authors like Joe Gregorio argue that REST does not need a description format [Gre07] because it cannot be reliable enough for the dynamics of the changing web. Nevertheless there are several approaches to provide the service description [Bru09]:

- **WSDL 1.1** is the most used standard for WS* services, but lacks capabilities to fully describe RESTful service characteristics.
- **WSDL 2.0** is the new standard and provides great flexibility to also describe RESTful services, but it is not in wide spread use and can be considered unsupported by most platforms.
- **WADL** The Web Application Description Language is an XML based standard as well and was specifically developed for RESTful services as counterpart to WSDL in the WS* world. It is well founded but is also not that common in real world service implementations.

2.3 Web Service composition

For larger business processes and workflows it is necessary to combine different web services that carry out a specified task together. We speak of web service composition when new processes or applications are built with existing web services by linking them together. The result of the composition is called a composite service and it can be part of another composition as well, leading to a recursive invocation of services. Dustdar and Schreiner describe that as follows [DS05]:

> [Web service composition] *allows the definition of increasingly complex applications by progressively aggregating components at higher levels of abstraction. A client invoking a composite service can itself be exposed as a web service.*

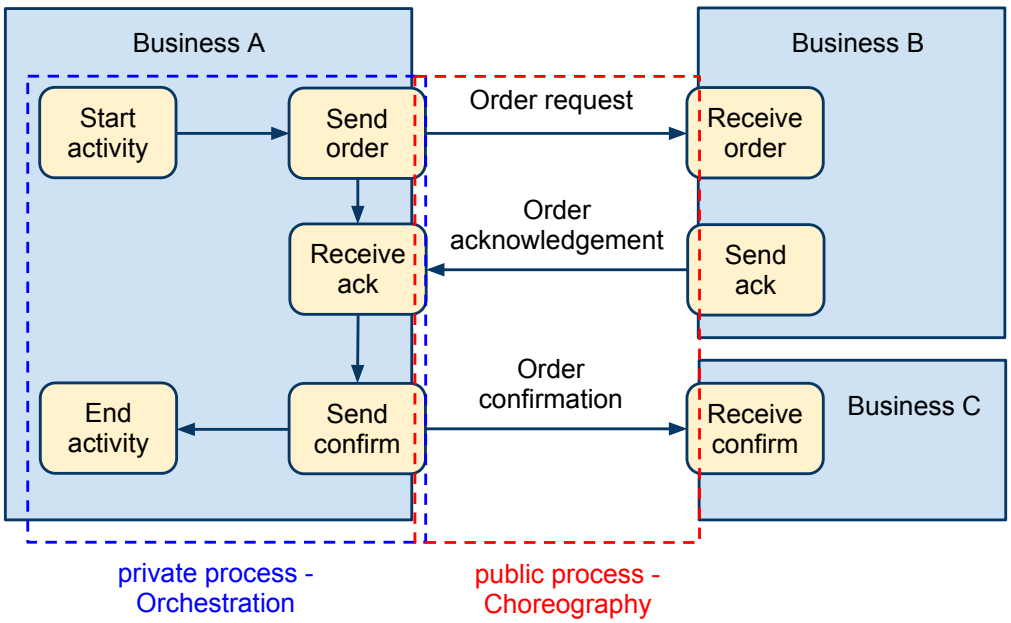


Figure 2.4: Example business activities to illustrate the difference between orchestration and choreography.

In principle there are two types of service selection strategies: *static*, which means that the services to be composed are selected at design time and *dynamic*, which means that concrete services are decided at runtime [DS05]. Service composition is a hot research topic, as there are several complex issues like how to represent such an abstract composition process, interoperability of services, data mapping or efficiency and performance of composition solutions. Scholars focus mainly on classical WS* services when they speak of web service composition, but recently there are also developments regarding RESTful services [Pau09].

Orchestration vs. choreography

There are currently two main approaches for syntactic web service composition: WS *orchestration* and WS *choreography*. We refer to orchestration as the private executable business process and to choreography as the public, observable exchange of messages (see figure 2.4 for an example). Both terms overlap somehow and can be described with the following properties [tBBG07]:

Orchestration. A central coordinator (the orchestrator) composes a business process of

web services and is responsible to invoke them and to form a workflow. Existing web services are reused and are part of the composition. A common industry standard protocol for web service orchestration is WS-BPEL (see section 2.3).

Choreography. Equal parties take part in a business collaboration and communicate in a peer-to-peer model. There is no central coordinator; instead there is a conversation definition that determines the interactions between the participants. WS-CDL is the corresponding protocol standard which exists in theory but has not been adopted widely in the industry.

WS-BPEL

The *Web Services Business Process Execution Language* provides an XML based vocabulary to describe web service compositions. It relies on WSDL and a process defined in WS-BPEL can be exposed as a service described by WSDL [tBBG07]. As already mentioned it is primarily intended for the web service orchestration approach, although it provides some support for choreography as well.

In WS-BPEL *processes* are defined in a block-structured manner and contain several *activities* that are the basic components of a process. *Partners* are external services that interact with a process; they are integrated via their WSDL descriptions as partner links. *Containers* serve as data providers that hold variables of input or output messages. A process is organized with structured activities that arrange basic activities, here are the most import ones summarized from the official OASIS standard [JE+07]:

- Basic activities:
  - **Invoke** – send a request to an external web service (to a partner)
  - **Receive and Reply** – provide a web service operation to a partner
  - **Assign** – copy data from one variable to another or insert new data from expressions
  - **Throw and Rethrow** – signal internal faults and propagate faults
  - **Wait** – wait for a certain period of time and delay the execution
  - **Exit** – immediately end a process
- Structured activities:
  - **Sequence** – execute a collection of activities sequentially
  - **If and Switch** – conditional behavior by executing a matching branch with associated activities
  - **While and RepeatUntil** – loops for repetitive execution of activities until a condition is met

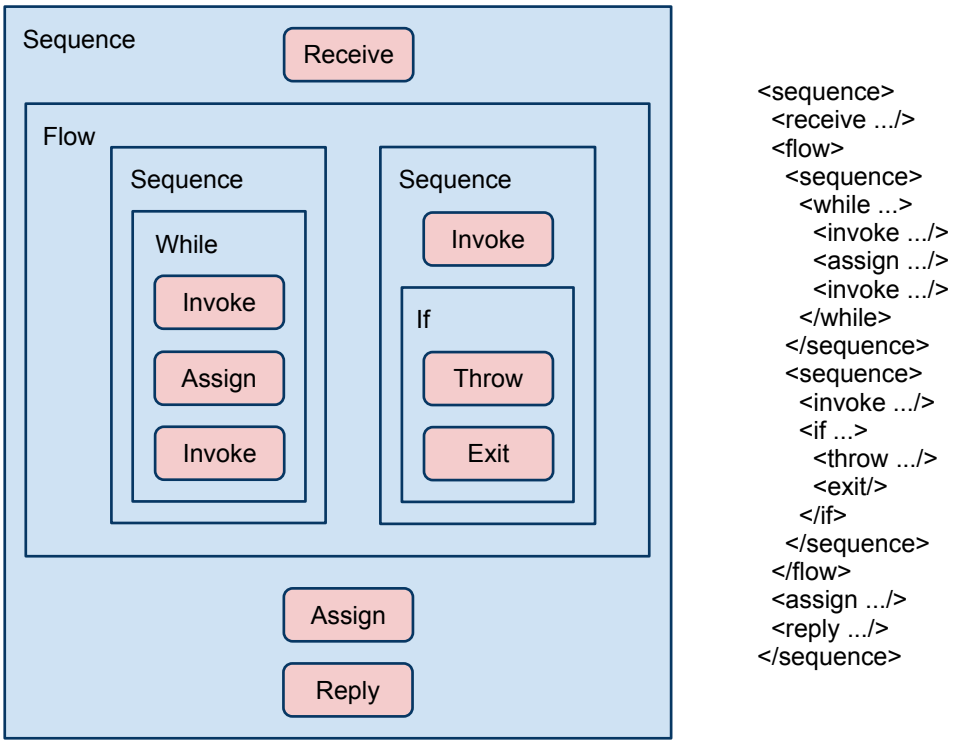
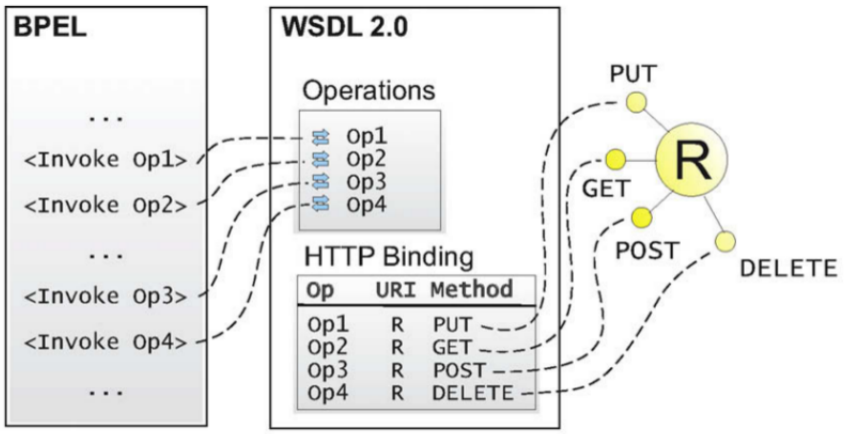


Figure 2.5: A BPEL process example with structured activities that contain basic activities and manage the behavior of the process.

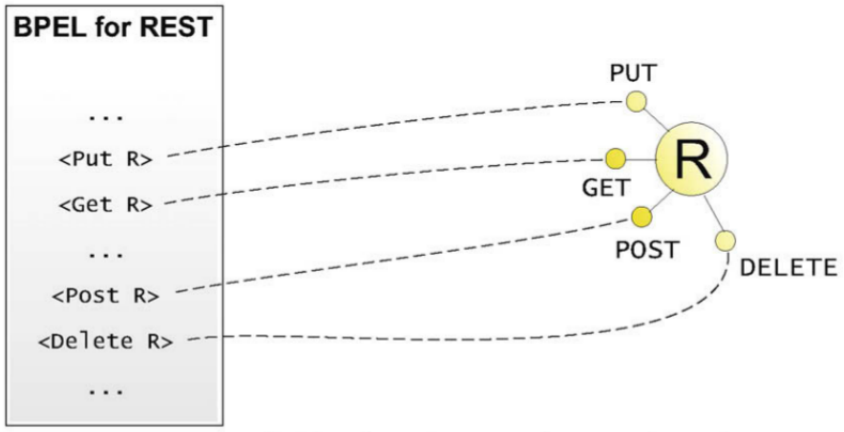
- **Pick** – events are associated with activities, which are executed when the event occurs
- **Flow** – execute activities in parallel and wait until all of them are finished
- **ForEach** – loop using a counter

Figure 2.5 shows an example how those activities play together in a block diagram and in the representing XML.

WS-BPEL is tightly coupled with WSDL 1.1 and is therefore not really suitable for RESTful web services. Even if WS-BPEL would support WSDL 2.0 (which is capable of expressing REST properties, see section 2.2) it would be too clumsy to express connections to RESTful services efficiently.



(a) Wrapping RESTful Web services through the WSDL 2.0 HTTP Binding



(b) Direct invocation of RESTful Web services using the BPEL for REST extensions

Figure 2.6: Solutions to compose RESTful web services in WS-BPEL either with WSDL 2.0 or BPEL for REST [Pau09].

BPEL for REST

BPEL for REST addresses the issue of integrating RESTful web services in process orchestration and provides an extension for WS-BPEL [Pau09][Pau08]. The four possible resource CRUD invocations of a RESTful web service could be mapped to operations in WSDL 2.0 and thereby used with the `<invoke>` BPEL language expression, but then service consumers would have to create the WSDL document for a RESTful web service themselves, which would contradict the principle that service providers should maintain the web service description [Pau09]. BPEL for REST takes an approach of a deeper BPEL language integration, so that the Resource Oriented Architecture of a RESTful web service can be better embedded and has the advantage of keeping resource semantics. Figure 2.6 visualizes the two possibilities of handling RESTful web services in WS-BPEL and also shows the GET, POST, PUT and DELETE expressions used in BPEL for REST to directly access remote resources. Cesare Pautasso claims that "explicitly controlling the RESTful interaction primitives used to invoke a service and native support for publishing the state of BPEL processes as resources from a process would be beneficial" [Pau09].

Mashups

Mashups are another form of web service composition with a focus on aggregating, mapping, filtering and remixing of web content. In contrast to the enterprise-centric WS* protocols, mashups are more end user oriented and loosely couple mostly simple services [LHSL07]. An important aspect of mashups is that they are user-generated, which distinguishes them from classical web service compositions that are mostly created by IT experts. The services that are used in mashups include Web 2.0 technologies like AJAX, semantic web protocols like RDF, syndication feeds like RSS and Atom, REST/SOAP based web services and even screen scraping of web sites [Mer09]. By using that Web APIs a mashup aims to expose a new web application. Mashups are created in a web browser and may be connected to mashup provider sites that may assist in the creation process. The resulting mashup application is executed partly server-side on the mashup provider and partly client-side to assemble the mashed content in the client web browser. The retrieval of mashup content may not only be the provider's responsibility, but also the client browser can be delegated to fulfill all or part of the communication with the external Web APIs. Figure 2.7 illustrates the architecture of mashups.

The big advantages of mashups are their ease of use (no developer needed to build it) and the ability to compose them ad hoc in a standard web browser. On the downside they are often limited to pre-defined services and they are not capable of implementing complex business tasks. A famous example of a mashup provider is Yahoo Pipes[2].

[2] `http://pipes.yahoo.com`

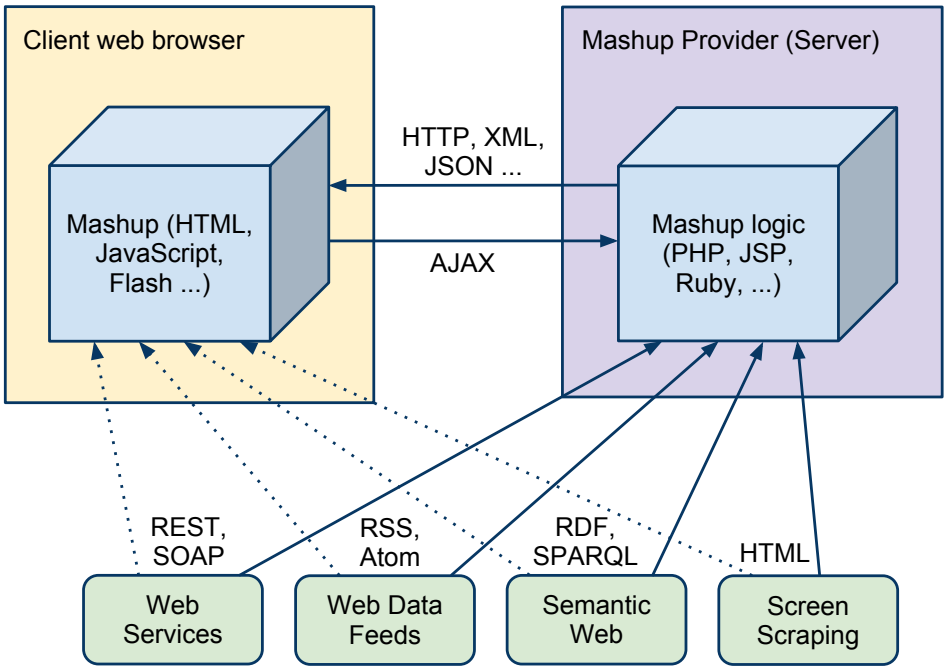


Figure 2.7: Mashup architecture with external Web APIs and their connection to server and client side.

2.4 Web Content Management Systems

Building a web site has become an increasingly complex task as there are many different people involved, e.g. "a team of content providers, editors and designers that strive to deliver up-to-date and correct information" [GN02]. *Content management system* (CMS) is a term that comes from content publishing and content repositories approaches [LLSL08] that deal with preserving structured information. A *Web Content Management System* provides content as a standard web application and allows collaboration and efficient administration of that content. However, when we use the acronym *CMS* in the web engineering domain, we refer to a Web CMS, strictly speaking.

One original purpose of a CMS was to relieve the technical burden of creating web content:

> *A Content Management System (CMS) can be defined as a database of information and a way to change and display that information, without spending a lot of time dealing with the technical details of presentation. Information is usually displayed in a web browser window.* [Sim05]

There are different types of CMS today, e.g. general purpose CMS, blogs, portals or wikis [Del07]. They all help to organize content in various ways and there are several requirements that all of them should meet [GN02]:

- **Separation of content and presentation.** Design templates or theming layers determine the layout and the appearance of the content. Multi-format content allows multilingual sites or adoption to mobile phones and PDAs.
- **Users, roles and permissions.** People interacting with the system must be authorized accordingly. Roles and permissions ensure a fine grained security policy.
- **Context awareness.** Content is personalized to the acting user and their predefined settings (e.g. browser version, previously visited pages, user preferences etc.).
- **Business processes and workflows.** Collaboration and interaction activities require coordination and management processes that can be automated and enforced by the system.
- **Extensibility.** The CMS must provide a comprehensive API and software module facility to allow developers to alter and extend the behavior of the system.

Most CMS have a database oriented architecture where content and settings are stored. They are often implemented in scripting languages and rely on a web server that delivers the dynamically created web pages. Popular systems written in PHP are Drupal[3], Wordpress[4], Joomla![5] and TYPO3[6], a CMS written in Python is Plone[7].

[3] Drupal: http://drupal.org
[4] Wordpress: http://wordpress.org
[5] Joomla!: http://www.joomla.org
[6] TYPO3: http://typo3.org.
[7] Plone: http://plone.org

2.5 Drupal

In this section I will introduce Drupal and the ecosystem around it, which is necessary to understand the developments that base upon them. Here is a brief description of what Drupal is [VW07]:

> *Drupal is used to build web sites. It's a highly modular, open source web content management framework with an emphasis on collaboration. It is extensible, standards-compliant, and strives for clean code and a small footprint. Drupal ships with basic core functionality, and additional functionality is gained by the installation of modules. Drupal is designed to be customized, but customization is done by overriding the core or by adding modules, not by modifying the code in the core. It also successfully separates content management from content presentation.*

Drupal is written in the scripting language PHP and makes use of procedural and object-oriented programming paradigms. It is developed as free and open source software by several thousand collaborating contributers world wide. Currently Drupal version 7 is being worked on, which will be the basis for the implementations introduced in this thesis. Drupal gained popularity because of its extensibility, scalability and flexibility and powers over 1% of all Internet web sites[8]. There are big sites among them, e.g. from IBM, NASA, Yahoo, Sony, MTV and Whitehouse.gov [Zie10].

Drupal core architecture

Drupal is a set of PHP scripts and bases on several underlying technologies outlined in figure 2.8. Drupal's core architecture is composed of a library of common functions and several core modules. This includes components for user management, session management, a URL and menu system, logging, localization (internationalization), templating (theming), a form system, basic content management and more [VW07]. There are further core modules that provide additional features on top of that basic functionality, e.g. user profiles or RSS feeds.

Modules are a central concept of extensibility in Drupal. They wrap certain features and interact with the core via API functions and the hook system. *Hooks* allow modules to take part in the data and control flow of Drupal core, e.g. modules can manipulate variables, add information or trigger other activities. A module can register to a hook by implementing a function with a certain naming scheme, so that this function is called when Drupal core invokes the hook. This architectural style can be seen as some sort of aspect-oriented programming; more details on concepts and Drupal programming styles

[8] Usage of content management systems for websites: http://w3techs.com/technologies/overview/content_management/all

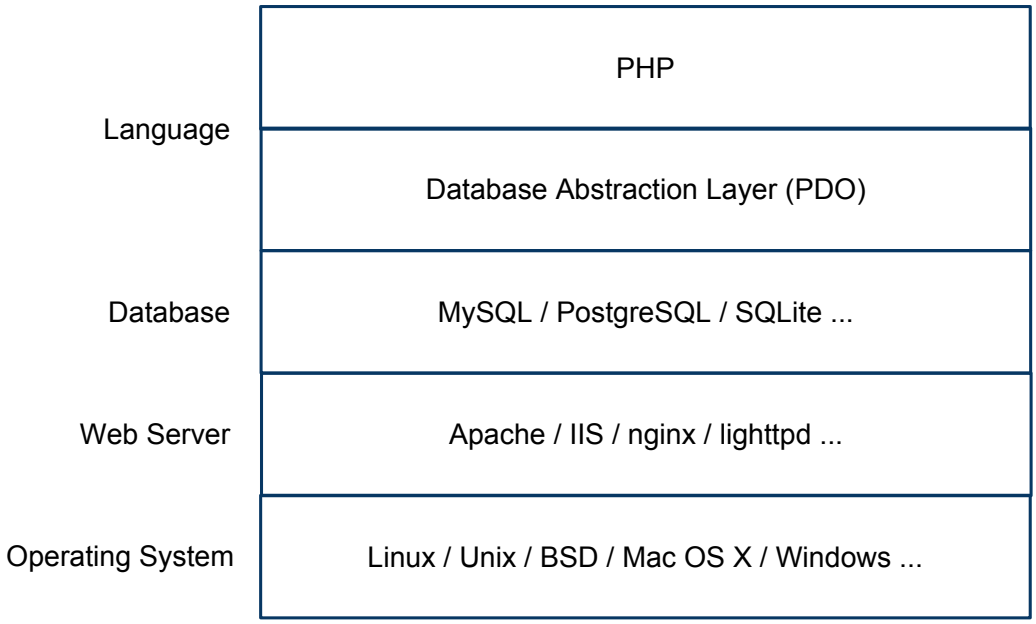


Figure 2.8: Drupal's technology stack [VW07]

can be found on drupal.org [dc09]. Currently there are over 6,000 contributed modules hosted on drupal.org[9] that extend the features of Drupal.

Besides hooks there are other "Drupalisms" that are important to understand how Drupal works. Configuration information is often organized in nested PHP arrays, a flexible and high-performance data structure. However, this has the disadvantage of a error-prone description, as syntactic mistakes in array keys often go unnoticed. *Callbacks* are function name strings that are stored as values in configuration arrays and are used to dynamically invoke functions when the array is processed. These arrays are also used as *renderables*, i.e. to represent form structures that are later rendered to XHTML. Jeff Eaton gave a good introduction to Drupal internals from an architect's point of view at Drupalcon San Francisco[10].

Content is often referred to as *nodes* in the technical Drupal vocabulary. Nodes represent the basic building block of a Drupal site, e.g. nodes are blog posts, pages or articles. Comments, files, ratings etc. can be attached to nodes [Zie10].

[9] Drupal contrib modules: http://drupal.org/project/modules

[10] How Drupal Works: An Architect's Overview: http://sf2010.drupal.org/conference/sessions/how-drupal-works-architects-overview

Entities and Fields

Entities are a new concept in Drupal 7 that aim to replace nodes as the generic content and data container. Thereby entities unify nodes, users, comments, profiles etc. as one common abstract representation. This allows modules to implement features only once for entities, which then applies to all kind of entity types (nodes, users etc.). Therefore entities are a powerful tool to even support future (yet unknown) entity types, instead of tying the module functionality to nodes only. "As example consider a rating module: Built upon the concept of entities users could utilize it to allow rating nodes, comments, taxonomy terms or even other users" [Zie10].

Fields are also a new development in Drupal 7 that derives from the contributed module *Content Construction Kit*[11] (CCK) in Drupal 6, which allowed to attach fields to nodes. Nodes have basic fields such as a title and a body, whereas CCK fields are additional custom properties, such as e.g. a date information or an image field. Those fields are configurable per content type, so that it is possible to build different content configurations with different data properties. However, in Drupal 7 this functionality has been reworked to a Drupal core module that not only equips nodes with fields but entities as well. This empowers site builders to assign fields to various entity types, so that data properties can be easily attached to nodes, users, comments, taxonomy terms etc. Fields can be configured not only per entity type, but also per *bundle*. A bundle can be described as one set of fields for a certain entity type [Zie10] [N+10]. An example would be the profile entity type, where one bundle is a user profile and a second bundle is a company profile, both with different fields.

Entity API and Entity Metadata

The API support for entities is very basic in Drupal core, so there is the Entity project[12] in the contributed section of drupal.org to leverage advanced aspects of entities. It consists of two major features, the Entity CRUD API and the Entity Metadata abstraction. The first one provides a class for full CRUD (Create Read Update Delete) support for entities and an extended controller class for additional needs as mass loading or deletion. The second one deals with describing entity properties as metadata by providing a uniform interface that exposes properties, fields and entity references of an entity type. Thus it is very useful for entity type agnostic modules that can make use of the metadata annotations to deal efficiently with arbitrary entity types. This means that "any module providing an entity would have to provide metadata only once to be integrated with all modules building upon the uniform interface" [Zie10]. The project was started and mainly developed by Wolfgang Ziegler to satisfy the need of data abstraction for the Rules module (see the next subsection).

[11] Content Constrction Kit: http://drupal.org/project/cck
[12] Entity project: http://drupal.org/project/entity

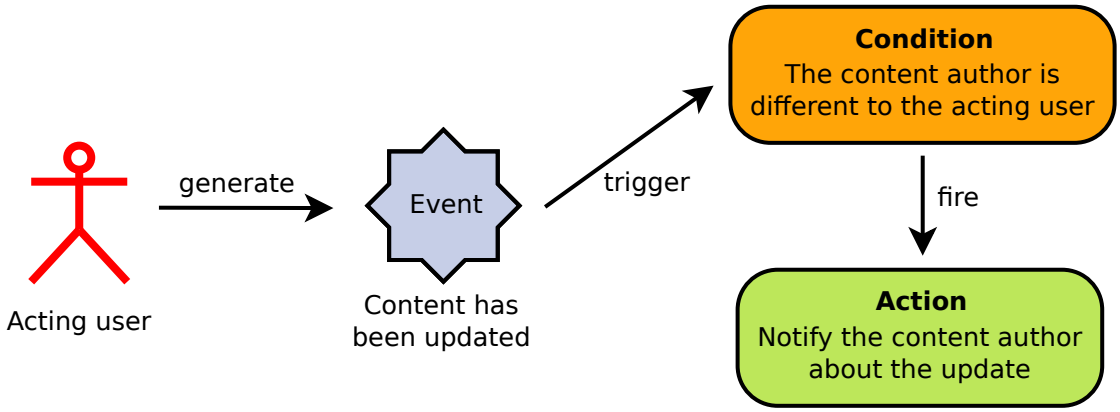


Figure 2.9: An Event-Condition-Action rule that reacts when a user updates a node to notify the node author [Z+10a]

Rules

The Rules module[13] is a workflow system for Drupal that allows site builders to easily define custom activities. It bases on the concept of Event-Condition-Action rules, where on the occurrence of a predefined event one or more conditions are evaluated and upon success one or more actions are executed. They are also called reactive rules and figure 2.9 shows an example flow in Drupal. The Rules module offers a wide range of events, conditions and action so that very many combinations of them can be used for flexible workflow building. This enables site builders to automate a lot of regular tasks without any programming effort – just by configuring rules accordingly. Rules can also be attached to more than one event and rules can be bundled in reusable *rule sets*. Those rule sets can then be executed as an action from another rule. Other supportive features around Rules include exportable configurations to copy/share rules, scheduling of rules to postpone execution and a modular design to allow Rules integration from other modules [Zie10] [Z+10a].

A major aspect of Rules is handling data that is shared between events, conditions and actions. Data is stored in variables that can be provided by events and actions, for example the "Content has been updated" event provides a node object. Version 2 of Rules relies therefore on the Entity Metadata module to offer so called *data selectors* for direct access to entity properties and relationships to other entities. This means that for example the name of the author of a node can be accessed by a chained selection from the node entity onwards to the user entity to the name property. Additionally Entity Metadata enables Rules to provide generic entity conditions and actions, such as for example creating, loading or deleting entities, which can be applied to any kind of

[13] Rules module: `http://drupal.org/project/rules`

entity type. Furthermore there is support for *data lists* and looping over them to execute an action for each item of the list [Zie10].

Rules Web

Wolfgang Ziegler has developed support for distributed rules in his master thesis published as *Rules Web* project on Github[14] [Zie10]. It includes so-called *Rules Web Hooks* that specify remote events for Rules, so that occurring events can be passed to other Drupal sites. This is realized via a notification system, where the source Drupal site exposes a remote event and other sites can subscribe to it. When the event is triggered all subscribed sites are informed and receive the event information (and possible data variables as payload). On the receiver site rules can be configured to process the remote event and to react with follow-up actions. All communication is done via HTTP requests and responses, remote event providers make use of the Services module[15] to expose remote events and subscribers use the REST client module by Hugo Wetterberg[16] to subscribe to an event.

This system is build on the concept of *remote proxies* that form an abstraction layer for different kinds of remote systems (see figure 2.10). Rules Web Hooks represent one remote proxy (one endpoint type); there are other endpoint types in the Rules Usecases project[17] to also support REST and SOAP services. Service invocations are integrated as Rules actions and require a service definition in code to describe operations, parameters, returned variables and other settings. Communication with SOAP services is achieved by using the PHP SOAP extension[18], RESTful services are accessed with the help of the REST client module by Hugo Wetterberg. As a result it is possible to invoke web services with Rules now, but the module lacks an administration user interface and it has not been published to drupal.org (it can be seen as an experimental proof of concept module).

[14] Rules Web: http://github.com/fago/rules_web

[15] Services module: http://drupal.org/project/services

[16] REST client module (renamed to HTTP client): http://github.com/hugowetterberg/http_client

[17] Rules Usecases: http://github.com/fago/rules_usecases

[18] PHP SOAP extension: http://php.net/manual/en/book.soap.php

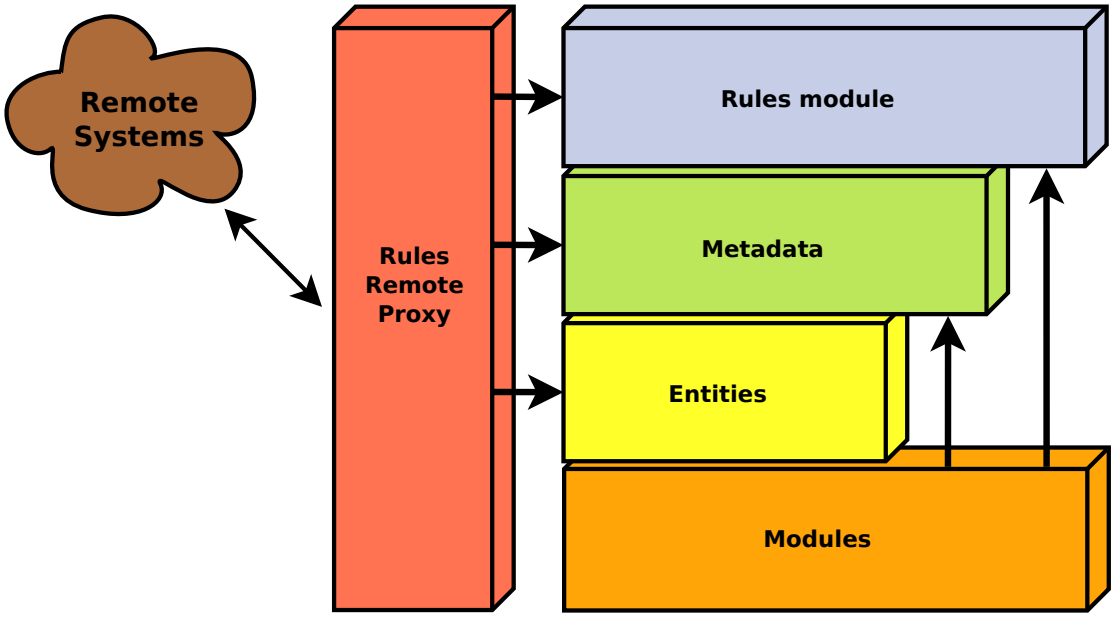


Figure 2.10: Module architecture of Rules Web. "A remote proxy may provide new entities, metadata as well as events, conditions and actions to the system." [Zie10]

CHAPTER 3

Objectives

The philosophers have only interpreted the world, in various ways. The point, however, is to change it.
– Karl Marx

This chapter layouts some finer grained objectives that form the goal and purpose of this thesis. I will describe properties and requirements that the developed system should achieve.

3.1 Web service client module

In order to efficiently deal with web services we need to wrap all functionality in a Drupal module. This module shall act as a web service client and shall manage the communication with different service types. SOAP and REST service types should be both supported by the module, which should provide an abstraction mechanism to allow an easy integration of other service types. The design of the module should take extensibility into account and should provide a decent developer API so that Drupal programmers can easily use a high level web service interface.

The work from Wolfgang Ziegler on *Rules Web* (see section 2.5) should be analyzed, extended and embraced to enhance the existing approach. The improvements should result in a finalized package published on drupal.org that is compatible to the upcoming Drupal 7 release. *Rules Web Hooks* shall be adapted to base on this new module and should be packaged for drupal.org as well.

3.2 Web service composition with Rules

Another major requirement is to consider the invocation of multiple web services in one workflow. Thus the planned web service client module should not only account for single, separated service operations, but for a composed usage of services. The aim is to leverage the Rules module (see section 2.5), which already provides workflow features and a "Rules language" to handle variables and data types between events, conditions and actions. When we manage to express web service invocations as Rules actions and provide mapping of different data structures between that actions, we should get a decent system to arrange multiple web services. The goal is to get a somewhat similar functionality compared to WS-BPEL (see section 2.3), so that a rule represents a process with service invocations, data assignments, loops and so on. Of course Rules is more limited in its language constructs and does not reach the richness of WS-BPEL or EMML (Enterprise Mashup Markup Language [All09]), but it should suffice to satisfy the basic needs of service composition. Furthermore it should keep creation and management of workflows simple and usable.

3.3 An automatic translation use case

The practical use case of the web service client module should be an automatic translation workflow use case. Several translation web services shall be used to acquire English translation suggestions for German terms in a Drupal taxonomy vocabulary. That suggestions shall then be forwarded to a machine learning component by communicating via a web service interface. The machine learning component then ranks the translations according to their relevance and returns the score as result of the web service call. The translations shall be stored with the score in a new vocabulary that is ready for human examination to finally select the correct translation. This workflow is comprised of multiple web service invocations that shall ensure the correct behavior of the web service client module. Figure 3.1 shows the web service calls that are necessary for this task. Chapter 5 describes the use case in detail.

3.4 Web service integration without programming effort

Handling external web services was most often connected to some development effort in order to accomplish service invocations. The developed web service client module should make it possible to administer web services without any programming effort. This requires an administrative user interface in Drupal to create, lookup, update and delete web service descriptions that are used to communicate with the actual services.

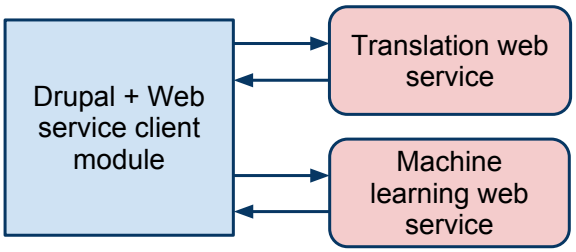


Figure 3.1: Service invocations in the automatic translation use case.

In conjunction with the Rules module and the provided Rules integration it should allow a complete configuration of web services in the Drupal administration user interfaces. However, basic knowledge of web services, operations and the involved data structures will still be needed in order to understand and configure the services correctly. A major difficulty in this regard is the graphical specification of complex data types that may be needed for a service, which should be resolved as well.

3.5 Automatic WSDL parsing

SOAP services provide a WSDL description in most cases (see section 2.2) which can be used to obtain metadata like operations and involved data types from the service. Service consumers can therefore dynamically configure their binding to the service by extracting the required information from the WSDL description. Concerning the web service client module this means that the manual specification of operations, data types, binding etc. is not needed for SOAP services as long as there is a WSDL description available. The module should provide a way to let users specify the location of a WSDL description and then generate the internal service information automatically. That reduces the configuration of a SOAP service to a minimum and is less error-prone than manually entering operations or data types.

3.6 Sharing of exportable web service descriptions

A web service description that is created on the platform should be exportable so that it can be easily transfered to other Drupal sites. This process requires a serialization of the descriptions to a structured string format. The format should be human-readable as well, so that it can be managed in revision control systems in a meaningful way. As a result it should be possible to share web service descriptions across system borders and to publish those descriptions in repositories or other online resources. The export

functionality requires a mirrored import functionality that is capable of restoring the original description from flattened export string. Furthermore it is important to install a decent dependency resolution mechanism in case that service descriptions share data types, so that the dependencies are exported as well.

Chapter 4

Realization

Developers, developers, developers, developers, developers, developers!
Developers, developers, developers, developers, developers, developers!
– Steve Ballmer at a developers' conference[1]

Now that we have some basic foundations (see chapter 2) and defined the scope and objectives (see chapter 3), we go into the concrete realization. This chapter consists of analysis, the system architecture considerations and some details on the implementation. The source code that was developed during this thesis can be found as web service client project on drupal.org[2].

4.1 Analysis

At the heart of the planned module are web services, so we need to consider how we will abstract and represent them in a way that they fit into existing Drupal and PHP facilities, as well as the Rules module (see section 2.5) environment.

Web service model

Support for SOAP, RESTful and REST-RPC hybrid services is required, which means that we need to specify common service properties that apply to all service types. However, different service types may require additional settings to properly describe how the service can be used. This leads to an abstract, basic and generic service description that is extensible per service type and also allows possible future service types that do not exist yet.

[1] YouTube video: http://www.youtube.com/watch?v=8To-6VIJZRE
[2] Web service client: http://drupal.org/project/wsclient

We can define that each web service has the following properties that are necessary to establish successful connections:

- *Name* and *Label*: A machine-readable name identifies the web service description internally and a human-readable label briefly describes the service.
- *Type*: The type of the web service determines how the service must be used and which type of implementation (endpoint) will handle the communication. This is *REST* or *SOAP* in our implementation.
- *URL*: Each service has a base URL that is used either directly for communication (in the case of a RESTful service) or as pointer to a document that formally describes the service (in the case of a SOAP service this would be the WSDL file).
- *Operations*: We can define that every web service has operations. This applies naturally to SOAP services and REST-RPC hybrids, but also applies to strict RESTful services by considering the four standard CRUD methods that form operations as well (see also section 2.3 for a similar example of modeling strict RESTful service operations in WSDL 2.0). An operation can have an arbitrary number of parameters and optionally a result.
- *Data types*: A service may deal with complex data types that are used as parameters or result types in an operation. They are described by a name and properties that are primitive or complex data types themselves.
- *Settings*: Depending on the type, a service may need to store additional endpoint type-specific settings (e.g. authorization credentials or data formatting details).

While name, label, type and URL are simple properties of a service description, operations, data types and settings are collections of complex structures. In the tradition of Drupal and PHP we organize complex data sets in associative array structures, that are easy to access in the programming language and run fast during program execution (see also section 2.5). Figure 4.1 visualizes the information structure of a web service description. Green properties are primitive fields, red properties are collections of complex structures and purple properties refer to other complex structures. Arrows represent references and the dashed line for variable types states that it may also be a primitive type, which does not need an explicit definition.

Depending on the endpoint type, the information structure of a web service description can be extended to store additional properties that are necessary to invoke the operations. For example in case of the REST endpoint a URL suffix may be needed for a specific operation.

Listing 4.1 is an example for the structure of a web service description, in this case a REST-RPC hybrid service with one operation (“translate”). Operation and data type

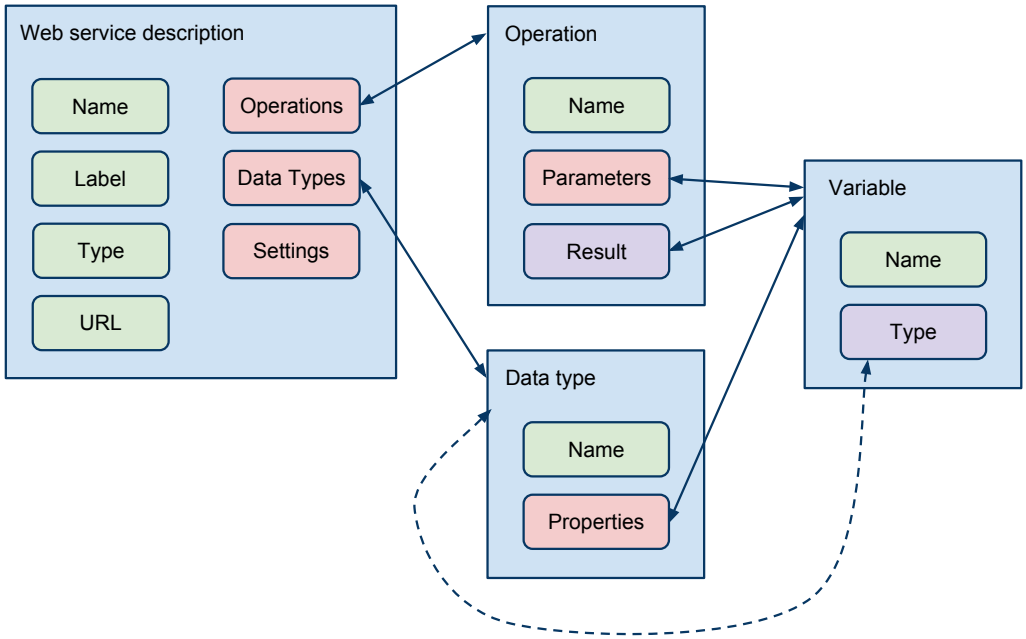


Figure 4.1: Information structure of a web service description.

information is provided in nested properties and contains details about the data format; it specifies how and what can be exchanged with the service.

```
<wsclient_service>
  <name>google</name>
  <label>Google Ajax APIs</label>
  <url>http://ajax.googleapis.com/ajax/services/</url>
  <operations>
    <translate>
      <label>Translate text</label>
      <url>language/translate</url>
      <parameter>
        <q>
          <type>text</type>
          <label>Text</label>
        </q>
        <!-- ... other parameters ommitted here ... -->
      </parameter>
      <result>
        <type>translation_result</type>
        <label>Translation result</label>
```

```
      </result>
    </translate>
  </operations>
  <datatypes>
    <translation_result>
      <label>Translation result</label>
      <property_info>
        <responseData>
          <type>struct</type>
          <label>Response data</label>
          <property_info>
            <translatedText>
              <type>text</type>
              <label>Translated text</label>
            </translatedText>
          </property_info>
        </responseData>
      </property_info>
    </translation_result>
  </datatypes>
  <type>rest</type>
  <settings />
</wsclient_service>
```

Listing 4.1: Example web service description represented in XML.

SOAP service layer

Because SOAP is a widely implemented protocol, we do not want to re-invent the wheel ourselves but use a software library for PHP. It should be capable of creating and exchanging SOAP messages as well as reading WSDL files to provide an abstraction layer on the actual operations and endpoints. There are two libraries for PHP that seem to be actively developed and to fulfill the requirements, one is NuSOAP[3] and the other is PHP SOAP[4]. As PHP SOAP is part of the official PHP distribution and is included in most PHP server installs, it is reasonable to choose this extension because of the larger user base.

PHP SOAP comes with a `SOAPClient` class that allows accessing SOAP services in an object-oriented way. It offers a constructor with an option to specify a URL to a WSDL file, which is then downloaded and processed. The web service operations are mapped dynamically to object methods, so that they can be invoked easily from the `SOAPClient` object. A usage example is given in listing 4.2, where the Geocoder.us SOAP service is used to retrieve the zip code of a given address.

[3]NuSOAP PHP library: http://nusoap.sourceforge.net/
[4]PHP SOAP extension: http://php.net/manual/en/book.soap.php

```
// Create new SOAPClient instance with metadata from the WSDL
    file.
$service = new SOAPClient('http://geocoder.us/dist/eg/clients
   /GeoCoderPHP.wsdl');
$result = $service->geocode_address('1600 Pennsylvania Av,
   Washington, DC');
$zip_code = $result[0]->zip;
// $zip_code is now 20502
```

Listing 4.2: Invoking a web service with PHP SOAP.

Although the SOAP extension works fine in most cases, it has some limitations. WSDL is only supported in version 1.1, which is not a big issue as version 2.0 is rarely used nowadays. Also the *Document/wrapped* operation parameter convention is not supported, where all parameters are automatically wrapped into one complex operation parameter that has the same name as the operation [AAM06]. Thus programmers cannot pass the parameters one by one to the `SOAPClient` method, but need to put them into a wrapping array data structure themselves, which is then the single parameter for the method. This is inconsistent and confusing for developers that are used to work with other common frameworks where the wrapping is hidden and automatically done.

RESTful service layer

RESTful services are somewhat easier to access, as they do not need such a sophisticated data encapsulation like SOAP envelopes. Nevertheless we need a library that supports different payload formats (commonly XML and JSON) and that provides an API to make use of the different HTTP request methods (GET, POST, PUT, DELETE). Drupal itself offers the `drupal_http_request()` [5] function for simple remote calls, but it does not support all HTTP request types and it lacks a proper exception handling in case of errors. A more advanced approach is implemented by the HTTP client module[6] that contains a `HTTPClient` class for object-oriented use with RESTful services. Additionally it offers support for various data formats that are wrapped implicitly, all HTTP request types, authentication mechanisms, exception handling and it is flexible for adjustments and extensions.

Listing 4.3 gives an example of using the HTTP client module for translating a German word to English with the Google translation service.

```
// Prepare a JSON formatter
```

[5] API for `drupal_http_request()` : http://api.drupal.org/api/drupal/includes--common.inc/function/drupal_http_request/7

[6] HTTP client module: http://drupal.org/project/http_client

```
$formatter = new HttpClientBaseFormatter(
   HttpClientBaseFormatter::FORMAT_JSON);
$service = new HTTPClient(NULL, $formatter);
// Translate the german word "Schule" to English
$parameters = array(
  'q' => 'Schule',
  'langpair' => 'de|en',
  'v' => '1.0',
);
// Invoke a HTTP GET request.
$result = $service->get('http://ajax.googleapis.com/ajax/
   services/language/translate', $parameters);
$translation = $result['responseData']['translatedText'];
// $translation contains now "School"
```

Listing 4.3: Invoking a RESTful service with the HTTP client module.

Complex web service data types

Web service operations that make use of primitive data types in their parameters and return values are relatively easy to handle – the type information is implicitly available, which is important for preparing service input variables and for further processing of service output variables. In case of complex data types that are required for the service operation, we need metadata about the type and its properties. This is not only required to embed the service in the system, but also for Web Service Composition (see chapter 2.3) where data types have to be transformed or adapted between different services.

For our goal of integrating web services with Rules we need to consider the already existing data type system of Rules and Entity Metadata. It takes into account high level Drupal entities such as nodes, users, comments etc. but also other data structures that can be defined by third party modules. The challenge is to map data type expectations from web services to the type system in Rules, so that we can seamlessly transfer data or data properties between the workflow components. SOAP services most often include XML schema definitions (XSD) about the complex data types in their WSDL file, which can be extracted and mapped automatically in most cases. RESTful service data types on the other hand are almost never described in machine processable formats [Gre07], but rather specified informally on the service provider's web page or in other casual ways. This leads to the requirement of letting users (site builders that integrate the service) specify complex data types with their properties, so that Rules knows about the metadata and can supply that information when building workflows with web services.

Import/Export format

An established web service description on one Drupal site is most probably interesting for other sites as well, so that they do not need to create such a description themselves, but simply reuse the existing configuration to connect to the web service. Sharing of configurations is accomplished by many Drupal modules through serialization to a string that contains executable PHP code. Although this is easy and straight forward, it imposes a major security risk to every Drupal site. Potentially arbitrary PHP code can come with a malicious configuration import which is then executed. Even if the permission to import web service descriptions is restricted to site administrators that should know what they are importing, a security risk still remains. So the serialization to PHP code does not satisfy the security requirements and is therefore off the table as option for an export format.

Another possibility is to use the existing web service description standards, e.g. WSDL or WADL. As stated in chapter 2.2 WSDL 1.1 is not capable of describing RESTful services, so it will not fit to our needs. WADL is specifically targeted at RESTful services, but it is not intended to describe SOAP services as well. WSDL 2.0 is technically capable of describing both service types, but it is not in wide spread use. However, the biggest problem is the extensibility of the web service client module; new endpoint types can be defined and additional settings can be stored. It seems difficult to anticipate future developments and if they will fit into the structure WSDL or WADL with all their properties.

This leads back to a custom format that is able to perfectly map all internal data structures that comprise a web service description. The Rules module leverages JSON as import/export format [Zie10] and it seems to be a viable solution in our case as well. PHP and Drupal have built-in support for JSON, so the programming effort for data conversion is kept to a minimum. JSON is also human-readable, lightweight and resource-efficient when it is parsed [NPRI09].

Developer API

Programmers need a simple and concise way to make use of existing web service descriptions, e.g. to issue web service invocations. The web service client module should provide an abstraction layer so that developers need to know as little as possible about the configuration in order to use it. This is especially important regarding the endpoint type of a service, meaning that services can be used without knowing whether they are RESTful or SOAP services. Listing 4.4 shows how a web service description object is loaded and a web service operation is invoked by calling a method on that object. Compared to listing 4.3 it does not require tedious setup routines anymore when using the service, because the settings were configured and stored with the web service description before.

```
// Load the Google translation service
$service = wsclient_service_load('google_translate');
// Invoke the 'translate' operation of the service
$result = $service->translate('Hallo Welt', 'de|en');
$translation = $result['responseData']['translatedText'];
// $translation contains now "hello world"
```

Listing 4.4: Loading a web service description and executing a web service operation.

Web service composition

For the realization of complex workflows that contain several web service invocations, we could develop our own workflow system that is capable of composing multiple web services. However, this seems to be a big task and would probably duplicate a lot of code that already exists in the Rules module, a workflow system in Drupal. The execution of a rule is triggered by an event, then conditions are evaluated and upon success actions are executed. Obviously we need to provide an integration to the Rules module, so that (multiple) web services can be used in a Rules configuration. Therefore some considerations:

1. Invoking a web service operation is a Rules action.
2. Preparing complex data structures as web service operation parameters is done as a "create data structure" Rules action beforehand.
3. A rule can contain an arbitrary amount of actions, also multiple web service invocation actions. Data that needs to be passed between services can be mapped with new data structures and "create data structure" Rules actions.

The arrangement of such actions is shown in figure 4.2 where some example invocations and data structure creations are carried out in the action block of a rule.

With this basic concept we can accomplish web service composition within Rules workflows and get additional features of the Rules language (e.g. loops, rule scheduling, rule sets, other plugins etc.) for free.

4.2 Architecture

For the realization of the web service client module we consider the following architectural conditions that will help us with a clean and elegant implementation style:

- **Object-oriented programming**: We will leverage PHP language features such as classes, interfaces and inheritance to make the implementation modular, coherent and extensible.

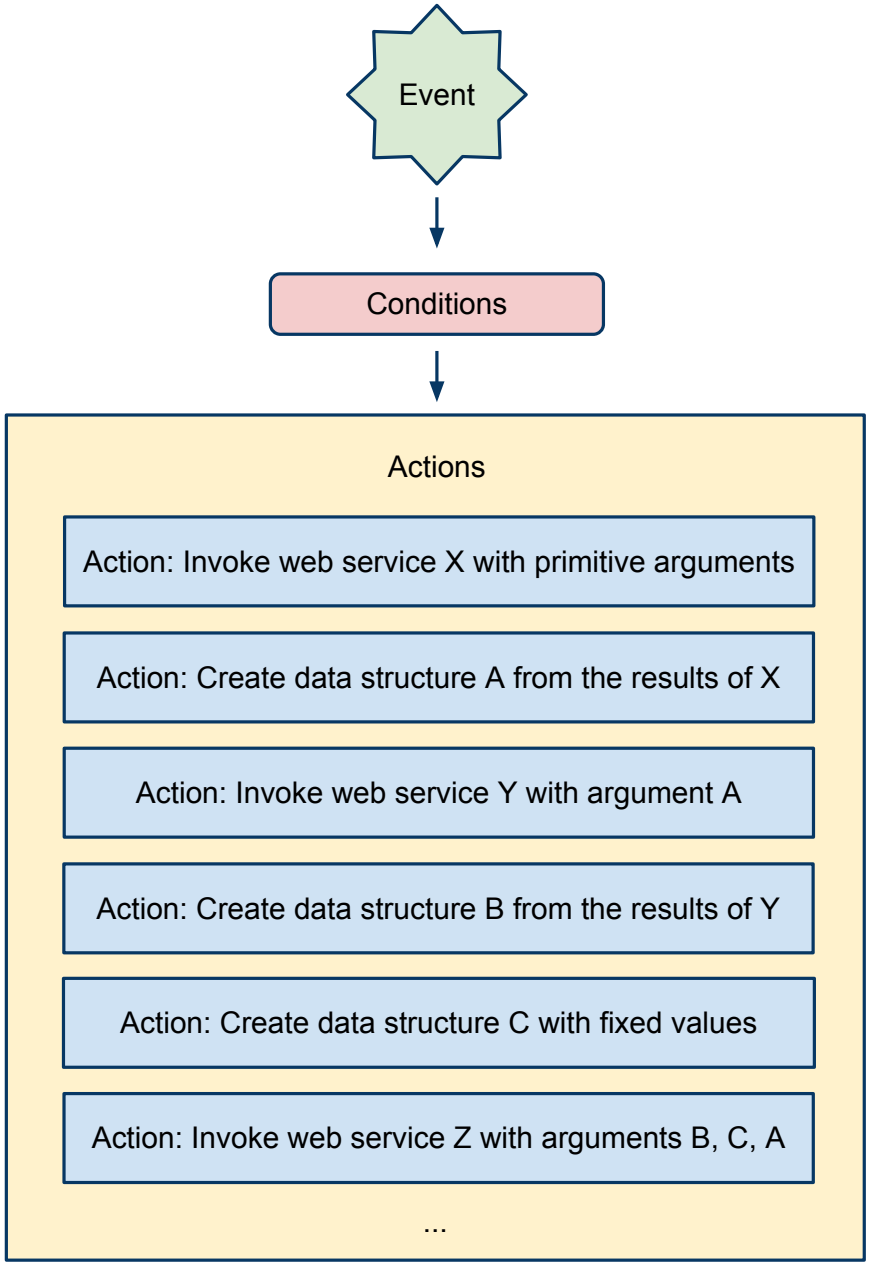


Figure 4.2: Web service composition in Rules with actions for invocation and data structure creation.

- **Drupal Entities**: Drupal 7 and the Entity API module offer a system to handle common storage operations (CRUD) and generic integration with other subsystems and modules (see chapter 2.5). We will define web service descriptions as entities, so that we benefit from an already existing abstraction layer that reduces development effort.
- **Modularity**: The usage of the web service client module may depend on the use case, e.g. some sites will only use it in form of a code dependency to another module, while others will need the full administration user interface. The functionalities of the module will be wrapped into submodules, so that the required code base is minimized if not all features are used.
- **Automated tests**: Drupal 7 also provides a unit testing framework called SimpleTest[7] that allows modules to implement test cases that verify the functionality of the module. This aspect does not strictly belong to the architecture, but will contribute to an improved and sustainable code base.

To realize the modularity, we decouple the whole web service client package into four Drupal modules.

1. `wsclient` : This is the core web service client module that implements the basic features to deal with web service descriptions. It provides integration with the Entity API module, the Rules module and the Features module (export, see section 4.3). It does only provide an abstract endpoint class, concrete service adapters (i.e. for SOAP and REST services) are separated into their own modules. A dependency to the Entity API module is necessary.
2. `wsclient_soap` : This module realizes the back end for SOAP services by providing a SOAP endpoint. It also handles the import web service descriptions from WSDL files and it depends on the `wsclient` module.
3. `wsclient_rest` : Also the endpoint for RESTful services is factored out to a separate module and also depends on the `wsclient` module.
4. `wsclient_ui` : The whole administration user interface is also located in its own module, so that the UI code is not loaded when only the developer API is required. Besides the dependency to the `wsclient` module it also depends on the Rules module, because it uses some Rules API functions.

Figure 4.3 illustrates the module structure and also shows the dependencies between them (solid arrows). Dashed arrows indicate no hard dependency but an optional integration if the referenced module is available in the system. Web service client modules

[7] Drupal's SimpleTest framework: http://drupal.org/simpletest

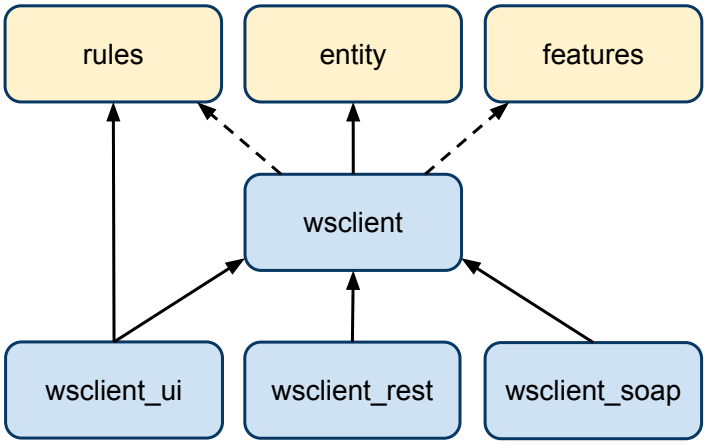


Figure 4.3: Web service client modules and their dependencies to other modules.

are marked as light blue and other external Drupal modules are marked as light yellow.

Figure 4.4 shows the structure of the core classes used in the web service client package (only the most important attributes and methods are outlined for the sake of simplicity and to give an overview). The `WSClientServiceDescription` class is at the center of the implementation and holds all information pieces that fully describe a web service (see also figure 4.1). It is derived from the `Entity` class which is provided by the Entity API module and which provides useful storage operations like `save()` and `delete()`. `WSClientServiceDescription` also implements the magic PHP method `__call` that catches all calls to not existing methods, so that a service operation can be directly invoked as method on the object (see listing 4.4 for an example).

The endpoint of a web service description is an important attribute that is determined by the type of the service (SOAP or REST in our case). For compatibility reasons, an endpoint has to implement the `WSClientEndpointInterface`; the most important method of the interface is `call()`, which is executed when an operation is invoked on the web service (i.e. the `invoke()` method of `WSClientServiceDescription` is called). The endpoint is responsible to handle the communication with the actual web service and to return a possible result. The abstract class `WSClientEndpoint` implements common functionality that is shared between `WSClientSOAPEndpoint` and `WSClientRESTEndpoint`. Both subclasses implement a `client()` method that constructs the underlying library to access the web service (i.e. a `SOAPClient` or a `HTTPClient` instance). Of course both classes also implement the `call()` method to invoke a service operation.

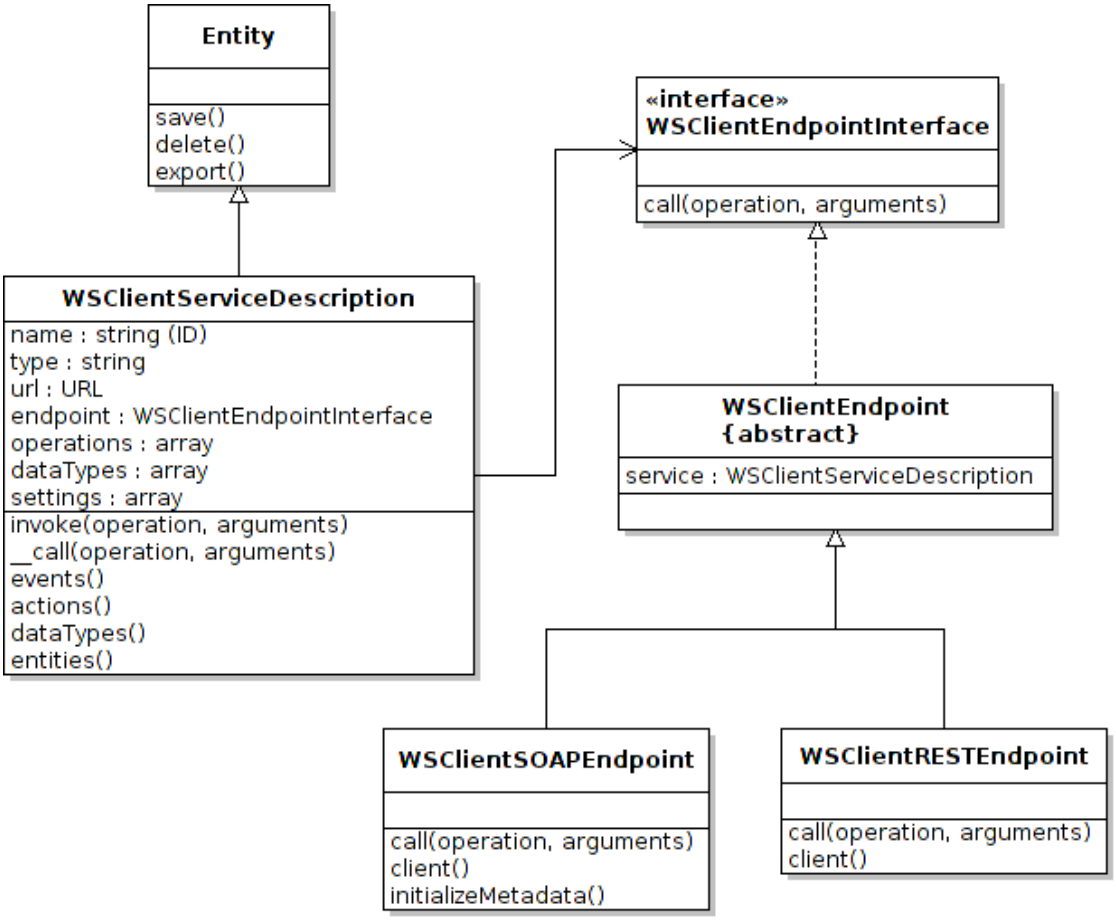


Figure 4.4: Class diagram of the web service client module.

Web Service descriptions as entities

The decision to use Drupal entities as framework for the web service descriptions is an important one – we need to store custom data (the web service descriptions) and want to access it in a standardized and simple way. Entities are a new concept in Drupal 7 and provide the facilities to easily integrate custom data structures in Drupal. The Entity API module extends the Drupal core entity features and helps to leverage the full potential of entities. This approach can be seen as an object-oriented mapping, where objects hold the data during program execution and a relational database retains the data for persistence. The mapping between objects and the database is carried out by the Drupal entity system.

To expose the web service descriptions as entities, we need to implement the following parts in our wsclient module:

- hook_schema() : This hook is located in the installation file of the module (wsclient.install) and specifies the database table and the database fields where the web service descriptions will be stored. It is invoked when the module is installed and creates the table in the database (see listing 4.5.

```
function wsclient_schema() {
  $schema['wsclient_service'] = array(
    'fields' => array(
      'id' => array(
        'type' => 'serial',
        'not null' => TRUE,
        'description' => 'The primary identifier for the
           web service.',
      ),
      'name' => array(
        'type' => 'varchar',
        'length' => '32',
        'not null' => TRUE,
        'description' => 'The name of the web service.',
      ),
      'label' => array(
        'type' => 'varchar',
        'length' => '255',
        'not null' => TRUE,
        'description' => 'The label of the web service.',
      ),
      'url' => array(
        'type' => 'varchar',
        'length' => '255',
        'not null' => TRUE,
        'description' => 'The url of the web service.',
      ),
      'operations' => array(
        'type' => 'text',
        'not null' => FALSE,
        'serialize' => TRUE,
        'description' => 'The operations this web service
           offers.',
      ),
      'datatypes' => array(
        'type' => 'text',
        'not null' => FALSE,
        'serialize' => TRUE,
        'description' => 'The complex data types used in
           the operations.',
      ),
      'type' => array(
        'type' => 'varchar',
```

```
        'length' => '255',
        'not null' => TRUE,
        'description' => 'The type of the remote endpoint
          .',
      ),
      'settings' => array(
        'type' => 'text',
        'not null' => FALSE,
        'serialize' => TRUE,
        'description' => 'The endpoint type specific
          settings.',
      ),
      'authentication' => array(
        'type' => 'text',
        'not null' => FALSE,
        'serialize' => TRUE,
        'description' => 'Data describing the
          authentication method.',
      ),
    ) + entity_exportable_schema_fields(),
    'primary key' => array('id'),
    'unique keys' => array(
      'name' => array('name'),
    ),
  );
  // Other secondary definitions ommitted here.
  return $schema;
}
```

Listing 4.5: Implementation of `hook_schema()` in the wsclient module.

- `hook_entity_info()` : Another hook that informs the system of the new web service description entity. It contains a pointer to the class that represents the entity, the name of the database table where it will be stored (the one described in the installation file), which properties are used to identify the entity and other details that are relevant for the system to fully recognize the entity (see listing 4.6).

```
function wsclient_entity_info() {
  return array(
    'wsclient_service' => array(
      'label' => t('Web service description'),
      'entity class' => 'WSClientServiceDescription',
      'controller class' => 'EntityAPIController',
      'base table' => 'wsclient_service',
      'module' => 'wsclient',
      'fieldable' => TRUE,
      'entity keys' => array(
```

```
        'id' => 'id',
        'name' => 'name',
        'label' => 'label',
      ),
      'exportable' => TRUE,
      'access callback' => 'wsclient_entity_access',
      'features controller class' => '
        WSClientFeaturesController',
    ),
  );
}
```

Listing 4.6: Implementation of `hook_entity_info()` in the wsclient module.

- `WSClientServiceDescription` : This is the class that extends the Entity base class and that is referenced in the entity info hook. It defines attributes that correspond to the database fields specified in the database schema. The attributes (properties) are mapped automatically to the database fields when entities are created, read, updated or deleted (CRUD). Of course the attribute data types used in an entity object need to match the types defined in the database schema, otherwise database exceptions will occur at runtime (the correctness of the mappings is not enforced).

As a result web service descriptions can be handled in an easy, object-oriented way, without worrying about how to access the database. Listing 4.7 is an example of the programmatic usage of web service descriptions in conjunction with CRUD operations.

```
// Create a web service description.
$service = new WSClientServiceDescription();
$service->name = 'google_api';
$service->label = 'Google Ajax APIs';
$service->url = 'http://ajax.googleapis.com/ajax/services/';
$service->type = 'rest';
// Save it to the database.
$service->save();
// Read a service description from the database.
$service = wsclient_service_load('google_api');
// Update a service description.
$service->label = 'Google Services';
$service->save();
// Delete a service description from the database.
$service->delete();
```

Listing 4.7: Entity CRUD operations on a web service description object.

Endpoints

The web service client module is designed to support multiple endpoint back ends that are some sort of plugins for various service types (e.g SOAP or RESTful services). As already explained in section 4.2, an endpoint is a class that needs to implement the `WSClientEndpointInterface` . Endpoints are registered with the wsclient module by implementing `hook_wsclient_endpoint_types()` that specifies the endpoint type name and the endpoint class. Currently we deal with three different endpoint types (additional service types may arise in the future):

- **`WSClientSOAPEndpoint`** : Represents the connection layer to SOAP services and uses the `SOAPClient` class from the PHP SOAP library for service calls. This class also contains a method to parse WSDL files when a new SOAP service description is initialized (see section 4.3).
- **`WSClientRESTEndpoint`** : Communicates with RESTful services by using the `HTTPClient` class from the HTTPClient Drupal module. The implementation currently only supports GET operations, but can be easily extended to support others as well.
- **`RulesWebHooksEndpoint`** : This is the endpoint class to realize Rules Web Hooks, which originates in the Rules Web module (see section 2.5) and which is ported to be compatible with the wsclient module. It is bundled as new Rules Web Hooks module[8] and is used as subscription/notification system to exchange remote Rules events.

Invoking web service operations

Figure 4.5 shows an example of the method call hierarchy that is executed when the Google Translate service operation is invoked. First a dynamic translate method is called on the web service description object, which is caught by the magic method handler `__call()` . The method name is then passed as operation name to the `invoke()` method, which checks if the operation exists and which maps the arguments to the named parameters. Also a hidden parameter (the version information that is required by the service) is added in this case. Next the endpoint interface is called, in this example the concrete implementation is the REST endpoint. In the last step the endpoint applies adjustments for the client library (i.e. the operation URL is looked up from the operation name) and invokes the web service operation. Responses from the web service are passed back up to the original operation caller.

[8] Rules Web Hooks: `http://drupal.org/project/rules_web_hooks`

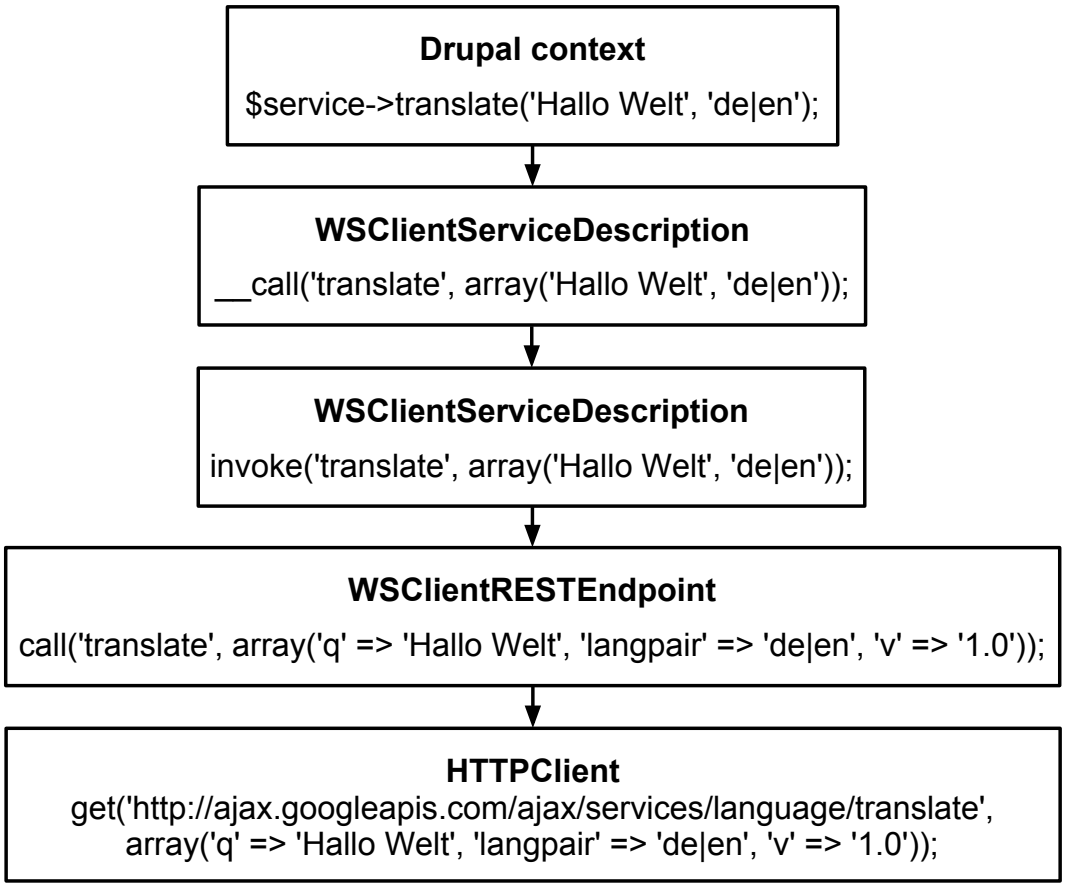


Figure 4.5: Method call hierarchy on a web service operation invocation.

4.3 Implementation

After reasoning in the analysis and architecture sections, this section will present details about the implementation of the objectives (see chapter 3).

Rules integration and service composition

The Rules module (see section 2.5 for an introduction) allows other modules to integrate their functionality within the Rules workflow system and the Rules language. For the web service client module this means that two important things need to be exposed to Rules: web service operations as Rules actions and complex web service data types, because Rules needs to know how to access operation parameters and return variables. There is online documentation for developers that explains in detail how Rules can be extended and used [Z+10b]. The code that achieves the Rules integration lives in the file

`wsclient.rules.inc` which contains several hook implementations and functions that are called from Rules:

- `wsclient_rules_action_info()` : This is a hook that returns information about additional actions that should be made available to Rules. All web service descriptions are loaded and all operations are mapped to action information arrays. Parameters of an operation are described as parameters of the corresponding action and result variables of an operation are specified as provided variables of the action. Additionally the name of the web service and the operation name are included as hidden parameters, because they are needed when the action is executed in order to know which web service and which operation should be invoked.

- `wsclient_rules_data_info()` : Also an information hook that provides web service specific data type details to Rules. This is relevant for services that deal with complex and nested data structures, so that in Rules all properties are accessible within a data structure. All data types of all web service descriptions are exposed to Rules. As a result complex data structures can be prepared to be used as parameters for a web service operation or specific parts of a returned web service result can be selected and processed in a Rules workflow.

- `wsclient_service_action()` : This function is registered as execution callback for all web service client actions. When a rule (or any other Rules component) that contains a web service client action is evaluated, this function is called to execute the web service operation. The arguments that are passed to this function contain the name of the web service description, the operation name and the parameters that should be forwarded to the web service operation. The actual web service invocation is carried out here and the response of the service is delivered back to the execution context of the calling Rules configuration.

That is basically the implementation on the web service client module side; it enables users/site administrators/developers to integrate web services in their Rules workflows. Another piece of development work has to be done on the Rules module side: a "create data structure" action is missing, which is needed to produce complex operation parameters. This action is not web service client specific, but may be needed by other modules that deal with arbitrary data types as well. The implementation of this action covers the following points:

- A new "creation callback" property is introduced for Rules data types, which is the name of the function that will initialize the data structure upon creation. It is needed because of different underlying data containers like arrays or PHP standard class objects that have to be created differently.

- Action information has to be provided, i.e. the name “Create a data structure”, a type parameter where all registered data types can be selected and the newly provided variable that will contain the new data structure.
- An action process callback dynamically evaluates the selected data type and adds all its type properties as parameters to the action configuration. Thus the static action information is extended by the details of the data type and the type properties are presented as input parameters to the action.
- Finally the action execution implementation invokes the type specific creation callback an returns the resulting new data structure es provided variable.

The development issue for the “Create data structure” action can be found in the Rules issue queue on drupal.org[9]. With that action in place, Rules is ready for basic web service composition.

The data flow between services can be managed with another very useful tool: the Rules data selector [Zie10]. When a Rules component (i.e. an action) is configured, parameters can either be provided in a direct input mode (the plain value) or variables available in the Rules configuration can be assigned with a data selector. This means that a property of a provided variable (e.g. the output of an action/service operation) can be mapped to a parameter of another action (e.g. the input of another service operation). Data selectors can not only be applied to service data, but also to other Drupal entities or data types. This also allows a convenient exchange of data between Drupal’s internal structures (e.g. content/nodes, users or other entities) and web services.

To clarify the service composition, figure 4.6 shows an example of two web service actions that are used in a Rules workflow. The rule is triggered on the event “after updating existing content” and first executes an action to invoke the Twitter search web service[10] where the node title is passed as argument. The service returns a complex data structure that contains a list of twitter messages and other useful data. A loop uses that list to execute actions for each item. First, the message contents is transmitted to the Google Translate service for translation to German. Second, the translation result is displayed as system message for testing purposes. This workflow can be configured completely in the administration user interface and does not require any programming effort. Figure 4.7 is a screenshot of the configuration overview page that lists all components (events, conditions, actions, loops, etc.) of this example rule.

Rules does not only provide an administrative user interface to compose rule configurations, but also a developer API that allows a programmatic setup. A similar example rule (without the event) can be configured and executed from code as shown in listing 4.8.

[9] Rules “create data structure” action development issue: http://drupal.org/node/849464
[10] Twitter search service: http://dev.twitter.com/doc/get/search

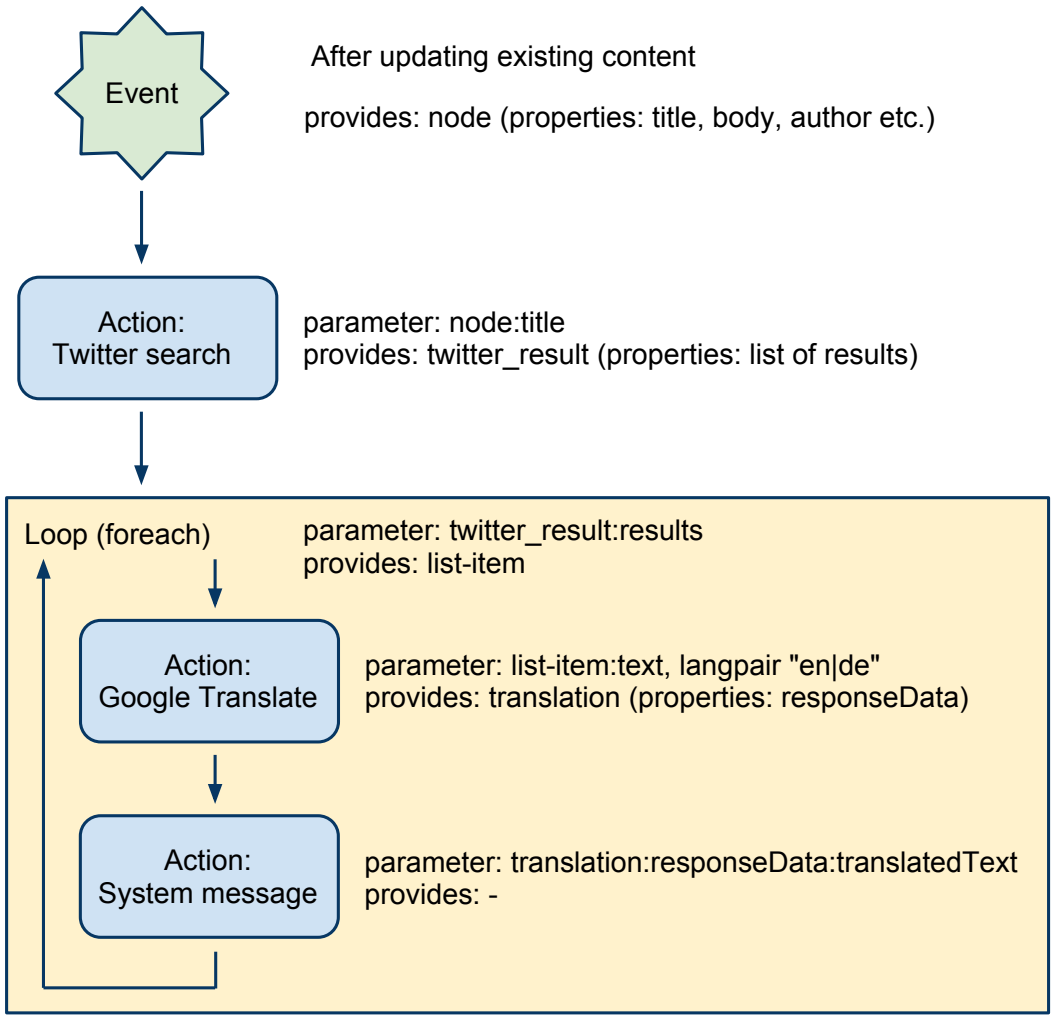


Figure 4.6: Rules configuration example with two web service actions and the use of data selectors to assign variables.

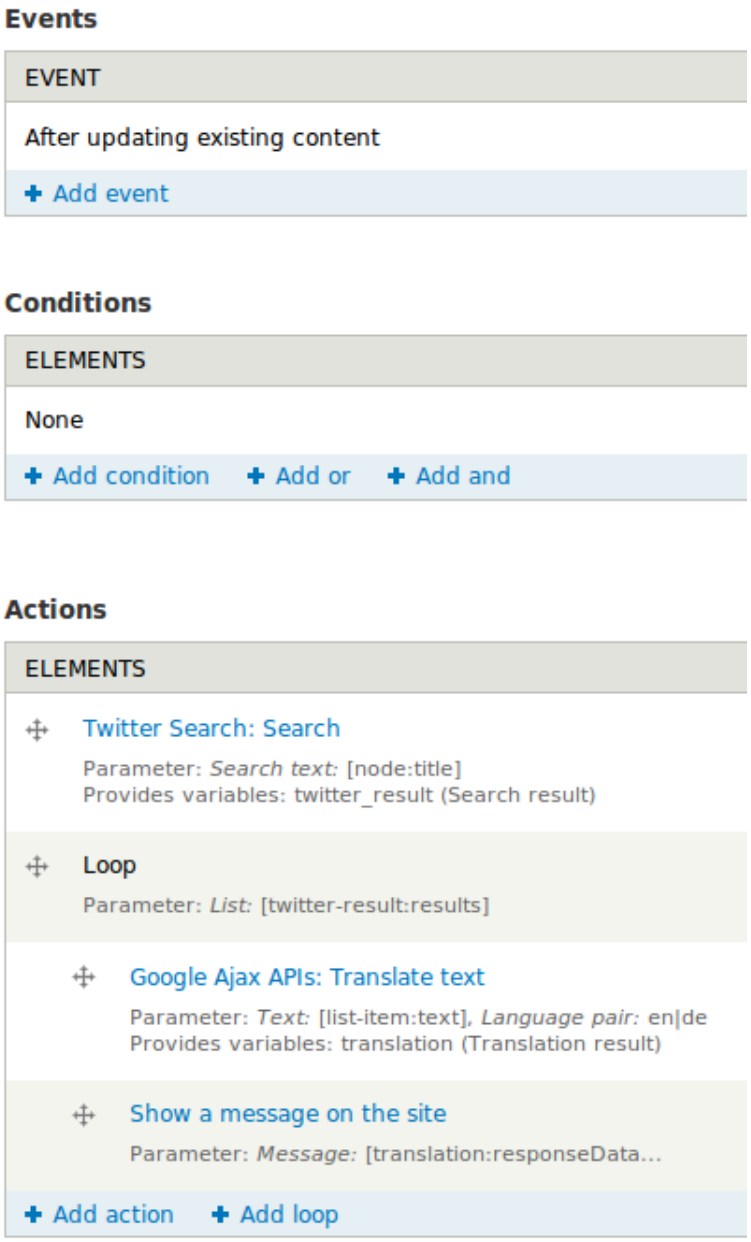


Figure 4.7: Screenshot of a rule configuration overview page with two web service actions.

```
// Create a new rule that accepts a text parameter.
$rule = rule(array('text' => array('type' => 'text')));
// Add the Twitter search web service action to the rule.
$rule->action('wsclient_twitter_search_search', array(
       'param_q:select' => 'text',
       'result:var' => 'twitter_result'));
// Create a loop that iterates over the Twitter messages.
$loop = rules_loop(array(
  'list:select' => 'twitter_result:results'));
// Add the Google Translate web service action to the loop.
$loop->action('wsclient_google_translate', array(
       'param_q:select' => 'list-item:text',
       'param_langpair' => 'en|de',
       'result:var' => 'translation'))
     ->action('drupal_message', array(
       'message:select' =>
         'translation:responseData:translatedText'));
// Add the loop the the rule.
$rule->action($loop);
// Execute the rule configuration with a text parameter.
$rule->execute('Example title');
```

Listing 4.8: A rule in code composing two web services by using the Rules developer API.

Administration user interface

Objective 3.4 requires an administration user interface (UI) in Drupal to accomplish the goal of a web service integration without programming effort. We aim to manage web service descriptions by providing interactive pages and forms where service metadata can be created and modified. Web service operations and data types can be configured so that they are available to Rules or other modules that want to make use of them. All user interface code will be separated out into a submodule (`wsclient_ui`), because the UI may not be needed in all use cases. Other modules that depend on the web service client module may not need the UI when they just make use of services internally.

Performing administrative configuration of Drupal entities (web service descriptions in our case) can be considered a common use case in Drupal. Many Drupal modules need to accomplish a similar task of managing entities in the UI; it seems to be reasonable to share code that is the same among them. The idea is to build a generic entity administration UI foundation in the Entity API module that modules can use, extend and override for their entities. This was realized with the concept of a basic UI controller class, that has default implementations for the menu system (URL paths to entity UI pages), an overview table where all entities (e.g. web service descriptions) are listed

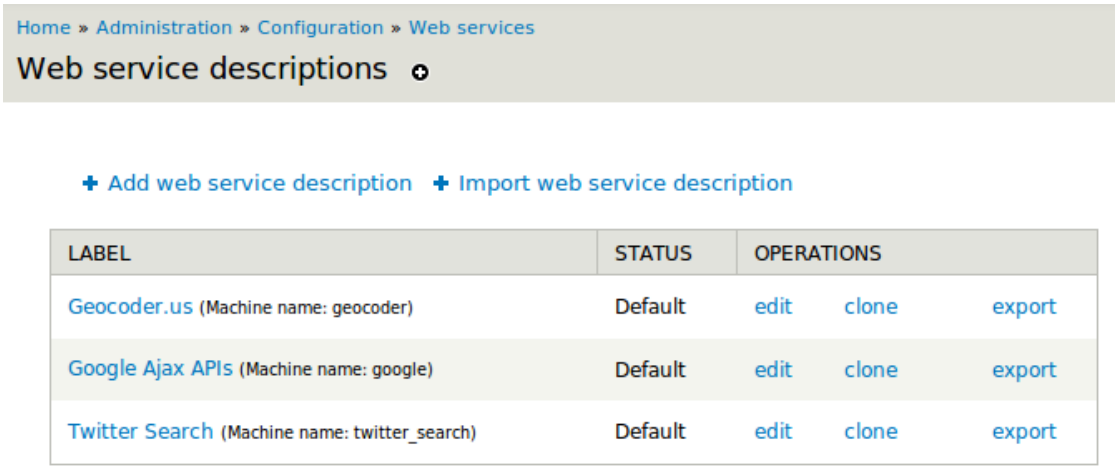


Figure 4.8: Screenshot of the web service client overview UI.

and simple forms to delete or import/export an entity. Complex forms like the entity edit form are entity type dependent and have to be implemented by the third party modules themselves. The documentation how modules can make use of the generic Entity UI can be found on drupal.org[11].

The web service client UI module provides a form to enter properties of the web service description (e.g. a label or the URL) and sub-forms for data types and operations. Figure 4.9 shows an example of such a web service description edit form. Figure 4.8 is a screenshot of the overview page where all web service descriptions in the system are listed. The implementation of these forms was relatively straight forward and consists mainly of pure UI code – the transformation of form values to web service properties was accomplished with the help of the Entity API in very few lines of code. Listing 4.9 shows this simplicity on the form submit function that leverages a convenient API function to map form values to entity object properties.

```
/**
 * Submit callback of the web service description form.
 */
function wsclient_service_form_submit($form, &$form_state) {
  $service = entity_ui_form_submit_build_entity($form,
     $form_state);
  $service->save();
  // ... further UI-specific code omitted here.
}
```

Listing 4.9: Submit callback for web service descriptions that leverages the Entity API.

[11] Making use of the Entity admin UI: http://drupal.org/node/1021576

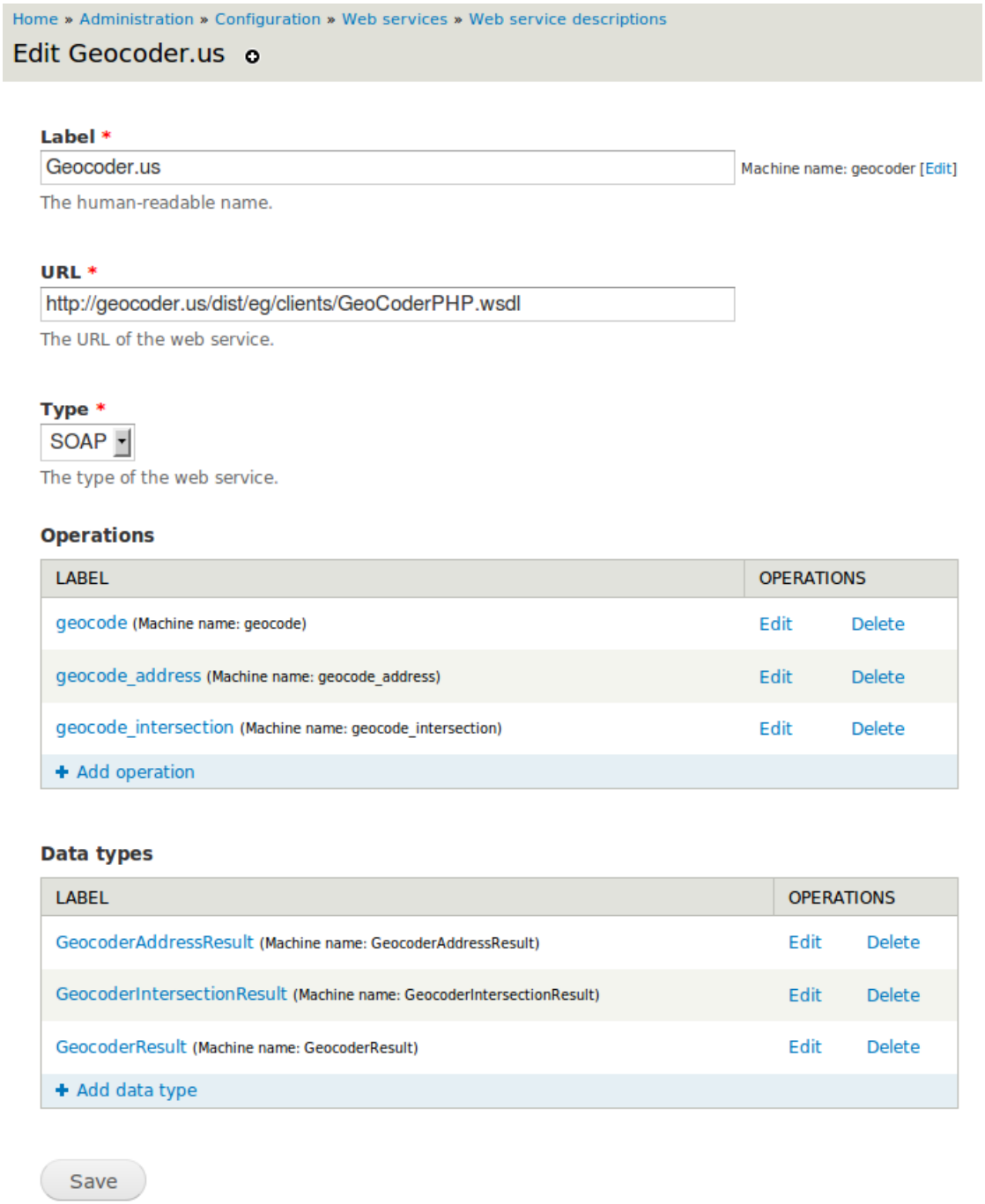


Figure 4.9: Screenshot of the edit form of a web service description.

The form for operations of a web service allows users to enter the important operation name and an arbitrary number of parameters (plus their data type). If a parameter should be a list of the selected type, it can be marked as "multiple". A parameter is per default required (on operation invocation a values has to be present for this parameter), but can also be determined to be optional. A operation result type can be defined, but is not mandatory (also has a "multiple" marker for lists).

There is also a form to enter custom data types that are complex data structures with user-defined properties. They are needed when an operation requires more than a primitive (e.g. a string or an integer) as parameter or result type. In the data type UI form one can define the properties (i.e their name and data type). Data structures can also be nested, so that a custom defined data type is used as property data type of another custom defined data type. With this concept it is possible to specify arbitrary structures with arbitrary property depth.

The UI forms introduced so far apply to all web service types, however endpoint type providing modules can customize and extend that configuration forms. One example is the SOAP submodule (`wsclient_soap`) that immediately imports operations and data types from the WSDL file when the web service description is created (see also section 4.3). Also the submodule for RESTful services (`wsclient_rest`) can add REST-specific settings, e.g. the HTTP request method for an operation or a URL fragment that should be added as operation URL.

WSDL parsing

SOAP web services are typically described with WSDL (see section 2.2) and the information is exposed at a well-known location along the service. It contains metadata about what operations the service offers, how they can be used (e.g. parameter data types) and where they can be accessed (endpoint information). So it seems reasonable that users do not have to enter that information for the web service client module, but read it automatically from those WSDL files that are available anyway.

For the implementation of such a WSDL parser we considered two options: either parse the XML with the PHP SimpleXML extension[12] or use the `SOAPClient` class of the PHP SOAP extension that provides methods to retrieve information about operations and data types. We decided to go with the latter approach, for the following reasons:

- No XML parsing effort. We do not have to care about the WSDL structure details and how that is mapped to the web service client operation and data type concept. That reduces the code size for this functionality tremendously and therefore makes it less error-prone.

[12]PHP SimpleXML extension: http://php.net/manual/de/book.simplexml.php

- Compatibility. We use `SOAPClient` to invoke web services, so we also use it to tell us what it actually can invoke. Incorrectly formatted WSDL files might result in different operations when parsing them separately, which we can avoid that way.
- WSDL versions. Although PHP SOAP only supports WSDL 1.1 at the moment, we can expect that future releases will also support newer WSDL versions, when they become popular. Thus we can ignore the rarely used WSDL 2.0 completely for now and rely on the PHP SOAP interface that will be adapted to new standards.

`SOAPClient` has two methods that can be used:

- `SoapClient::__getFunctions()` : Returns a list of service operations with their parameter types and result type. That operation details can be mapped to the web service descriptions used in the wsclient module (i.e. the operation name, parameter names and types, result type; see figure 4.1 for the targeted properties). Unfortunately the metadata for one operation is concatenated in a string, which has to be tokenized in order to extract all details separately.
- `SoapClient::__getTypes()` : Returns a list of complex data type structures with their properties that are used in the service operations. As for the operations, this information can be transfered to the data types in the wsclient web service descriptions. Again, each type definition is concatenated in a string and must be disassembled (not a difficult task as the pattern is simple).

Besides tokenizing the operation and data type string definitions we must also map primitive data type names to the internally used data type names, e.g. “string” is called “text” or “float” is “decimal” in the Entity API. The whole conversion is done by simple functions that return suitable information arrays for the web service descriptions. With the help of that functions we can provide a `initializeMetadata()` method in our SOAP endpoint class that constructs service operations and data types in the web service description. It can be used when a new SOAP service is created (e.g. in the UI) to auto-populate the metadata or for changed service definitions to override the existing metadata information.

All in all this approach works well in most cases, however there is one little shortcoming: lists in very complex nested data types cannot always be detected. We consider this a minor drawback that will not affect most services; however it can be easily corrected in the UI after a WSDL file was parsed.

Export

We elaborated in the analysis (section 4.1) that JSON is the preferred export format for our web service descriptions. As for the user interface implementation we can reason

about where to realize the export/import functionality: in the web service client module or might it be useful as generic solution in the Entity module? If we consider the use case of exporting an entity it seems obvious that this is indeed a standard feature for many entity providing modules. Therefore we developed import and export methods for the standard entity controller that converts an entity object to/from JSON. Also two API functions were added (`entity_export()` and `entity_import()`) that do the transformations for any entity type. For exporting entity objects are first converted to arrays and then JSON encoded, for importing the JSON string is converted back to an array and then processed in the create method of the entity controller. Listing 4.10 shows an example of such a JSON encoded export string. The "token", "statusKey" and "rdf_mapping" properties refer to entity API specific details.

The export/import functionality can also be used from the UI, links for exporting are available for each web service description. The JSON export is shown on the export page and can be copied and pasted into the import form of another Drupal site. This simple mechanism allows sharing of web service descriptions between independent Drupal sites.

Features export

The simple Entity API export focuses on single entities and does not take possible dependencies to other web service descriptions into account, e.g. a web service description may use a data type from another description in an operation. Obviously the web service description will not work without its dependency – this is where the Features module[13] is needed. Features bundles exportable items (e.g. entities), checks for their dependencies, exports them and creates the source code for a module that contains the export including dependencies. This "feature module" can be transfered to another Drupal instance and upon activation the exported items are available there.

The Entity API module already has an integration with Features, but it lacks a fine grained data type dependency resolution for web service descriptions. We can easily add that by overriding the `EntityDefaultFeaturesController` class with our own where we check for dependencies upon export. Features uses a piping mechanism for the export items where dependencies can be added to that pipe when an item is processed. We need to check for two kind of dependencies:

- *Data types*: Does a web service description use data types from other service descriptions? If so, add that dependencies to the list of exported items.
- *Module dependencies*: What endpoint type is a web service description using (e.g. REST or SOAP)? Add the module that provides that endpoint to the list

[13] Features module: `http://drupal.org/project/features`

of module dependencies (otherwise the service can not be used because of the missing endpoint implementation).

With that advanced export capabilities in place we have accomplished a reliable import/export functionality for web service descriptions.

```
$service = wsclient_service_load('twitter_search');
$export = $service->export();
// Now follows the content of $export (JSON).
{
  "settings" : [],
  "operations" : { "search" : {
      "label" : "Search",
      "parameter" : { "q" : { "type" : "text", "label" : "
         Search text" } },
      "result" : { "type" : "wsclient_twitter_search_result",
          "label" : "Search
result" }
    }
  },
  "datatypes" : {
    "result" : {
      "label" : "Search result",
      "property info" : { "results" : { "type" : "list\
         u003ctweet\u003e",
"label" : "Tweet list" } }
    },
    "tweet" : {
      "label" : "Tweet data",
      "property info" : { "text" : { "type" : "text", "label"
          : "Tweet text" } }
    }
  },
  "name" : "twitter_search",
  "label" : "Twitter Search",
  "url" : "http:\/\/search.twitter.com\/search.json",
  "type" : "rest",
  "token" : "CZ4spciv-QUotnhY8ANkZbbHLtHMpLrjwbMINvukH7E",
  "authentication" : null,
  "statusKey" : "status",
  "rdf_mapping" : []
}
```

Listing 4.10: Example JSON export of a web service description.

Chapter 5

Automatic translation use case

I have no dress except the one I wear every day. If you are going to be kind enough to give me one, please let it be practical and dark so that I can put it on afterwards to go to the laboratory.
– Marie Curie, instructions regarding a proposed gift of a wedding dress for her marriage to Pierre in July 1895.

As a proof of concept and to bring the developments into practical use, a use case that facilitates the power of the web service client module will be implemented. This chapter will outline the details of an automatic translation workflow that is used to retrieve English translation suggestions for a German taxonomy vocabulary in Drupal.

The goal is to invoke multiple translation web services to get a range of English translations and to rank them with the help of a machine learning component. This machine learning component is an external software entity that can be accessed via a web service interface. It processes all translation suggestions and returns a rank per translation to indicate the likeliness of being a correct and suitable translation. Finally the translation with the best rank can be selected and stored in the Drupal vocabulary translations. Figure 5.1 shows an overview of how an example term is translated and ranked with the help of external web services.

5.1 Requirements

To carry out all parts of the workflow we need to fulfill some requirements. First, we need to identify translation web services that can deliver suitable German to English translations for our vocabulary. Second, we need to negotiate an interface to the machine learning component that is capable of exchanging German terms and their possible English translations.

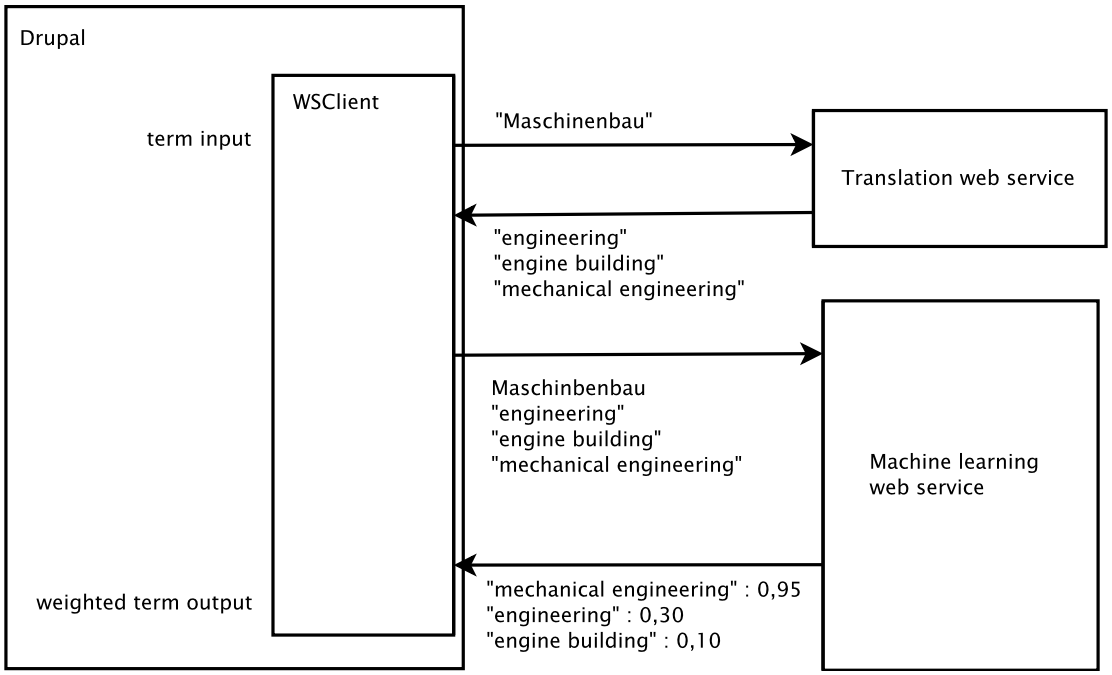


Figure 5.1: Example term translation and the involved web service calls.

Translation web services

The German vocabulary targeted for translation contains terms that are nouns and typically one or two words long. After researching and investigating various translation services that are free to use and publicly available, we could come up with four services that fit to our use case requirements:

- **Google dictionary**[1]: Google provides an online dictionary that offers translations between many languages including German to English. It returns multiple translation proposals for a term. There is no official web service interface, but there is an unofficial way to access the service via HTTP/REST[2].
- **Yahoo Babel Fish**[3]: Babel Fish is one of the oldest Internet translation services and was purchased by Yahoo some years ago. It focuses mainly on translating text or whole web sites and returns only a single result. However, it also works

[1] Google dictionary: http://www.google.com/dictionary

[2] On Google's Unofficial Dictionary API: http://googlesystem.blogspot.com/2009/12/on-googles-unofficial-dictionary-api.html

[3] Yahoo Babel Fish: http://babelfish.yahoo.com

with single words and provides a decent output. Unfortunately there is no machine readable API available that can be accessed as either SOAP or REST web service.

- **dict.cc**[4]: This service offers community driven translations, mostly from German to English. Users of the site can contribute translations and verify other proposed translations. The service returns multiple results in a dictionary style. As for Yahoo Babel Fish there is no machine-readable web service interface available.
- **MyMemory**[5]: Another community powered online service that works with the help of contributed user translations. Similar to Yahoo Babel Fish it is also specialized on translating whole sentences or texts, but also delivers at least one reasonable result for single terms. Meanwhile MyMemory offers a SOAP and REST web service API[6], which was not available at the time of the implementation of this use case.

As we see there are a couple of services that lack a proper web service interface, so we need to extract the result data with the help of a wrapper.

Web data extraction with dapper.net

For accessing the translation services that do not have a web service API we need a conversion tool that allows data extraction from the result pages that contain the translated terms. While web data extraction is an interesting and well-established research field itself with many different approaches [LRNdST02] [ZNW+06], we simply make use of dapper.net[7], an online tool for web scraping. It allows users to create so called *Dapps* that are configurations for specific web sites where input and output variables can be selected from the targeted web sites. In our case the input variables are the German term and the translation direction (German to English), and the output variable is the set of translation results displayed on the site. After that setup the Dapp is ready and is exposed as REST web service on dapper.net. Now our Drupal site (e.g. the web service client module) is able to use the Dapp's web service as gateway to retrieve the translation results from the translation services. There are different formats available for those results, e.g. XML or JSON. Figure 5.2 illustrates the application flow around dapper.net for the dict.cc Dapp example.

We have successfully configured and tested Dapps for dict.cc, MyMemory and Yahoo Babel Fish. The data extraction worked very well in most test cases, however not always perfect as some data items got lost on edge cases. The translation Dapps are

[4] dict.cc: http://www.dict.cc
[5] MyMemory: http://mymemory.translated.net
[6] MyMemory API: http://mymemory.translated.net/doc/spec.php
[7] Dapper.net: http://open.dapper.net

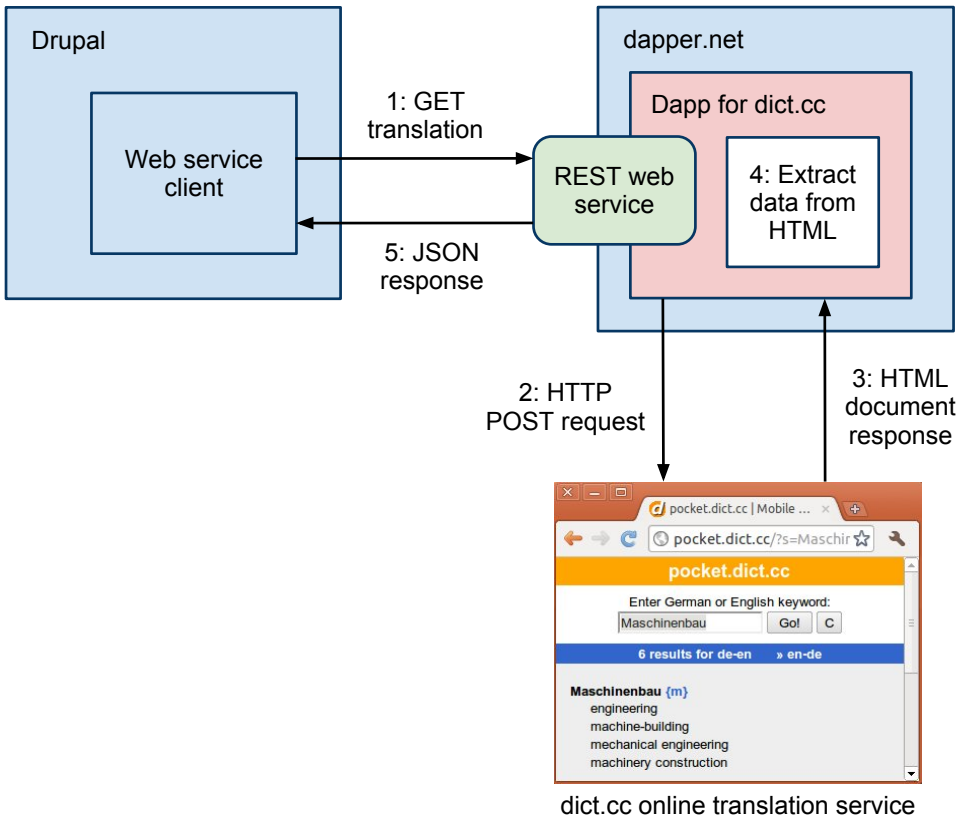


Figure 5.2: Dict.cc translation service wrapped with the dapper.net data extraction tool.

available online at dapper.net[8]. The process of creating a Dapp is a simple task and can be quickly accomplished by selecting the target site, specifying form parameters and choosing result sections to be extracted and returned.

Machine learning component

The evaluation of the collected translations is done by an external machine learning component that is accessed via a web service interface, too (SOAP). It is implemented based on the WEKA[9] algorithms and was setup and configured by Alexander Seewald for our needs. I will not go into the details of machine learning here, as this is clearly out

[8]Dapps used to wrap translation services:
http://open.dapper.net/user-dapps.php?userId=51695

[9]WEKA: http://www.cs.waikato.ac.nz/ml/weka/

of scope for this thesis (a paper about this application is to be released). Important for us is only the interface how translations are sent to the component and how the translation scores are received. We provide the following information when calling the machine learning web service:

- The original German term, the term description, the term synonyms and the parent terms in the Drupal taxonomy.
- The set of translation items which include the web service origins (which services returned that particular translation) and the position/rank within the service results (e.g. a term on the first place might indicate a more appropriate translation). Additionally back translations to German are appended that might help in determining the score of a translation.

The data for a translation is wrapped in complex SOAP data structures and then transmitted with the help of the web service client module. The returned score is saved with the translation suggestion and after evaluating all translations they can be sorted according to the score. In the end the results are presented in the Drupal taxonomy user interface where one can select the translation that should finally be applied as correct one.

5.2 Workflow building

For the realization of the whole translation workflow we realized a small custom Drupal module that uses the web service client module and the Drupal API. At the time of building this workflow the WSClient user interface was not ready yet, so the web service descriptions were created in code. Here is a summary of the steps that were necessary to execute the automatic translation:

1. *Service definitions*. Before working with the translation services and the ranking service we had to specify the operations and the data types that are involved. We also implemented small test cases that were used to check that each service correctly works.
2. *Translation storage*. Received translations are stored in a separate vocabulary and a term reference field is added that points back to the original German term in the source vocabulary. Fields for all the properties of a translation (e.g. web service origin(s) or score) have to be prepared for storing all related information.

3. *Process queue*. To be able to translate vocabularies with many terms, we made use of the Drupal queue system[10]. It is designed to process an arbitrary amount of items in batches so that no PHP execution timeouts are reached and it guarantees that every item (every term in this case) is processed. The scheduling of those batches is done with the Drupal cron system[11] which can execute certain tasks periodically.

4. *Translation collection*. A worker function invokes all translation services for a single German term, saves the results and calls the services again to back-translate each suggestion, which are also saved. The back-translations are used to gain additional information about the term and to further determine the correctness of a translation. All translations are cleaned up (white spaces are stripped off, conversion to all lower case words) and translations that contain special characters are filtered out.

5. *Translation ranking*. The German target term and its corresponding set of translations are then passed to the WEKA web service one by one. WEKA in turn computes a score for this translation. The score is added to the saved translated term. This is repeated for all translation suggestions.

6. *WEKA feedback*. After all terms have been processed and the translations are available in order of their ranking, a human administrator can choose the correct translation. The human feedback can also be shared with the WEKA machine learning component which learns from correct and incorrect translations. WEKA benefits from this online learning to improve future translation rankings.

5.3 Results

Starting from a German vocabulary with more than 1.800 terms the four translation services were invoked several times per term. Not only the English translations were retrieved, but also back translations to German were collected. All this resulted in more than 40.000 translations including the back translations. The scheduling of translation batches took 24 hours until all terms were processed and the web services invocations were completed.

The suggested translations were inspected by a human for quality assurance, also to select the finally correct translation and move it to the English target vocabulary. The methodology proved to be reliable, we found that in 90% of all cases the correct translation had the best score and was ranked first place by the machine learning component.

[10] Drupal queue documentation:
http://api.drupal.org/api/drupal/modules--system--system.queue.inc/group/queue

[11] Drupal cron jobs: http://drupal.org/cron

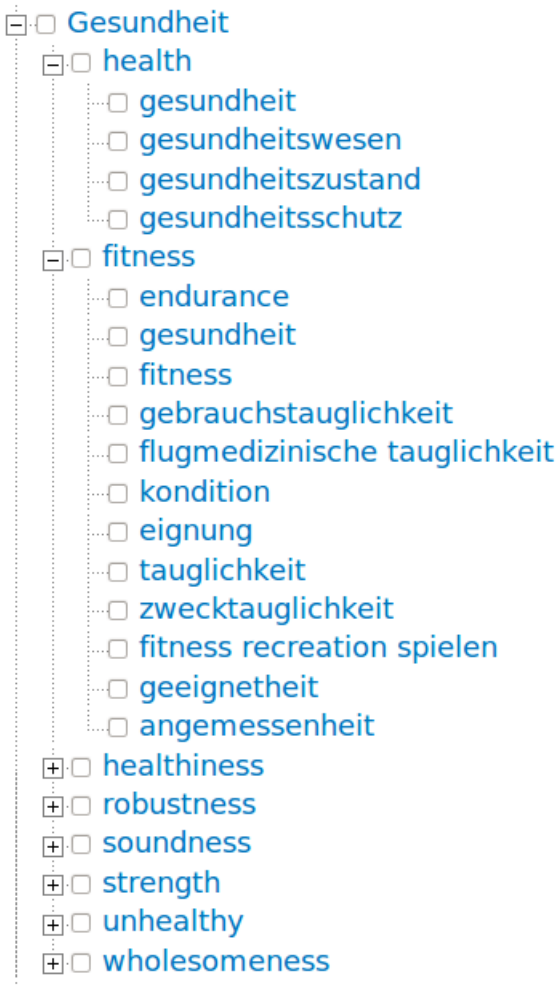


Figure 5.3: Translation tree for the German term "Gesundheit" with back translations at the second level. The terms are ordered according to the WEKA scores.

Terms with a special meaning (e.g. abbreviations or proper names) were also translated, but the results were ignored as the term itself could be used untranslated in English as well in most cases. Figure 5.3 shows an example of the translations that were received (this screenshot was taken from the Taxonomy Manger[12] user interface we used to examine the terms and to administer them).

[12]Taxonomy Manager: http://drupal.org/project/taxonomy_manager

CHAPTER 6

Related work

One is not born, but rather becomes, a woman.
– Simone de Beauvoir

This chapter will establish connections to other projects and research areas that are interesting for our work. This thesis is about consuming web services and combining them in workflows, from a client-side point of view. I will briefly introduce the opposite point of view (the server side), i.e. how web services can be provided within Drupal. Furthermore I will discuss other web service composition approaches.

6.1 Web service providers in Drupal

Providing web services in Drupal means to answer incoming requests not with the standard HTML page generation, but to process the special web service requests and to respond appropriately. Drupal offers the flexibility for modules to take part in many aspects of handling a request so that a web service providing module can be implemented without modifying any Drupal core components. Therefore the development of such a module can be completely encapsulated and does not influence any other independent module or functionality. I will describe two relevant modules that aim to provide generic web services.

Services module

The services module[1] has a long development history and supports a broad range of web service types (e.g. SOAP, XML-RPC, REST etc.) and formats (e.g. JSON, XML, etc.). It is a stable and matured implementation that offers an endpoint system, where

[1] Services module: http://drupal.org/project/services

resources can be exposed at certain URL paths. It also offers integration for authentication systems (e.g. OAuth) and provides hooks for other modules that want to add resource types or endpoint types. Services has developed towards the REST principles (see also chapter 2.2), but does not strictly enforce them (this is necessary to incorporate the different web service types, e.g. SOAP services do not really fit to the resource oriented architecture). Standard resources in Drupal (e.g. nodes or users) are described and are available in various representations out of the box. The module also comes with an UI where endpoints can be configured and resources can be assigned to them.

RESTful Web Services module

A relatively new project is the RESTful Web Services module[2] that has been created by myself and Wolfgang Ziegler. It emerged from the need of using the Entity API module for providing any Drupal entity as web service resource. The Services module has currently no generic entity support, and integrating that functionality did not seem feasible as we also had some other important design goals in mind. The differences to Services are that entities are automatically exposed as resources, there is no endpoint concept as resources are always live on a default URL path, REST principles are strictly enforced (no support for message-oriented service types like SOAP, XML-RPC etc.) and authentication is ignored and must be achieved on another abstraction level. Representation formatters can make use of the resource metadata information, so that also semantically important connections can be incorporated into the representation (e.g. relevant for RDF formatters or to convert id properties to REST conforming URL references). More information about this module is outlined in a blog post by Wolfgang Ziegler[3].

6.2 WS-BPEL composition projects

WS-BPEL (see also chapter 2.3) is a description standard for the orchestration of mostly classical SOAP-based web services. For building these compositions several IDE plug-ins and graphical tools are available, for example the Eclipse BPEL Designer Project[4], the ActiveVOS platform[5] or the Oracle BPEL Process Manager[6] [Lou08]. They assist developers by providing graphical elements that represent BPEL language items which can be linked together. This is very similar to the workflow building we do with the web service client and Rules module. In both cases users (developers) construct mul-

[2] RESTful Web Services module: http://drupal.org/project/restws

[3] "Restful web services in Drupal 7" blog post: http://wolfgangziegler.net/node/14984

[4] Eclipse BPEL Designer Project: http://www.eclipse.org/bpel/

[5] ActiveVOS: http://www.activevos.com/

[6] Oracle BPEL Process Manager: http://www.oracle.com/technetwork/middleware/bpel/overview

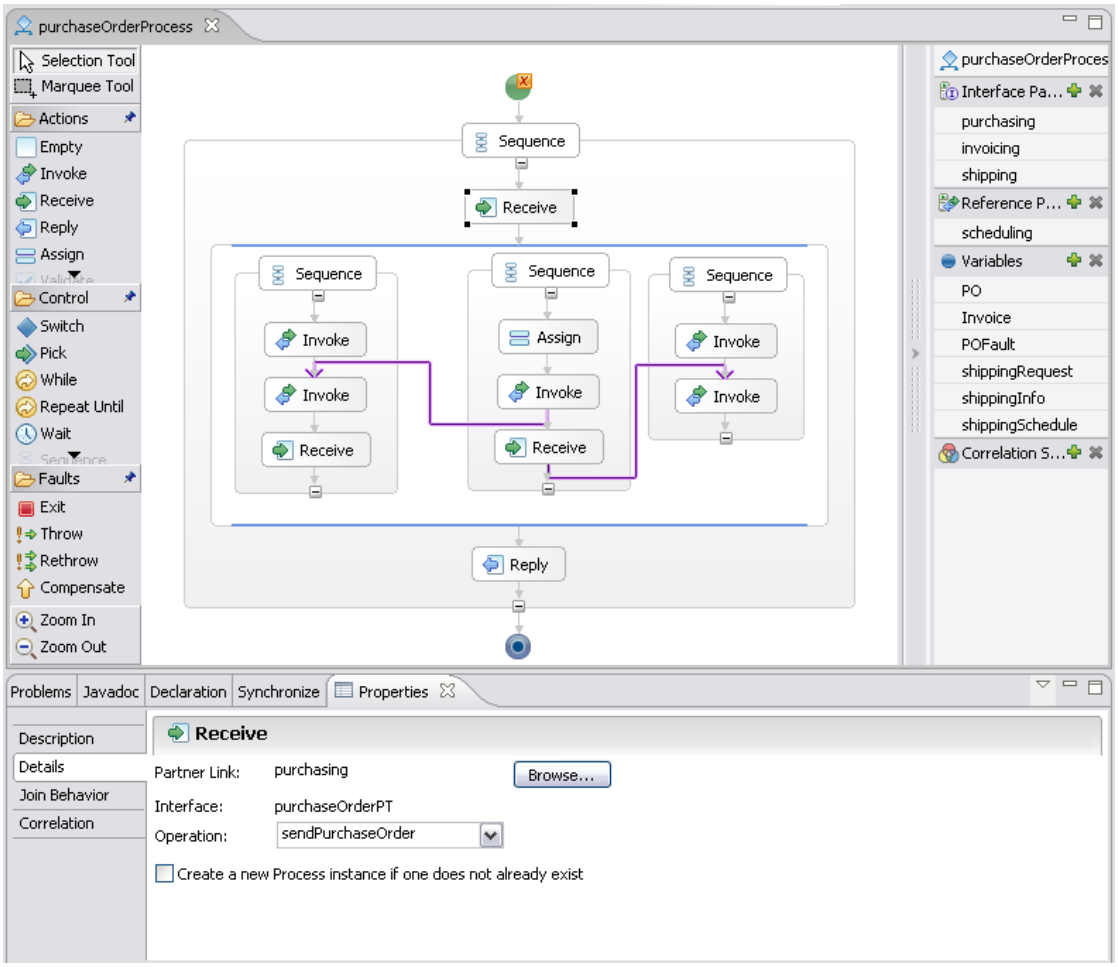


Figure 6.1: Screenshot of the Eclipse BPEL Designer project.

tiple web service invocations that serve a more complicated use case. The differences are that while BPEL processes can be expressed with more language features and are therefore more complex, a workflow with Rules is more limited and simpler, but also integrates deeper with the Drupal environment (e.g. other arbitrary Rules actions can be used in the workflow, besides web service calls). A consequence is that those Rules workflows are tied to a Drupal execution environment, while BPEL processes can be exported to XML representations that can be executed in any BPEL implementing server application. Figure 6.1 shows an example of the graphical composing screen of the Eclipse BPEL Designer project.

Another difference is that BPEL specifically targets business processes and SOAP web services with strict service contracts; our work is more lightweight and also addresses RESTful web services. This is important for integration with modern web 2.0

applications and adds flexibility when incorporating web services to typical workflows in a content management system like Drupal (see also chapter 5 for our use case application).

6.3 Web services in other content management systems

Besides Drupal there are also other content management systems that integrate with web services in certain ways.

- **Plone**: the Web Services API for Plone[7] is a package that provides an XML-RPC interface for the Plone CMS. It acts as web service provider and exposes Plone resources and also comes with a client library to access Plone via this interface. Documentation can be found online[8].
- **Typo3**: there is a Webservices extension[9] that aims to provide a library to easily expose and consume web services. The project is in an experimental state and does not seem to be active.
- **Alfresco**[10]: This Java-based enterprise CMS offers two separate APIs for remote access, so called "Web scripts"[11] (a RESTful API relying on simple HTTP requests) and a Web Services API[12] (providing SOAP services with WSDL files). Both provide a very detailed and also complex implementation that allows many configuration options and specify fine grained features for remote interaction. Alfresco also implements the Content Management Interoperability Services (CMIS) standard which is maintained by the OASIS [EIOO10] and which defines a domain model as well as bindings so that applications can work universally with a CMS.

Researching this topic revealed that most other systems focus on providing their own resources via a remotely accessible interface, while building a web service client abstraction layer is implemented rarely. This indicates that our work is hard to compare to those other systems, as our goals and use cases are quite the opposite to the usual web service providing approaches. It also means that our work is experimental and unique;

[7] wsapi4plone: https://weblion.psu.edu/trac/weblion/wiki/WebServicesApiPlone

[8] wsapi4plone documentation: http://packages.python.org/wsapi4plone.core/

[9] Typo3 Webservices extension: http://forge.typo3.org/projects/extension-extbase_webservices

[10] Alfresco: http://www.alfresco.com/

[11] Alfresco Web Scripts: http://wiki.alfresco.com/wiki/Web_Scripts

[12] Alfresco Web Services API: http://wiki.alfresco.com/wiki/Alfresco_Content_Management_Web_Services

it will have to prove useful in the future and it will have to justify the architectural considerations and the implementation design.

Chapter 7

Conclusion and Outlook

I think I am justified — though where so many hours have been spent in convincing myself that I am right, is there not some reason to fear I may be wrong?
– Jane Austen

In this last chapter I will recapitulate the work described in this thesis and will point out plans for the future. First we will revisit the goals and objectives from chapter 3.

7.1 Evaluation

To measure the overall success of our work I will compare each objective with the result and outcome from our realization.

Web service client module. We have successfully created a web service client module for Drupal and published it on drupal.org[1]. It fulfills the requirement of being a flexible solution for different web service types and comes with support for SOAP and RESTful web services (including REST-RPC hybrids). We managed to design the module not only for good usability in the user interface, but also created a good developer experience for programmers with a clean API. We embraced the work from Wolfgang Ziegler [Zie10] and implemented a well-founded web service abstraction layer that is extensible and easy to use. The Entity API module[2] helped to solve basic configuration storage needs and simplified the code, so that the module implementation could concentrate on the core features that the module accomplishes.

[1] Web service client module: http://drupal.org/project/wsclient
[2] Entity API module: http://drupal.org/project/entity

Web service composition with Rules. We elaborated on the usefulness of a workflow engine for web service composition and integrated the web service client module with the Rules module[3]. A web service invocation has been realized as action in the Rules Event-Condition-Action system. This integration enables workflow builders to use all the existing events, conditions and actions to combine them with web service calls. We showed that multiple web services can be used in a workflow and we completed Rules with data type actions that solved the problem of transferring and re-assigning data structures between service calls. The language elements of Rules (e.g. conditions, loops, rule sets etc.) provide a powerful tool set to orchestrate web services. We admit that the features of Rules are limited compared to composition languages like WS-BPEL, but Rules has the advantage of a deep integration to the Drupal platform and easily satisfies workflow needs in a CMS.

An automatic translation use case. The practical usefulness of the web service client module was verified with the task of collecting different translations from web services and to combine the results with another web service. The web service client module proved to be robust, easy to use and worked out flawlessly when implementing the workflow around the use case. As a by-product we investigated web data extraction techniques to turn web applications into accessible web services (see chapter 5 for details).

Web service integration without programming effort. The web service client module provides a user interface that allows site administrators to specify web service descriptions. Web service operations and complex data types can be defined in the UI, so that no code has to be written in order to setup a web service connection. The other part of invoking web services is accomplished with the Rules user interface, where web service operations can be added as actions that are executed when the rule is triggered. Of course site administrators still have to be familiar with web services and how they can be specified, but no programming effort is needed to build web service client descriptions.

Automatic WSDL parsing. We managed to retrieve metadata of SOAP services from their WSDL files with the help of the PHP SOAP extension. This is a huge relief for site administrators that now do not have to manually enter all details of the service. Operations and data types of a service are extracted automatically upon creation of the service description.

Sharing of exportable web service descriptions. We realized a comprehensive import/-export solution that allows transferring of web service descriptions to other Drupal

[3]Rules module: http://drupal.org/project/rules

installations. The Entity API module was of great help again, which we extended to provide JSON formatted exports for any entity type. For complex exports of web service descriptions that have dependencies to others, we introduced an integration for the Features module[4]. It is capable of resolving the dependencies and bundles the exports as Drupal module.

7.2 Future work

Although our objectives have been satisfied, some small details in the implementation remain open. There is still some work to do to complete all goals of the web service client project.

- RESTful web services have not been implemented and tested completely, because we dealt mostly with GET operations that retrieve data from a service. To have full CRUD support in the web service client REST module we will have to realize create, update and delete operations as well, which has not been done yet. However, the implementation of PUT, POST and DELETE requests should be an easy task and straight forward, as the REST module does not require big changes and is prepared for such additions.

- The user interface does not take details for different service types into account, i.e. it only handles the generic type-independent web service description. Ideally service type providing modules like the REST module would extend the user interface to also include their settings (e.g. the operation URL for a RESTful service). These settings of course work already in the developer API and only a mapping to elements in the UI pages is missing.

- Sometimes it may be convenient to quickly test a web service operation from the UI, which is currently only possible by creating a test Rules configuration and executing it manually. A direct integration of such an execution into the web service operation UI would be less cumbersome, but has not been realized yet.

- Extracting SOAP service metadata from WSDL files does not work for edge cases with very complex data types (nested lists) at the moment. The fact that the PHP SOAP extension lacks a perfect detection does not affect most services; however, implementing a custom XML parser could solve that problem, but would mean quite some programming effort.

- A minor issue is that if a SOAP service WSDL file changes, then the internal web service description must be discarded and must be newly created in order to do

[4]Features module: http://drupal.org/project/features

the automatic metadata extraction from the WSDL again in the UI. This could easily be solved by providing a button to re-parse the WSDL, but has never been a priority during the development of the module.

- Authorization and authentication has only been implemented in the form of HTTP basic authentication. It would be interesting to also work with trending concepts such as OAuth [HLRH11] which is very popular among RESTful web services.

7.3 Summary

All together we managed to release a web service module for Drupal 7 that has a solid foundation and that incorporates modern design patterns. It is based on the important concept of entities and the Entity API that will shape the future of Drupal. We leveraged the entity system for the storage of web service descriptions and we were able to completely avoid writing any database related code besides the initial database schema definition. Furthermore we improved the Entity module in a way so that it is not only useful for the web service client module but potentially also for any third-party module that works with entities. Common basic features like an administration UI or import/-export functionality were developed to be generically usable for any entity.

A web service description is the internal abstraction model that was chosen during the implementation of this project. All web service types can be considered as a set operations and a set of data types that are used in the operations. Parameters and return variables form the signature of an operation. We showed that this concept can be successfully applied to SOAP and RESTful web service types. Web service descriptions are extensible and the particular service type can add service specific configurations.

Web service composition was explored as research topic and was applied to a real world scenario in Drupal. The Rules module integration of the web service client module is a great opportunity to embed web service invocations in workflows. We have provided a flexible system for multiple web service calls that can be configured completely on Drupal administration pages. Using web services and integrating them into Drupal is now a much simpler task and can be accomplished without writing code.

The original work of Wolfgang Ziegler [Zie10] on web services has been embraced and has been developed to a mature solution. The relevance of the web service client module has not only been outlined theoretically, but has also been proven on the translation workflow use case and several other practical applications. As the implementation is licensed as free and open source software and is published on drupal.org, the Drupal community and others can take part in any further development and can use the module for their own needs. We look forward to future adoptions and how our approach will influence general web service integration in Drupal.

Appendix A

Acronyms

AJAX Asynchronous JavaScript + XML

API Application Programming Interface

CCK Content Construction Kit

CMIS Content Management Interoperability Services

CMS Content Management System

CRUD Create Read Update Delete

EMML Enterprise Mashup Markup Language

FTP File Transfer Protocol

GNU GNU's Not Unix

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IDE Integrated Development Environment

IT Information Technology

JSON JavaScript Object Notation

OASIS Organization for the Advancement of Structured Information Standards

PDO PHP Data Objects

RDF Resource Description Framework

REST Representational State Transfer

RFC Request for Comments

ROA Resource Oriented Architecture

RPC Remote Procedure Call

RSS Really Simple Syndication

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

UDDI Universal Description, Discovery and Integration

UI User Interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

W3C World Wide Web Consortium

WEKA Waikato Environment for Knowledge Analysis

WADL Web Application Description Language

WSDL Web Services Description Language

WS-BPEL Web Services Business Process Execution Language

WS-CDL Web Services Choreography Description Language

WWW World Wide Web

XHTML eXtensible HyperText Markup Language

XML eXtensible Markup Language

XSD XML Schema Definition

Appendix B

Index

List of Figures

List of Tables

Listings

Appendix C

Bibliography

[AAM06] Asif Akram, Rob Allan, and David Meredith. Best practices in web service style, data binding and validation for use in data-centric scientific applications. August 2006.

[All09] Open Mashup Alliance. OMA EMML Specification 1.0, 2009. http://www.openmashup.org/omadocs/v1.0/index.html.

[BB08] Robert Battle and Edward Benson. Bridging the semantic web and web 2.0 with representational state transfer (REST). *Web Semant.*, 6(1):61–69, 2008.

[BLFM98] T. Berners-Lee, R. Fielding, and L. Masinter. Rfc 2396: Uniform resource identifiers (URI): Generic syntax, aug 1998. http://tools.ietf.org/html/rfc2396.

[Bru09] Alexander Bruckner. Tool supported workflow integration of restful web services. Master's thesis, Vienna Univserity of Technology, 2009.

[Cro06] D. Crockford. The application/json media type for javascript object notation (JSON), jul 2006. http://tools.ietf.org/html/rfc4627.

[dc09] Drupal documentation community. Drupal programming from an object-oriented perspective. http://drupal.org/node/547518, 2009.

[Del07] Daniel B. Delgado. Inspiring teamwork & communication with a content management system. In *SIGUCCS '07: Proceedings of the 35th annual ACM SIGUCCS fall conference*, pages 55–59, New York, NY, USA, 2007. ACM.

[DS05] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *Int. J. Web Grid Serv.*, 1(1):1–30, 2005.

[EIOO10] David Choy Emc, Al Brown Ibm, Ryan Mcveigh Oracle, and Florian Muller Opentext. Oasis content management interoperability services (cmis) tc, 2010. http://docs.oasis-open.org/cmis/CMIS/v1.0/cd07/cmis-spec-v1.0.html.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1, 1999. http://tools.ietf.org/html/rfc2616.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. Chair-Taylor, Richard N.

[FT00] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 407–416, New York, NY, USA, 2000. ACM.

[GN02] M. Grossniklaus and M. C. Norrie. Information concepts for content management. pages 150–159, 2002.

[Gre07] Joe Gregorio. Do we need WADL? Blogpost, 2007. http://bitworking.org/news/193/Do-we-need-WADL.

[HLRH11] E. Hammer-Lahav, D. Recordon, and D. Hardt. The oauth 2.0 authorization protocol. Technical report, http://tools.ietf.org/html/draft-ietf-oauth-v2-12, 2011.

[Jaz07] Mehdi Jazayeri. Some trends in web application development. In *FOSE '07: 2007 Future of Software Engineering*, pages 199–213, Washington, DC, USA, 2007. IEEE Computer Society.

[JE+07] Diane Jordan, John Evdemon, et al. Web services business process execution language version 2.0, OASIS standard, 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[LHSL07] Xuanzhe Liu, Yi Hui, Wei Sun, and Haiqi Liang. Towards service composition based on mashup. pages 332–339, jul. 2007.

[LLSL08] Qing Li, Rynson W. H. Lau, Timothy K. Shih, and Frederick W. B. Li. Technology supports for distributed and collaborative learning over the internet. *ACM Trans. Internet Technol.*, 8(2):1–24, 2008.

[Lou08] Panagiotis Louridas. Orchestrating web services with bpel. *IEEE Software*, 25:85–87, 2008.

[LRNdST02] Alberto H. F. Laender, Berthier A. Ribeiro-Neto, Altigran S. da Silva, and Juliana S. Teixeira. A brief survey of web data extraction tools. *SIGMOD Rec.*, 31(2):84–93, June 2002.

[Mer09] Duane Merrill. Mashups: The new breed of web app. IBM DeveloperWorks, 2009. http://www.ibm.com/developerworks/xml/library/x-mashups.html.

[N+10] Károly Négyesi et al. Field API tutorial, 2010. http://drupal.org/node/707832.

[NPRI09] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of json and xml data interchange formats: A case study. In *CAINE*, pages 157–162, 2009.

[O'R05] Tim O'Reilly. What is web 2.0?: Design patterns and business models for the next generation of software, September 2005. http://oreilly.com/web2/archive/what-is-web-20.html.

[Ove07] Hagen Overdick. The resource-oriented architecture. *Services, IEEE Congress on*, 0:340–347, 2007.

[Pap08] M. P. Papazoglou. *Web services: principles and technology*. Pearson Prentice Hall, 2008.

[Pau08] Cesare Pautasso. Bpel for rest. In *BPM '08: Proceedings of the 6th International Conference on Business Process Management*, pages 278–293, Berlin, Heidelberg, 2008. Springer-Verlag.

[Pau09] Cesare Pautasso. Restful web service composition with bpel for rest. *Data & Knowledge Engineering*, 68(9):851–866, 2009. Sixth International Conference on Business Process Management (BPM 2008) - Five selected and extended papers.

[PTDL07] MP Papazoglou, P Traverso, S Dustdar, and F Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–+, nov. 2007.

[PZL08] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big"' web services: making the right architectural decision. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 805–814, New York, NY, USA, 2008. ACM.

[RR07] Leonard Richardson and Sam Ruby. *Restful web services*. O'Reilly, 2007.

[SHM08] Derek T. Sanders, . J. A. Hamilton, Jr., and Richard A. MacDonald. Supporting a service-oriented architecture. In *SpringSim '08: Proceedings of the 2008 Spring simulation multiconference*, pages 325–334, San Diego, CA, USA, 2008. Society for Computer Simulation International.

[Sim05] Doug L. Simpson. Content for one: developing a personal content management system. In *SIGUCCS '05: Proceedings of the 33rd annual ACM SIGUCCS fall conference*, pages 338–342, New York, NY, USA, 2005. ACM.

[Sta06] Michael Stal. Using architectural patterns and blueprints for service-oriented architecture. *IEEE Software*, 23:54–61, 2006.

[tBBG07] M. ter Beek, A. Bucchiarone, and S. Gnesi. Web service composition approaches: From industrial standards to formal methods. pages 15–15, may. 2007.

[TP02] Aphrodite Tsalgatidou and Thomi Pilioura. An overview of standards and related technology in web services. *Distrib. Parallel Databases*, 12(2-3):135–162, 2002.

[UG98] Tommie Usdin and Tony Graham. Xml: not a silver bullet, but a great pipe wrench. *StandardView*, 6(3):125–132, 1998.

[VW07] John VanDyk and Matt Westgate. *Pro Drupal Development*. Apress, Berkely, CA, USA, 2007.

[W3C04] Web Services Architecture Working Group W3C. Web services glossary, 2004. http://www.w3.org/TR/ws-gloss/.

[Wil10] Erik Wilde. Representational state transfer (REST). Web Architecture lecture slides, UC Berkeley School of Information, 2010. http://dret.net/lectures/web-fall10/rest.

[Z+10a] Wolfgang Ziegler et al. Drupal rules module documentation, 2010. http://drupal.org/node/298476.

[Z+10b] Wolfgang Ziegler et al. Rules developer documentation, 2010. http://drupal.org/node/878718.

[Zie10] Wolfgang Ziegler. Enhanced reacitve rules for drupal. Master's thesis, Vienna University of Technology, 2010. https://more.zites.net/thesis.

[ZNW+06] Jun Zhu, Zaiqing Nie, Ji-Rong Wen, Bo Zhang, and Wei-Ying Ma. Simultaneous record detection and attribute labeling in web data extraction. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 494–503, New York, NY, USA, 2006. ACM.